Quartus II Handbook Version 9.0

Volume 1: Design and Synthesis

101 Innovation Drive
San Jose, CA 95134
www.altera.com

QII5V1-9.0

Copyright © 2009 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

Contents

Chapter Revision Dates xxi

Section I. Design Flows

Chapter 1. Design Planning with the Quartus II Software

Introduction 1-1
Creating Design Specifications 1-2
Device Selection 1-2
Device Migration Planning 1-3
Planning for Device Programming/Configuration 1-4
Early Power Estimation 1-5
Early Power Estimator File 1-5
Early Pin Planning and I/O Analysis 1-6
Creating a Top-Level Design File for I/O Analysis 1-7
Selecting Third-Party EDA Tool Flows 1-8
Synthesis Tools 1-8
Simulation Tools 1-8
Formal Verification Tools 1-9
Planning for On-Chip Debugging Options 1-9
Design Practices and HDL Coding Styles 1-11
Design Recommendations 1-11
Recommended HDL Coding Styles 1-12
Planning for Hierarchical and Team-Based Design 1-13
Flat Compilation Flow with No Design Partitions 1-13
Incremental Compilation with Design Partitions 1-13
Top-Down Versus Bottom-Up Incremental Flows 1-14
Top-Down Incremental Compilation Flow 1-14
Bottom-Up and Team-Based Incremental Compilation Flow 1-15
Mixed Incremental Compilation Flow 1-15
Planning Design Partitions 1-16
Creating a Design Floorplan 1-17
Fast Synthesis and Early Timing Estimation 1-17
Conclusion 1-18
Referenced Documents 1-18
Document Revision History 1-19

Chapter 2. Quartus II Incremental Compilation for Hierarchical and Team-Based Design

Introduction 2-1
Deciding Whether to Use an Incremental Compilation Flow 2-2
Flat Compilation Flow with No Design Partitions 2-2
Quartus II Smart Compilation 2-3
Incremental Compilation Flow with Design Partitions 2-3
Top-Down versus Bottom-Up Compilation Flows 2-6
Quick Start Guide—Summary of Incremental Compilation 2-7
Preparing a Design for Incremental Compilation 2-8
Compiling a Design Using Incremental Compilation 2-9
Deciding which Design Blocks Should Be Design Partitions 2-9
Impact of Design Partitions on Design Optimization 2-11

Partition Statistics Reports . . . 2-11
Partition Timing Reports . . . 2-13
Incremental Compilation Advisor . . . 2-13
Using Partitions with Third-Party Synthesis Tools . . . 2-15
Synopsys Synplify Pro/Premier and Mentor Graphics Precision RTL Plus . . . 2-15
Other Synthesis Tools . . . 2-15
Design Partition Assignments Compared to Physical Placement Assignments . . . 2-15
Creating Design Partition Assignments . . . 2-16
Creating Design Partitions with the Design Partition Planner . . . 2-16
Creating Design Partitions In the Design Partitions Window . . . 2-18
Creating Design Partitions in the Project Navigator . . . 2-18
Creating Design Partitions with Tcl Scripting . . . 2-19
Partition Name . . . 2-19
Setting the Netlist Type for Design Partitions . . . 2-19
Fitter Preservation Level . . . 2-21
Empty Partitions . . . 2-22
Where Are the Netlist Databases Stored? . . . 2-23
What Changes Trigger a Partition's Automatic Resynthesis? . . . 2-23
Determining Which Partitions Are Resynthesized Due to Source Code Changes . . . 2-24
Forcing Use of the Post-Fitting Netlist When a Partition has Changed . . . 2-25
Creating a Design Floorplan with LogicLock Location Assignments . . . 2-25
Taking Advantage of the Early Timing Estimator . . . 2-28
What LogicLock Changes Trigger Refitting? . . . 2-28
Exporting and Importing Partitions . . . 2-29
Quartus II Exported Partition Files (.qxp) . . . 2-30
Bottom-Up Incremental Compilation Summary . . . 2-30
Preparing a Design for Bottom-Up Incremental Compilation . . . 2-31
Creating and Compiling Lower-Level Projects . . . 2-31
Exporting Lower-Level Projects . . . 2-32
Including or Importing Lower-Level Projects into the Top-Level Project . . . 2-32
Performing an Incremental Compilation in the Top-Level Project . . . 2-32
Netlist Types for Imported Partitions . . . 2-33
Exporting a Lower-Level Partition to be Used in a Top-Level Project . . . 2-34
Exporting a Lower-Level Block within a Project . . . 2-35
Using a .qxp File as a Source File in the Top-Level Project . . . 2-36
Importing a Lower-Level Partition Into the Top-Level Project . . . 2-36
Importing Assignments and Advanced Import Settings . . . 2-38
Design Partition Properties after Importing . . . 2-38
Importing Design Partition Assignments Within the Subdesign . . . 2-38
Synopsys Design Constraint Files for the Quartus II TimeQuest Timing Analyzer . . . 2-38
Importing LogicLock Assignments . . . 2-38
Importing Other Instance Assignments . . . 2-39
Importing Global Assignments . . . 2-39
Advanced Import Settings . . . 2-39
Generating Bottom-Up Design Partition Scripts for Project Management . . . 2-40
Project Creation . . . 2-41
Excluded Partitions . . . 2-41
Assignments from the Top-Level Design . . . 2-41
Virtual Pin Assignments . . . 2-42
LogicLock Region Assignments . . . 2-43
Global Signal Promotion Assignments . . . 2-43
Makefile Generation . . . 2-44
Importing SDC Constraints from Lower-Level Partitions . . . 2-45
Creating an .sdc File with Project-Wide Constraints . . . 2-46

Creating an .sdc File with Partition-Specific Constraints 2-47
Consolidating the SDC Files in the Top-Level Design 2-48
Recommended Design Flows and Compilation Application Examples 2-49
Reducing Compilation Time When Changing a Source File for One Partition 2-49
Optimizing a Timing-Critical Partition to Achieve Timing Closure 2-50
Preserving Results for Some Partitions Before Adding Other Partitions 2-51
Debugging Incrementally with the SignalTap II Logic Analyzer 2-52
Implementing a Team-Based Bottom-Up Design Flow 2-53
Performing Design Iteration in a Bottom-Up Design Flow 2-56
Creating Hard-Wired Macros (or Precompiled Design Blocks) for IP Reuse 2-57
Using an Exported Partition to Send a Design without Including Source Files 2-59
Incremental Compilation Restrictions 2-60
Preserving Exact Timing Performance 2-61
When Placement and Routing May Not Be Preserved Exactly 2-61
Using Incremental Compilation with Quartus II Archive Files 2-61
Formal Verification Support 2-61
Importing Encrypted IP Cores in Bottom-Up Flows 2-62
SignalProbe Pins and Engineering Change Management with the Chip Planner 2-62
Linked Partitions Due to SignalProbe Pins or ECO Changes 2-62
Exported Partitions 2-63
SignalTap II Embedded Logic Analyzer in Bottom-Up Compilation Flows 2-64
Logic Analyzer Interface in Bottom-Up Compilation Flows 2-64
Migrating Projects with Design Partitions to Different Devices 2-64
HardCopy Compilation and Migration Flows 2-64
HardCopy APEX and HardCopy Stratix Devices 2-64
HardCopy ASIC Migration Flows 2-65
HardCopy ASIC Stand-Alone Compilations 2-65
Assignments Made in HDL Source Code in Bottom-Up Flows 2-65
Restrictions on Megafunction Partitions 2-65
Register Packing and Partition Boundaries 2-66
I/O Register Packing 2-66
Bottom-Up Design Partition Script Limitations 2-67
Warnings About Extra Clocks Due to Bottom-Up Design Partition Scripts 2-67
Synopsys Design Constraint Files for the TimeQuest Timing Analyzer in Bottom-Up Design Partition Scripts 2-67
Wildcard Support in Bottom-Up Design Partition Scripts 2-67
Derived Clocks and PLLs in Bottom-Up Design Partition Scripts 2-67
Pin Assignments for GXB and LVDS Blocks in Bottom-Up Design Partition Scripts 2-68
Virtual Pin Timing Assignments in Bottom-Up Design Partition Scripts 2-68
Top-Level Ports that Feed Multiple Lower-Level Pins in Bottom-Up Design Partition Scripts 2-68
Scripting Support 2-69
Preparing a Design for Incremental Compilation 2-69
Creating Design Partitions 2-69
Setting Properties of Design Partitions 2-70
Creating Floorplan Location Assignments—Excluding or Filtering Certain Device Elements (Such as RAM or DSP Blocks) 2-71
Generating Bottom-Up Design Partition Scripts 2-71
Command Line Support 2-72
Exporting a Partition to be Used in a Top-Level Project 2-73
Importing a Lower-Level Partition into the Top-Level Project 2-74
Makefiles 2-74
Recommended Design Flows and Compilation Application Examples—Scripting and Command-Line Operation 2-74
Reducing Compilation Time When Changing a Source File for One Partition—Command-Line

Example 2-75
Optimizing the Placement for a Timing-Critical Partition 2-75
Conclusion 2-76
Referenced Documents 2-76
Document Revision History 2-77

Chapter 3. Quartus II Design Flow for MAX+PLUS II Users

Introduction 3-1
Chapter Overview 3-1
Typical Design Flow 3-2
Device Support 3-2
Quartus II GUI Overview 3-3
Project Navigator 3-3
Node Finder 3-3
Tcl Console 3-4
Messages 3-4
Status 3-4
Change Manager 3-4
Setting Up MAX+PLUS II Look and Feel in the Quartus II Software 3-5
MAX+PLUS II Look and Feel 3-5
Compiler Tool 3-7
Analysis and Synthesis 3-7
Partition Merge 3-8
Fitter 3-8
Assembler 3-8
Timing Analyzer 3-9
EDA Netlist Writer 3-9
Design Assistant 3-9
MAX+PLUS II Design Conversion 3-9
Converting an Existing MAX+PLUS II Design 3-9
Converting MAX+PLUS II Graphic Design Files 3-11
Importing MAX+PLUS II Assignments 3-11
Quartus II Design Flow 3-12
Creating a New Project 3-12
Design Entry 3-13
Making Assignments 3-15
Assignment Editor 3-15
Timing Assignments 3-16
Synthesis 3-17
Functional Simulation 3-17
Place and Route 3-19
Timing Analysis 3-20
Timing Closure Floorplan 3-22
Timing Simulation 3-23
Quartus II Simulator Tool 3-23
EDA Timing Simulation 3-24
Power Estimation 3-25
Programming 3-25
Conclusion 3-26
Quartus II Command Reference for MAX+PLUS II Users 3-26
Referenced Documents 3-33
Document Revision History 3-34

Chapter 4. Quartus II Support for HardCopy Series Devices

Introduction . . . 4-1
HardCopy Series Device Support . . . 4-1
HardCopy Series Design Benefits . . . 4-2
Quartus II Features for HardCopy Planning . . . 4-2
HardCopy Development Flow . . . 4-3
Designing the FPGA First . . . 4-4
Designing the HardCopy Device First . . . 4-6
HardCopy Device Resource Guide . . . 4-7
HardCopy Companion Device Selection . . . 4-10
HardCopy Recommended Settings in the Quartus II Software . . . 4-11
Limit DSP and RAM to HardCopy Device Resources . . . 4-11
Enable Design Assistant to Run During Compile . . . 4-12
Timing Settings . . . 4-13
TimeQuest Timing Analyzer . . . 4-13
Setting Up the TimeQuest Timing Analyzer . . . 4-14
Constraints for Clock Effect Characteristics . . . 4-14
Quartus II Software Features Supported for HardCopy Designs . . . 4-16
Physical Synthesis Optimization . . . 4-16
LogicLock Regions . . . 4-16
PowerPlay Power Analyzer . . . 4-16
Incremental Compilation . . . 4-16
HardCopy Utilities Menu . . . 4-17
Companion Revisions . . . 4-18
Compiling the HardCopy Companion Revision . . . 4-19
Comparing HardCopy and FPGA Companion Revisions . . . 4-20
Generate a HardCopy Handoff Report . . . 4-20
Archive HardCopy Handoff Files . . . 4-21
HardCopy Advisor . . . 4-21
HardCopy Design Readiness Check . . . 4-23
Execution of HardCopy Design Readiness Check . . . 4-23
Stratix III Support . . . 4-24
Setting Check . . . 4-24
Summary . . . 4-25
Global Setting . . . 4-25
Instance Setting . . . 4-25
Operating Setting . . . 4-25
I/O Check . . . 4-26
Input Pin Placement for Global and Regional Clock . . . 4-26
PLL Usage Check . . . 4-27
PLL Real-Time Reconfigurable Check . . . 4-27
PLL Clock Outputs Driving Multiple Clock Network Types Check . . . 4-27
PLL with No Compensation Mode Check . . . 4-27
PLL with Normal or Source Synchronous Mode Feeding Output Pin Check . . . 4-27
RAM Usage Check . . . 4-28
Initialized Memory Dependency Testing . . . 4-28
Performing ECOs with Quartus II Engineering Change Management with the Chip Planner . . . 4-29
Migrating One-to-One Changes . . . 4-29
Migrating Changes that Must be Implemented Differently . . . 4-30
Changes that Cannot be Migrated . . . 4-31
Overall Migration Flow . . . 4-31
Preparing the Revisions . . . 4-31
Applying ECO Changes . . . 4-31
Formal Verification of FPGA and HardCopy Revisions . . . 4-32

HardCopy Floorplan View . . . 4-33
Legacy HardCopy Device Support . . . 4-34
Features . . . 4-35
HARDCOPY_FPGA_PROTOTYPE, HardCopy Stratix, and Stratix Devices . . . 4-36
HardCopy Design Flow . . . 4-37
The Design Flow Steps of the One-Step Process . . . 4-38
Compile the Design for an FPGA . . . 4-38
Migrate the Compiled Project . . . 4-38
Close the Quartus FPGA Project . . . 4-39
Open the Quartus HardCopy Project . . . 4-39
Compile for HardCopy Stratix Device . . . 4-39
How to Design HardCopy Stratix Devices . . . 4-39
HardCopy Timing Optimization Wizard . . . 4-40
Tcl Support for HardCopy Migration . . . 4-42
Design Optimization and Performance Estimation . . . 4-42
Design Optimization . . . 4-43
Performance Estimation . . . 4-43
Buffer Insertion . . . 4-46
Placement Constraints . . . 4-46
Location Constraints . . . 4-46
LAB Assignments . . . 4-47
LogicLock Assignments . . . 4-47
Checking Designs for HardCopy Design Guidelines . . . 4-48
Altera-Recommended HDL Coding Guidelines . . . 4-48
Design Assistant . . . 4-48
Design Assistant Settings . . . 4-48
Running Design Assistant . . . 4-48
Reports and Summary . . . 4-49
Generating the HardCopy Design Database . . . 4-49
Static Timing Analysis . . . 4-51
Early Power Estimation . . . 4-51
HardCopy Stratix Early Power Estimation . . . 4-51
Tcl Support for HardCopy Stratix . . . 4-52
Conclusion . . . 4-52
Referenced Documents . . . 4-52
Document Revision History . . . 4-53

Section II. Design Guidelines

Chapter 5. Design Recommendations for Altera Devices and the Quartus II Design Assistant

Introduction . . . 5-1
Synchronous FPGA Design Practices . . . 5-2
Fundamentals of Synchronous Design . . . 5-2
Hazards of Asynchronous Design . . . 5-3
Design Guidelines . . . 5-4
Combinational Logic Structures . . . 5-4
Combinational Loops . . . 5-4
Latches . . . 5-5
Delay Chains . . . 5-5
Pulse Generators and Multivibrators . . . 5-6
Clocking Schemes . . . 5-7
Internally Generated Clocks . . . 5-8
Divided Clocks . . . 5-8

Ripple Counters 5-8
Multiplexed Clocks 5-9
Gated Clocks 5-10
Synchronous Clock Enables 5-11
Recommended Clock-Gating Methods 5-11
Design Techniques to Save Power 5-12
Checking Design Violations Using the Design Assistant 5-13
Quartus II Design Flow with the Design Assistant 5-13
The Design Assistant Settings Page 5-15
Message Severity Levels 5-15
Design Assistant Rules 5-16
Summary of Rules and IDs 5-16
Design Should Not Contain Combinational Loops 5-17
Register Output Should Not Drive Its Own Control Signal Directly or through Combinational Logic 5-18
Design Should Not Contain Delay Chains 5-18
Design Should Not Contain Ripple Clock Structures 5-18
Pulses Should Not Be Implemented Asynchronously 5-18
Multiple Pulses Should Not Be Generated in the Design 5-19
Design Should Not Contain SR Latches 5-19
Design Should Not Contain Latches 5-19
Combinational Logic Should Not Directly Drive Write Enable Signal of Asynchronous RAM 5-20
Design Should Not Contain Asynchronous Memory 5-20
Gated Clocks Should Be Implemented According to Altera Standard Scheme 5-21
Logic Cell Should Not Be Used to Generate Inverted Clock 5-21
Gated Clock Is Not Feeding At Least A Pre-Defined Number Of Clock Ports to Effectively Save Power: <n> 5-21
Clock Signal Source Should Drive Only Input Clock Ports 5-22
Clock Signal Should Be a Global Signal 5-22
Clock Signal Source Should Not Drive Registers that Are Triggered by Different Clock Edges 5-23
Combinational Logic Used as a Reset Signal Should Be Synchronized 5-23
External Reset Should Be Synchronized Using Two Cascaded Registers 5-23
External Reset Should Be Synchronized Correctly 5-24
Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized Correctly 5-25
Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized 5-25
Output Enable and Input of the Same Tri-State Nodes Should Not Be Driven by the Same Signal Source 5-26
Synchronous Port and Asynchronous Port of the Same Register Should Not Be Driven by the Same Signal Source 5-26
More Than One Asynchronous Signal Source of the Same Register Should Not Be Driven by the Same Source 5-26
Clock Port and Any Other Signal Port of the Same Register Should Not Be Driven by the Same Signal Source 5-26
Nodes with More Than Specified Number of Fan-outs: <n> 5-27
Top Nodes with Highest Fan-out: <n> 5-27
Data Bits Are Not Synchronized When Transferred between Asynchronous Clock Domains . 5-27
Multiple Data Bits Transferred Across Asynchronous Clock Domains Are Synchronized, But Not All Bits May Be Aligned in the Receiving Clock Domain 5-28
Data Bits Are Not Correctly Synchronized When Transferred Between Asynchronous Clock Domains 5-28
Only One VREF Pin Should Be Assigned to HardCopy Test Pin in an I/O Bank 5-28

A PLL Drives Multiple Clock Network Types 5-29
Data Bits Are Not Synchronized When Transferred to the State Machine of Asynchronous Clock Domains 5-29
No Reset Signal Defined to Initialize the State Machine 5-29
State Machine Should Not Contain Unreachable State 5-29
State Machine Should Not Contain a Deadlock State 5-30
State Machine Should Not Contain a Dead Transition 5-30
Enabling and Disabling Design Assistant Rules 5-30
Using the Assignment Editor 5-30
Using Verilog HDL 5-31
Using VHDL 5-31
Using TCL Commands 5-32
Viewing Design Assistant Results 5-32
Summary Report 5-33
Settings Report 5-33
Detailed Results Report 5-34
Messages Report 5-34
HardCopy Test Pins Report 5-34
Rule Suppression Assignments Report 5-34
Ignored Design Assistant Assignments Report 5-35
Custom Rules Report 5-35
Custom Rules 5-35
XML File Format for Custom Rules 5-35
Specifying the Path to the Custom Rules File 5-37
Custom Rules Coding Examples 5-37
Targeting Clock and Register-Control Architectural Features 5-41
Clock Network Resources 5-42
Reset Resources 5-42
Register Control Signals 5-43
Targeting Embedded RAM Architectural Features 5-43
Conclusion 5-44
Referenced Documents 5-44
Document Revision History 5-45

Chapter 6. Recommended HDL Coding Styles

Introduction 6-1
Quartus II Language Templates 6-1
Using Altera Megafunctions 6-2
Instantiating Altera Megafunctions in HDL Code 6-3
Instantiating Megafunctions Using the MegaWizard Plug-In Manager 6-4
Creating a Netlist File for Other Synthesis Tools 6-5
Instantiating Megafunctions Using the Port and Parameter Definition 6-5
Inferring Multiplier and DSP Functions from HDL Code 6-6
Multipliers—Inferring the LPM_MULT Megafunction from HDL Code 6-6
Multiply-Accumulators and Multiply-Adders—Inferring ALTMULT_ACCUM and ALTMULT_ADD Megafunctions from HDL Code 6-8
Inferring Memory Functions from HDL Code 6-12
RAM Functions—Inferring ALTSYNCRAM and ALTDPRAM Megafunctions from HDL Code . 6-13
Use Synchronous Memory Blocks 6-14
Avoid Unsupported Reset and Control Conditions 6-14
Check Read-During-Write Behavior 6-16
Single-Clock Synchronous RAM with Old Data Read-During-Write Behavior 6-17
Single-Clock Synchronous RAM with New Data Read-During-Write Behavior 6-19
Simple Dual-Port, Dual-Clock Synchronous RAM 6-21

True Dual-Port Synchronous RAM 6-23
Specifying Initial Memory Contents at Power-Up 6-27
ROM Functions—Inferring ALTSYNCRAM and LPM_ROM Megafunctions from HDL Code . . 6-29
Shift Registers—Inferring the ALTSHIFT_TAPS Megafunction from HDL Code 6-32
Simple Shift Register 6-33
Shift Register with Evenly Spaced Taps 6-34
Coding Guidelines for Registers and Latches 6-36
Register Power-Up Values in Altera Devices 6-36
Secondary Register Control Signals Such as Clear and Clock Enable 6-38
Latches 6-42
Unintentional Latch Generation 6-42
Inferring Latches Correctly 6-43
General Coding Guidelines 6-46
Tri-State Signals 6-46
Clock Multiplexing 6-47
Adder Trees 6-51
Architectures with 4-Input LUTs in Logic Elements 6-51
Architectures with 6-Input LUTs in Adaptive Logic Modules 6-52
State Machines 6-53
Verilog HDL State Machines 6-54
VHDL State Machines 6-58
Multiplexers 6-60
Quartus II Software Option for Multiplexer Restructuring 6-61
Multiplexer Types 6-61
Default or Others Case Assignment 6-63
Implicit Defaults 6-63
Degenerate Multiplexers 6-65
Buses of Multiplexers 6-67
Cyclic Redundancy Check Functions 6-68
If Performance is Important, Optimize for Speed 6-68
Use Separate CRC Blocks Instead of Cascaded Stages 6-68
Use Separate CRC Blocks Instead of Allowing Blocks to Merge 6-68
Take Advantage of Latency if Available 6-69
Save Power by Disabling CRC Blocks When Not in Use 6-69
Use the Device Synchronous Load (sload) Signal to Initialize 6-69
Comparators 6-69
Counters 6-71
Designing with Low-Level Primitives 6-71
Conclusion 6-72
Referenced Documents 6-72
Document Revision History 6-73

Chapter 7. Managing Metastability with the Quartus II Software

Introduction 7-1
Metastability Analysis in the Quartus II Software 7-2
What is a Synchronization Register Chain? 7-3
Identifying Synchronizers for Metastability Analysis 7-4
Using the Global Synchronizer Identification Setting 7-4
Refining Synchronizer Identification Using the Instance-Specific Assignment 7-5
How Timing Constraints Affect Synchronizer Chain Identification and Metastability Analysis . . 7-6
Metastability and MTBF Reporting 7-6
Metastability Report 7-7
MTBF Summary Report 7-7
Synchronizer Summary 7-8

Synchronizer Chain Statistics Report in the TimeQuest Timing Analyzer 7-8
Synchronizer Data Toggle Rate in MTBF Calculation 7-9
MTBF Optimization 7-9
Synchronization Register Chain Length 7-10
Reducing Metastability Effects 7-11
Apply Complete System-Centric Timing Constraints for the TimeQuest Timing Analyzer 7-11
Force the Identification of Synchronization Registers 7-11
Set the Synchronizer Data Toggle Rate 7-12
Optimize Metastability During Fitting 7-12
Increase the Length of Synchronizers to Protect and Optimize 7-12
Set Fitter Effort to Standard Fit instead of Auto Fit 7-12
If Possible, Increase the Number of Stages Used in Synchronizers 7-12
If Possible, Select a Faster Speed Grade Device 7-13
Scripting Support 7-13
Identifying Synchronizers for Metastability Analysis 7-13
Synchronizer Data Toggle Rate in MTBF Calculation 7-13
report_metastability TimeQuest and Tcl Command 7-14
MTBF Optimization 7-14
Synchronization Register Chain Length 7-14
Conclusion 7-14
Referenced Documents 7-15
Document Revision History 7-15

Chapter 8. Best Practices for Incremental Compilation Partitions and Floorplan Assignments
Introduction 8-1
Overview: Incremental Compilation 8-1
Choosing the Netlist Type and Fitter Preservation Level 8-2
Top-Down versus Bottom-Up Compilation Flows 8-3
Generating Bottom-Up Design Partition Scripts for Project Management 8-4
Why Plan Partitions and Floorplan Assignments for Incremental Compilation? 8-4
Partition Boundaries and Optimization 8-5
Creating Design Partitions: General Partitioning Guidelines 8-6
Plan Design Hierarchy and Source Design Files 8-6
Using Partitions with Third-Party Synthesis Tools 8-7
Partition Design by Functionality and Block Size 8-8
Partition Design by Clock Domain and Timing Criticality 8-8
Consider What Is Changing 8-8
Creating Design Partitions: Design Guidelines 8-9
Register Partition Inputs and Outputs 8-9
Minimize Cross-Partition-Boundary I/O 8-10
Avoid the Need for Logic Optimization Across Partitions 8-11
Keep Logic in the Same Partition for Optimization and Merging 8-12
Keep Constants in the Same Partition as Logic 8-13
Avoid Unconnected Partition I/O 8-14
Avoid Signals That Drive Multiple Partition I/O or Connect I/O Together 8-14
Invert Clocks in Destination Partitions 8-16
Connect I/O Directly to I/O Register for Packing Across Partition Boundaries 8-16
Do Not Use Internal Tri-States 8-20
Include All Tri-State and Enable Logic in the Same Partition 8-20
Include Bidirectional I/O Registers in the Same Partition 8-21
Summary of Guidelines Related to Logic Optimization Across Partitions 8-22
Creating Design Partitions: Consider Additional Design Suggestions 8-23
Balance Logic Resources 8-23
Balance Global Routing Signals and Clock Networks if Required 8-24

Assign Virtual Pins in Bottom-Up Flows 8-25
Perform Timing Budgeting if Required 8-26
Consider a Cascaded Reset Structure 8-26
Drive Clocks Directly in Bottom-Up Flows 8-28
Recreate PLLs for Lower-Level Partitions if Required in Bottom-Up Flows 8-28
Checking Partition Quality 8-29
Incremental Compilation Advisor 8-29
Design Partition Planner 8-29
Floorplan Partition Coloring 8-31
Viewing Design Partition Planner and Floorplan Side-by-Side 8-32
Partition Statistics Report 8-33
Report Partition Timing in the TimeQuest Timing Analyzer 8-34
Ensure Partition Assignments Do Not Impact the Quality of Results 8-34
Introduction to Design Floorplans 8-35
The Difference between Logical Partitions and Physical Regions 8-35
Why Create a Floorplan? 8-36
Why Create a Floorplan in Bottom-Up Flows? 8-36
Why Create a Floorplan in Top-Down Flows? 8-36
When to Create a Floorplan 8-37
Early Floorplan 8-38
Late Floorplan 8-38
Creating a Design Floorplan: Placement Guidelines 8-38
Assigning Partitions to LogicLock Regions 8-39
How to Size and Place Regions 8-39
Modifying Region Size and Origin 8-40
I/O Connections 8-41
LogicLock Resource Exclusions 8-41
Creating Non-Rectangular Regions 8-42
Checking Floorplan Quality 8-43
Incremental Compilation Advisor 8-43
LogicLock Region Resource Estimates 8-43
LogicLock Region Properties Statistics Report 8-43
Critical Path Settings for Chip Planner 8-44
Locate the Quartus II TimeQuest Timing Analyzer Path in the Chip Planner 8-44
Inter-Region Connection Bundles 8-44
Routing Utilization 8-44
Ensure Floorplan Assignments Do Not Impact Quality of Results 8-44
Recommended Design Flows and Application Examples 8-45
Create a Floorplan for the Entire Design in a Top-Down Flow 8-45
Create a Floorplan as the Project Lead in a Bottom-Up Flow 8-46
Create a Floorplan Assignment for One Design Block with Difficult Timing 8-47
Potential Issues with Creating Partitions and Floorplan Assignments 8-47
Logic and Resource Utilization Effects 8-48
Routing Utilization Effects 8-48
Conclusion 8-48
Referenced Documents 8-49
Revision History 8-49

Section III. Synthesis

Chapter 9. Quartus II Integrated Synthesis

Introduction 9-1
Design Flow 9-2

Language Support . . . 9-4
Verilog HDL Support . . . 9-4
Verilog-2001 Support . . . 9-5
SystemVerilog Support . . . 9-5
Initial Constructs and Memory System Tasks . . . 9-7
Verilog HDL Macros . . . 9-7
VHDL Support . . . 9-8
VHDL Standard Libraries and Packages . . . 9-9
VHDL wait Constructs . . . 9-10
AHDL Support . . . 9-11
Schematic Design Entry Support . . . 9-11
State Machine Editor . . . 9-11
Design Libraries . . . 9-12
Specifying a Destination Library Name in the Settings Dialog Box . . . 9-13
Specifying a Destination Library Name in the Quartus II Settings File or Using Tcl . . . 9-13
Specifying a Destination Library Name in a VHDL File . . . 9-13
Mapping a VHDL Instance to an Entity in a Specific Library . . . 9-14
Using Parameters/Generics . . . 9-16
Setting Default Parameter Values and BDF Instance Parameter Values . . . 9-16
Passing Parameters Between Two Design Languages . . . 9-18
Incremental Compilation . . . 9-20
Partitions for Preserving Hierarchical Boundaries . . . 9-20
Parallel Synthesis . . . 9-21
Quartus II eXported Partition (.qxp) File as Source . . . 9-21
Quartus II Synthesis Options . . . 9-22
Setting Synthesis Options . . . 9-24
Analysis & Synthesis Settings Page of the Settings Dialog Box . . . 9-24
Quartus II Logic Options . . . 9-25
Synthesis Attributes . . . 9-25
Synthesis Directives . . . 9-27
Optimization Technique . . . 9-28
Speed Optimization Technique for Clock Domains . . . 9-28
Auto Gated Clock Conversion . . . 9-29
Timing-Driven Synthesis . . . 9-30
SDC Constraint Protection . . . 9-31
PowerPlay Power Optimization . . . 9-31
Limiting DSP Block Usage in Partitions . . . 9-32
Restructure Multiplexers . . . 9-34
Synthesis Effort . . . 9-35
State Machine Processing . . . 9-36
Manually Specifying State Assignments Using the syn_encoding Attribute . . . 9-37
Manually Specifying Enumerated Types Using the enum_encoding Attribute . . . 9-39
Safe State Machines . . . 9-41
Power-Up Level . . . 9-42
Inferred Power-Up Levels . . . 9-43
Power-Up Don't Care . . . 9-43
Remove Duplicate Registers . . . 9-44
Remove Redundant Logic Cells . . . 9-44
Preserve Registers . . . 9-44
Disable Register Merging/Don't Merge Register . . . 9-45
Noprune Synthesis Attribute/Preserve Fan-out Free Register Node . . . 9-46
Keep Combinational Node/Implement as Output of Logic Cell . . . 9-46
Don't Retime, Disabling Synthesis Netlist Optimizations . . . 9-47
Don't Replicate, Disabling Synthesis Netlist Optimizations . . . 9-48

Maximum Fan-Out 9-49
Controlling Clock Enable Signals with Auto Clock Enable Replacement and direct_enable 9-50
Megafunction Inference Control 9-51
Multiply-Accumulators and Multiply-Adders 9-51
Shift Registers 9-51
RAM and ROM 9-52
RAM to Logic Cell Conversion 9-53
RAM Style and ROM Style—for Inferred Memory 9-53
Turning Off Add Pass-Through Logic to Inferred RAMs/ no_rw_check Attribute Setting 9-54
RAM Initialization File—for Inferred Memory 9-57
Multiplier Style—for Inferred Multipliers 9-58
Full Case 9-60
Parallel Case 9-61
Translate Off and On / Synthesis Off and On 9-62
Ignore translate_off and synthesis_off Directives 9-63
Read Comments as HDL 9-63
Use I/O Flipflops 9-64
Specifying Pin Locations with chip_pin 9-65
Using altera_attribute to Set Quartus II Logic Options 9-66
Verilog HDL 9-67
VHDL 9-67
Analyzing Synthesis Results 9-69
Analysis and Synthesis Section of the Compilation Report 9-69
Project Navigator 9-69
Analyzing and Controlling Synthesis Messages 9-70
Quartus II Messages 9-70
VHDL and Verilog HDL Messages 9-71
Setting the HDL Message Level 9-72
Enabling or Disabling Specific HDL Messages by Module/Entity 9-73
Node-Naming Conventions in Quartus II Integrated Synthesis 9-74
Hierarchical Node-Naming Conventions 9-74
Node-Naming Conventions for Registers (DFF or D Flipflop Atoms) 9-74
Register Changes During Synthesis 9-76
Synthesis and Fitting Optimizations 9-76
State Machines 9-77
Inferred Adder-Subtractors, Shift Registers, Memory, and DSP Functions 9-77
Packed Input and Output Registers of RAM and DSP Blocks 9-77
Preserving Register Names 9-78
Node-Naming Conventions for Combinational Logic Cells 9-78
Preserving Combinational Logic Names 9-79
Scripting Support 9-80
Adding an HDL File to a Project and Setting the HDL Version 9-80
Quartus II Synthesis Options 9-81
Assigning a Pin 9-83
Creating Design Partitions for Incremental Compilation 9-83
Conclusion 9-84
Referenced Documents 9-84
Document Revision History 9-85

Chapter 10. Synopsys Synplify Support

Introduction 10-1
Altera Device Family Support 10-1
Design Flow 10-2
Output Netlist File Name and Result Format 10-5

Synplify Optimization Strategies 10-6
Using Synplify Premier to Optimize Your Design 10-6
Implementations in Synplify Pro or Premier 10-7
Timing-Driven Synthesis Settings 10-7
Clock Frequencies 10-7
Multiple Clock Domains 10-8
Input and Output Delays 10-8
Multicycle Paths 10-8
False Paths 10-8
FSM Compiler 10-9
FSM Explorer in Synplify Pro and Premier 10-10
Optimization Attributes and Options 10-10
Retiming in Synplify Pro and Premier 10-10
Maximum Fan-Out 10-10
Preserving Nets 10-10
Register Packing 10-11
Resource Sharing 10-11
Preserving Hierarchy 10-11
Register Input and Output Delays 10-11
syn_direct_enable 10-12
Standard I/O Pad 10-12
Altera-Specific Attributes 10-12
altera_chip_pin_lc 10-13
altera_implement_in_esb or altera_implement_in_eab 10-13
altera_io_powerup 10-13
altera_io_opendrain 10-14
Exporting Designs to the Quartus II Software Using NativeLink Integration 10-14
Running the Quartus II Software from within the Synplify Software 10-15
Using the Quartus II Software to Run the Synplify Software 10-15
Running the Quartus II Software Manually Using the Synplify-Generated Tcl Script 10-16
Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File 10-16
Individual Clocks and Frequencies 10-17
Input and Output Delay 10-17
Multicycle Path 10-17
False Path 10-17
Passing Constraints to the Quartus II Software using Tcl Commands 10-18
Global Signals 10-18
Default or Global Clock Frequency 10-18
Individual Clocks and Frequencies 10-18
Virtual Clocks 10-19
Route Delay Option 10-19
Multiple Clocks in Different Clock Groups 10-19
Multiple Clocks with Different Frequencies in the Same Clock Group 10-20
Inter-Clock Relationships—Delays and False Paths between Clocks 10-21
False Paths 10-21
Multicycle Paths 10-22
Maximum Path Delays 10-23
Guidelines for Altera Megafunctions and Architecture-Specific Features 10-25
Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager 10-26
Using MegaWizard Plug-In Manager-Generated Verilog HDL Files for Megafunction Instantiation 10-27
Using MegaWizard Plug-In Manager-Generated VHDL Files for Megafunction Instantiation 10-27
Changing Synplify’s Default Behavior for Instantiated Altera Megafunctions 10-27

Instantiating Intellectual Property Using the MegaWizard Plug-In Manager and IP Toolbench 10-28
Using Generated Verilog HDL Files for Black Box IP Function Instantiation 10-28
Using Generated VHDL Files for Black Box IP Function Instantiation 10-29
Other Synplify Software Attributes for Creating Black Boxes 10-29
Including Files for Quartus II Placement and Routing Only 10-30
Inferring Altera Megafunctions from HDL Code 10-31
Inferring Multipliers 10-31
Inferring RAM 10-33
RAM Initialization 10-35
Inferring ROM 10-36
Inferring Shift Registers 10-36
Incremental Compilation and Block-Based Design 10-37
Creating a Design with Separate Netlist Files for Incremental Compilation 10-38
Using MultiPoint Synthesis with Incremental Compilation 10-39
Set Compile Points and Create Constraint Files 10-39
Additional Considerations for Compile Points 10-41
Creating a Quartus II Project for Compile Points and Multiple .vqm Files 10-41
Creating Multiple .vqm Files for Incremental Compilation Using Separate Synplify Projects . . 10-43
Manually Creating Multiple .vqm Files Using Black Boxes 10-43
Creating a Quartus II Project for Multiple .vqm Files 10-47
Performing Incremental Compilation in the Quartus II Software 10-48
Conclusion 10-49
Referenced Documents 10-49
Document Revision History 10-50

Chapter 11. Mentor Graphics Precision Synthesis Support

Introduction 11-1
Device Family Support 11-2
Design Flow 11-2
Creating and Compiling a Project in the Precision Synthesis Software 11-6
Creating a Project 11-6
Compiling the Design 11-6
Mapping the Precision Synthesis Design 11-6
Setting Timing Constraints 11-7
Setting Mapping Constraints 11-8
Assigning Pin Numbers and I/O Settings 11-8
Assigning I/O Registers 11-9
Disabling I/O Pad Insertion 11-10
Preventing the Precision Synthesis Software from Adding I/O Pads 11-10
Preventing the Precision Synthesis Software from Adding an I/O Pad on an Individual Pin 11-10
Controlling Fan-Out on Data Nets 11-10
Synthesizing the Design and Evaluating the Results 11-11
Obtaining Accurate Logic Utilization and Timing Analysis Reports 11-11
Exporting Designs to the Quartus II Software Using NativeLink Integration 11-11
Running the Quartus II Software from within the Precision Synthesis Software 11-12
Running the Quartus II Software Manually Using the Precision Synthesis-Generated Tcl Script 11-13
Using Quartus II Software to Launch the Precision Synthesis Software 11-14
Passing Constraints to the Quartus II Software 11-14
create_clock 11-14
set_input_delay 11-15
set_output_delay 11-16
set_max_delay 11-16

set_min_delay 11-17
set_false_path 11-17
set_multicycle_path 11-18
Guidelines for Altera Megafunctions and Architecture-Specific Features 11-19
Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager 11-19
Using MegaWizard Plug-In Manager-Generated Verilog HDL Files for Megafunction Instantiation 11-20
Using MegaWizard Plug-In Manager-Generated VHDL Files for Megafunction Instantiation 11-20
Instantiating Intellectual Property Using the MegaWizard Plug-In Manager and IP Toolbench 11-20
Using Generated Verilog HDL Files for Black Box IP Function Instantiation 11-21
Using Generated VHDL Files for Black Box IP Function Instantiation 11-22
Inferring Altera Megafunctions from HDL Code 11-22
Multipliers 11-23
Using the GUI 11-23
Using Attributes 11-23
Multiplier-Accumulators and Multiplier-Adders 11-25
Controlling DSP Block Inference 11-25
RAM and ROM 11-27
Incremental Compilation and Block-Based Design 11-28
Creating a Design with Precision RTL Plus Incremental Synthesis 11-28
Creating Partitions with the incr_partition Attribute 11-29
Creating Multiple EDIF Netlist Files Using Separate Precision Projects or Implementations 11-30
Creating Black Boxes in Verilog HDL 11-32
Creating Black Boxes in VHDL 11-33
Creating Quartus II Projects for Multiple EDIF Files 11-34
Creating a Single Quartus II Project for a Top-Down Incremental Compilation Flow 11-35
Creating Multiple Quartus II Projects for a Bottom-Up Flow 11-36
Hierarchy and Design Considerations 11-36
Conclusion 11-37
Referenced Documents 11-37
Document Revision History 11-38

Chapter 12. Mentor Graphics LeonardoSpectrum Support

Introduction 12-1
Design Flow 12-1
Optimization Strategies 12-3
Timing-Driven Synthesis 12-3
Global Power Tab 12-3
Clock Power Tab 12-4
Input and Output Power Tabs 12-4
Other Constraints 12-4
Encoding Style 12-4
Resource Sharing 12-5
Mapping I/O Registers 12-5
Timing Analysis with the Leonardo-Spectrum Software 12-5
Exporting Designs Using NativeLink Integration 12-6
Generating Netlist Files 12-6
Including Design Files for Black Boxed Modules 12-6
Passing Constraints with Scripts 12-7
Integration with the Quartus II Software 12-7
Guidelines for Altera Megafunctions and LPM Functions 12-7
Instantiating Altera Megafunctions 12-8

Inferring Altera Memory Elements . . . 12-8
Inferring Multipliers and DSP Functions . . . 12-9
Simple Multipliers . . . 12-9
Multiplier Accumulators . . . 12-9
Multiplier Adders . . . 12-10
Controlling DSP Block Inference . . . 12-10
Global Attribute . . . 12-11
Module Level Attributes . . . 12-11
Signal Level Attributes . . . 12-12
Guidelines for Using DSP Blocks . . . 12-14
Block-Based Design with the Quartus II Software . . . 12-15
Hierarchy and Design Considerations . . . 12-15
Creating a Design with Multiple .edif Files . . . 12-16
Generating Multiple .edif Files Using the LogicLock Option . . . 12-16
Creating a Quartus II Project for Multiple .edif Files Including LogicLock Regions . . . 12-18
Generating Multiple .edif Files Using Black Boxes . . . 12-19
Black Box Methodology in Verilog HDL . . . 12-20
Black Boxing in VHDL . . . 12-21
Creating a Quartus II Project for Multiple .edif Files . . . 12-23
Incremental Synthesis Flow . . . 12-24
Modifications Required for the LogicLock_Incremental.tcl Script File . . . 12-24
Running the Tcl Script File in LeonardoSpectrum . . . 12-25
Conclusion . . . 12-26
Referenced Documents . . . 12-26
Document Revision History . . . 12-26

Chapter 13. Analyzing Designs with Quartus II Netlist Viewers

Introduction . . . 13-1
When to Use Viewers: Analyzing Design Problems . . . 13-1
Quartus II Design Flow with Netlist Viewers . . . 13-3
RTL Viewer Overview . . . 13-4
State Machine Viewer Overview . . . 13-5
Technology Map Viewer Overview . . . 13-5
Introduction to the User Interface . . . 13-6
Schematic View . . . 13-7
Schematic Symbols . . . 13-7
Selecting an Item in the Schematic View . . . 13-13
Moving and Panning in the Schematic View . . . 13-14
Hierarchy List . . . 13-14
Selecting an Item in the Hierarchy List . . . 13-15
Enable or Disable the Auto Hierarchy List . . . 13-15
State Machine Viewer . . . 13-16
State Diagram View . . . 13-16
State Transition Table . . . 13-17
State Encoding Table . . . 13-17
Selecting an Item in the State Machine Viewer . . . 13-17
Switching Between State Machines . . . 13-18
Navigating the Schematic View . . . 13-18
Traversing and Viewing the Design Hierarchy . . . 13-18
Flattening the Design Hierarchy . . . 13-18
Viewing the Contents of a Design Hierarchy within the Current Schematic . . . 13-19
Viewing Contents of Atom Primitives . . . 13-19
Viewing the Properties of Instances and Primitives . . . 13-20
Viewing LUT Representations in the Technology Map Viewer . . . 13-21

Grouping Combinational Logic into Logic Clouds 13-22
Logic Clouds in the RTL Viewer 13-23
Logic Clouds in the Technology Map Viewer 13-23
Manually Group and Ungroup Logic Clouds 13-24
Changing the Constant Signal Value Formatting 13-24
Zooming and Magnification 13-24
Schematic Debugging and Tracing Using the Bird's Eye View 13-26
Full Screen View 13-26
Partitioning the Schematic into Pages 13-26
Moving between Schematic Pages 13-27
Moving Back and Forward Through Schematic Pages 13-27
Following Nets Across Schematic Pages 13-27
Go to Net Driver 13-28
Customizing the Schematic Display in the RTL Viewer 13-29
Filtering in the Schematic View 13-29
Filter Sources Command 13-30
Filter Destinations Command 13-30
Filter Sources and Destinations Command 13-31
Filter Between Selected Nodes Command 13-31
Filter Selected Nodes and Nets Command 13-31
Filter Bus Index Command 13-32
Filter Command Processing 13-32
Filtering Across Hierarchies 13-33
Expanding a Filtered Netlist 13-34
Reducing a Filtered Netlist 13-35
Probing to Source Design File and Other Quartus II Windows 13-35
Moving Selected Nodes to Other Quartus II Windows 13-36
Probing to the Viewers from Other Quartus II Windows 13-37
Viewing a Timing Path 13-38
Other Features in the Schematic Viewer 13-39
Tooltips 13-39
Radial Menu 13-41
Enabling and Disabling the Radial Menu 13-42
Customizing the Shortcut Commands 13-43
Changing the Time Interval 13-43
Rollover 13-44
Displaying Net Names in the Schematic 13-44
Displaying Node Names in the Schematic 13-44
Find Command 13-44
Exporting and Copying a Schematic Image 13-45
Printing 13-46
Debugging HDL Code with the State Machine Viewer 13-46
Simulation of State Machine Gives Unexpected Results 13-47
Conclusion 13-50
Document Revision History 13-50

Additional Information

About this Handbook Info-1
How to Contact Altera Info-1
Third-Party Software Product Information Info-1
Typographic Conventions Info-2

Chapter Revision Dates

The chapters in this book, *Quartus II Handbook Version 9.0 Volume 1: Design and Synthesis*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1 Design Planning with the Quartus II Software
Revised: *March 2009*
Part Number: *QII51016-9.0.0*

Chapter 2 Quartus II Incremental Compilation for Hierarchical and Team-Based Design
Revised: *March 2009*
Part Number: *QII51015-9.0.0*

Chapter 3 Quartus II Design Flow for MAX+PLUS II Users
Revised: *March 2009*
Part Number: *QII51002-9.0.0*

Chapter 4 Quartus II Support for HardCopy Series Devices
Revised: *March 2009*
Part Number: *QII51004-9.0.0*

Chapter 5 Design Recommendations for Altera Devices and the Quartus II Design Assistant
Revised: *March 2009*
Part Number: *QII51006-9.0.0*

Chapter 6 Recommended HDL Coding Styles
Revised: *March 2009*
Part Number: *QII51007-9.0.0*

Chapter 7 Managing Metastability with the Quartus II Software
Revised: *March 2009*
Part Number: *QII51018-9.0.0*

Chapter 8 Best Practices for Incremental Compilation Partitions and Floorplan Assignments
Revised: *March 2009*
Part Number: *QII51017-9.0.0*

Chapter 9 Quartus II Integrated Synthesis
Revised: *March 2009*
Part Number: *QII51008-9.0.0*

Chapter 10 Synopsys Synplify Support
Revised: *March 2009*
Part Number: *QII51009-9.0.0*

Chapter 11 Mentor Graphics Precision Synthesis Support
Revised: *March 2009*
Part Number: *QII51011-9.0.0*

Chapter 12 Mentor Graphics LeonardoSpectrum Support
Revised: *March 2009*
Part Number: *QII51010-9.0.0*

Chapter 13 Analyzing Designs with Quartus II Netlist Viewers
Revised: *March 2009*
Part Number: *QII51013-9.0.0*

Section I. Design Flows

The Altera® Quartus® II design software provides a complete design environment that easily adapts to your specific design requirements. This handbook is arranged in chapters, sections, and volumes that correspond to the major stages in the overall design flow. For a general introduction to features and the standard design flow in the software, refer to the *Introduction to the Quartus II Software* manual.

This section starts the *Quartus II Handbook* with an introduction to design planning and a collection of various specialized design flows in the Quartus II software.

This section includes the following chapters:

- Chapter 1, Design Planning with the Quartus II Software

  This chapter discusses important FPGA design planning issues such as device selection, early power estimation, I/O pin planning, and design planning. It provides recommendations and describes various tools available for Altera FPGAs to help you improve design productivity.

  Use this chapter when starting your design for an overview of various planning considerations.

- Chapter 2, Quartus II Incremental Compilation for Hierarchical and Team-Based Design

  This chapter documents Altera's incremental design and compilation flow, which allows you to preserve the results and performance for unchanged logic in your design as you make changes elsewhere, reduces design iteration time by up to 70% so you achieve timing closure efficiently, and facilitates modular hierarchical and team-based design flows using top-down or bottom-up methodologies

  Use this chapter to learn about using the incremental compilation flow, and read about recommended incremental design flows using Quartus II features.

- Chapter 3, Quartus II Design Flow for MAX+PLUS II Users

  There are many features in the Quartus II software to help users of the legacy MAX+PLUS® II software easily transition to the Quartus II software design environment. This chapter describes how to convert MAX+PLUS II designs to Quartus II projects, and highlights the similarities and differences between the MAX+PLUS II and Quartus II design flows.

  Use this chapter if you are using the legacy MAX+PLUS II software.

- Chapter 4, Quartus II Support for HardCopy Series Devices

  Using the Quartus II software, you can leverage an FPGA device as a prototype and seamlessly migrate your design to a HardCopy ASIC to reduce cost for volume production. This chapter describes the Quartus II support for HardCopy design flows.

  Use this chapter if you want to migrate your design to a HardCopy ASIC.

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

1. Design Planning with the Quartus II Software

QII51016-9.0.0

Introduction

This chapter discusses important FPGA design planning issues, provides recommendations, and describes various tools available for Altera® FPGAs to help you improve design productivity.

The inherent flexibility of advanced FPGAs means that the pin layout, power consumption, area utilization, and timing performance for each design block are all dependent on the final design implementation. Due to the significant increase in FPGA device densities over the last few years, designs are increasingly complex and may involve multiple designers. The system architect must resolve these design issues when integrating design blocks, often leading to problems that affect the overall time to market and thereby increase cost. Many potential problems can be solved earlier in the design cycle by performing good design planning.

This chapter contains the following sections:

- "Creating Design Specifications" on page 1–2
- "Device Selection" on page 1–2
- "Planning for Device Programming/Configuration" on page 1–4
- "Early Power Estimation" on page 1–5
- "Early Pin Planning and I/O Analysis" on page 1–6
- "Selecting Third-Party EDA Tool Flows" on page 1–8
- "Planning for On-Chip Debugging Options" on page 1–9
- "Design Practices and HDL Coding Styles" on page 1–11
- "Planning for Hierarchical and Team-Based Design" on page 1–13
- "Fast Synthesis and Early Timing Estimation" on page 1–17

Before reading the design planning guidelines discussed in this chapter, consider your design priorities: What are the important factors for your design? More device features, density, or performance can increase system cost. Signal integrity and board issues may impact I/O pin locations. Power, timing performance, and area utilization affect each other, and compilation time is affected by optimizations for these factors.

The Quartus® II software optimizes designs for the best average results, but you can change settings to focus on one aspect of the design results and trade off other aspects. Certain tools or debugging options can lead to restrictions in your design flow. If you know what is important in a particular design, this knowledge will help you choose the tools, features, and methodologies that you should use with the design. This chapter cannot cover every possible consideration for planning a complex FPGA design, but once you understand your design priorities, you can use the design planning issues described here as a guide to help ensure a productive and successful FPGA design flow.

This chapter provides an introduction to various design and planning features in the Quartus II software. For a general overview of the Quartus II design flow and features, refer to the *Introduction to the Quartus II Software* manual. For more details about specific Quartus II features and methodologies, this chapter provides references to other appropriate chapters in the *Quartus II Handbook*. After you have selected a device family, you can refer to the Design Guidelines section of the device's Literature page on Altera's website to check if additional guidelines are available for your device family.

Creating Design Specifications

Before you create your logic design or complete your system design, you should have detailed design specifications. The specifications define what the system should do, specify the I/O interfaces for the FPGA, and include a block diagram of basic design functions. Taking the time to create these specifications will help improve design efficiency.

Creating a test plan at this phase can also help you design for testability and manufacturability. For example, do you want to perform any built-in self-test functions to drive interfaces? If so, you could use a UART interface with a Nios® II processor inside the FPGA device. You might require the ability to validate all the design interfaces. Refer to "Planning for On-Chip Debugging Options" on page 1–9 for guidelines related to analyzing and debugging the device after it is in the system.

It is also useful to consider a common design directory structure at this point, if your design includes multiple designers. This will ease the design integration stages. "Planning for Hierarchical and Team-Based Design" on page 1–13 provides more suggestions for team-based designs.

Device Selection

The first stage in design planning is choosing the best device for your application. The device selection affects the rest of your design cycle, including board specification and layout. Most of this planning is performed outside of the Quartus II software, but this section provides a few suggestions to aid in the planning process.

It is important to choose the device family that best suits your design requirements. Different families offer different trade-offs, including cost, performance, logic and memory density, I/O density, power utilization, and packaging. You should also consider feature requirements, such as I/O standards support, high-speed transceivers, global/regional clock networks, and the number of phase-locked loops (PLLs) available in the device. You can review important features of each device family in the Selector Guides available on the Altera website. Each device family also has a device handbook or set of data sheets that documents the device features in detail.

Determining the required device density can be a challenging part of the design planning process. Devices with more logic resources and higher I/O counts can implement larger and potentially more complex designs, but may have a higher cost. Select a device that meets your design requirements with some safety margin, in case you want to add more logic later in the design cycle or reserve logic and memory for on-chip debugging (refer to "Planning for On-Chip Debugging Options" on page 1–9). Consider requirements for specific types of dedicated logic blocks, such as memory blocks of different sizes, or digital signal processing (DSP) blocks to implement certain arithmetic functions.

If you have prior designs that target Altera devices, you can use their resource utilization as an estimate for your new design. You can compile existing designs in the Quartus II software with the device selection set to **Auto** to review the resource utilization and find out which device density fits the design.

Coding style, device architecture, and the optimization options used in the Quartus II software can significantly affect a design's resource utilization.

For resource utilization estimates for certain configurations of Altera's intellectual property (IP) designs, refer to the User Guides for Altera megafunctions and IP MegaCore® functions on the IP Megafunctions page on the Altera website (www.altera.com/literature/lit-ip.jsp). You can use these numbers to help estimate the resource utilization of your design.

Device Migration Planning

Determine whether you want the option of migrating your design to another device density to allow flexibility when the design nears completion, or whether you want to migrate to a HardCopy® ASIC when the design reaches volume production. In some cases, designers may target a smaller (and less expensive) device and then move to a larger device if necessary to fit their design. Other designers may prototype their design in a larger device to reduce optimization time and achieve timing closure more quickly, and then migrate to a final smaller device after prototyping. Similarly, many designers compile and optimize their design for an FPGA device before moving to a HardCopy ASIC when the design is complete and ready for higher-volume production. If you would like this flexibility, you should specify these migration options in the Quartus II software at the beginning of your design cycle. Specify the target migration devices in the **Migration compatibility** or **Companion device** sections of the **Device** page in the **Settings** dialog box.

Selecting a migration device has an impact on pin placement because some pins may serve different functions in different device densities or package sizes. When making pin assignments in the Quartus II software, the Pin Migration View in the Pin Planner highlights pins that change function between your migration devices. (Refer to "Early Pin Planning and I/O Analysis" on page 1–6 for more details.) Selecting a companion device may force you to restrict logic utilization to ensure that your design is compatible with a selected HardCopy device. Adding migration or companion devices later in the design cycle is possible, but requires extra effort to check pin assignments, and may require design changes to fit into the new target device. It is much easier to consider these issues early in the design cycle than at the end, when the design is near completion and ready for migration.

In addition, if you are planning to use a HardCopy ASIC, review HardCopy guidelines early in the design cycle for any Quartus II settings that should be used or other restrictions you should consider. It is especially important to use complete timing constraints if you want to migrate to a HardCopy device because of the rigorous verification requirements for ASICs.

For more information about timing requirements and analysis for HardCopy designs, refer to the *HardCopy Series Handbook*.

Planning for Device Programming/Configuration

Another important part of the device planning is determining how you want to program or configure the device in your system. Choosing your programming or configuration method early allows system and board designers to determine what companion devices, if any, are required for your system. Your board layout also depends on the type of programming or configuration method you plan to use for programmable devices. Many programming options use a JTAG interface to connect to the devices, so you may require a JTAG chain be set up on the board.

The device family handbooks describe the configuration options available for a given device family. For more details about configuration options, refer to the *Configuration Handbook*. For information about programming CPLD devices, refer to your device data sheet or handbook. Programming and configuration of Altera devices includes the following options:

- Using enhanced configuration devices—These devices combine industry-standard flash memory with a feature-rich configuration controller, including device features such as concurrent and dynamic configuration, data compression, clock division, and an external flash memory interface. You can also implement remote and local system updates with enhanced configuration devices.
- Using Flash memory devices with a memory controller, such as an Altera MAX® device—The flash memory controller can interface with a PC or microprocessor to receive configuration data via a parallel port.
- Using the Quartus II Serial Flash Loader (SFL)—This scheme allows you to configure the FPGA and program serial configuration devices using the same JTAG interface.
- Using the Quartus II Parallel Flash Loader (PFL)—This solution quickly retrieves data from a JTAG interface and generates data formatted for the receiving target flash device, significantly reducing the flash device programming time. If your system already contains a common flash interface (CFI) flash memory, you can utilize it for the FPGA configuration storage as well, because the PFL feature supports many common industry-standard flash devices. If you choose this method, check the list of supported flash devices early in your system design cycle and plan accordingly. Refer to *AN 386: Using the MAX II Parallel Flash Loader with the Quartus II Software* for the list of supported Flash devices.

Early Power Estimation

You can use the Quartus II power estimation and analysis tools to provide information to PCB board and system designers. You can perform early power estimation before you have created any source code, or when you have a preliminary version of the design, and then perform the most accurate analysis when the design is complete.

Device power consumption must be accurately estimated to develop an appropriate power budget and to design the power supplies, voltage regulators, heat sink, and cooling system. Power estimation and analysis has two significant planning requirements:

- Thermal planning—You must ensure that the cooling solution is sufficient to dissipate the heat generated by the device. In particular, the computed junction temperature must fall within normal device specifications.
- Power supply planning—Power supplies must provide adequate current to support device operation.

Power consumption in FPGA devices is dependent on the design, providing a challenge during early board specification and layout. The Altera PowerPlay Early Power Estimator spreadsheet allows you to estimate power utilization before the design is complete, by processing information about the device resources that will be used in the design, as well as the operating frequency, toggle rates, and environmental considerations.

If you have an existing design or a partially-completed design, the power estimator file generated by the Quartus II software can provide input to the spreadsheet for your current design (refer to "Early Power Estimator File").

When the design is complete, the PowerPlay Power Analyzer tool in the Quartus II software provides an accurate estimation of power to help ensure that thermal and supply budgets are not violated.

The PowerPlay Early Power Estimator spreadsheets for each supported device family are available on the PowerPlay Early Power Estimator and Power Analyzer page of the Altera website.

Estimating power consumption early in the design cycle allows planning of power budgets and avoids surprises for designers developing the PCB.

For more information about power estimation and analysis, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook.*

Early Power Estimator File

When entering data into the Early Power Estimator spreadsheet, you must include the device resources, operating frequency, toggle rates, and other parameters. Specifying these values requires familiarity with the design. If you do not have an existing design, estimate the number of device resources used in your design and enter it manually. If you have an existing design or a partially completed design, you can generate a power estimator file.

First, compile your design in the Quartus II software. After compilation is complete, on the Project menu, click **Generate PowerPlay Early Power Estimator File**. This command instructs the Quartus II software to write out a power estimator Comma-Separated Value (**.csv**) file (or a text [**.txt**] file for older device families).

The PowerPlay Early Power Estimator spreadsheet includes the Import Data macro, which parses the information in the power estimation file and transfers it into the spreadsheet. If you do not want to use the macro, you can transfer the data into the Early Power Estimator spreadsheet manually.

If the existing Quartus II project represents only a portion of your full design, you should enter the additional resources used in the final design manually. You can edit the spreadsheet and add additional device resources after importing the power estimation file information.

Early Pin Planning and I/O Analysis

It is important to plan top-level FPGA I/O pins early, so board designers can start developing the PCB design and layout. The FPGA device's I/O capabilities influence pin locations and other types of assignments. In cases where the board design team specifies an FPGA pin-out, it is crucial that the pin locations be verified in the FPGA place-and-route software as soon as possible to avoid board design changes.

Traditionally, designers and system architects could not check the validity of FPGA pin assignments until the design was complete. You can now create a preliminary pin-out for an Altera FPGA using the Quartus II Pin Planner before the source code is designed, based on standard I/O interfaces (such as memory and bus interfaces) and any other I/O-related assignments defined by system requirements. Refer to "Creating a Top-Level Design File for I/O Analysis" on page 1–7. Quartus II I/O Assignment Analysis checks that the pin locations and assignments are supported in the target FPGA architecture. You can use **I/O Assignment Analysis** to validate I/O-related assignments that you make or modify throughout the design process.

The Pin Planner enables easy I/O pin assignment planning, assignment, and validation. Use the Pin Planner Package view to make pin location and other assignments using a device package view instead of pin numbers. The Pads view displays I/O pads in order around the silicon die to help you follow pad distance and pin placement guidelines. With the Pin Planner, you can identify I/O banks, voltage reference (VREF) groups, and differential pin pairings to help you through the I/O planning process. If migration devices are selected (including HardCopy devices) as described in "Device Migration Planning" on page 1–3, the Pin Migration view highlights pins that change function in the migration device when compared to the currently selected device. Selecting pins in the Device Migration view cross-probes to the rest of the Pin Planner, so you can use device migration information when planning your pin assignments. You can also configure board trace models of selected pins for use in "board-aware" signal integrity reports generated with the **Enable Advanced I/O Timing** option. This option ensures you get very accurate I/O timing analysis. You have the option to use a Microsoft Excel spreadsheet to start the I/O planning process if you normally use a spreadsheet in your design flow, and you can export a Comma-Separated Value (**.csv**) file containing your I/O assignments for spreadsheet use when all pins are assigned.

When planning is complete, the pin location information can be passed to PCB designers. The Pin Planner is tightly integrated with certain PCB design EDA tools, and can read pin location changes from these tools to check the suggested changes. It is important that pin assignments match between the Quartus II software and your schematic and board layout tools to ensure the design works correctly on the board where it is placed, especially if changes to the pin-out must be made. The system architect can use the Quartus II software to pass pin information to team members designing individual logic blocks, for better timing closure when they compile their design. When the design is complete, the Quartus II Fitter reports should be used for the final sign-off of pin assignments.

Starting FPGA pin planning early—before the HDL design is complete—improves the confidence in early board layouts, reduces the chance of error, and improves the design's overall time to market.

For more information about I/O assignment and analysis, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*. For more information about passing I/O information between the Quartus II software and third-party EDA tools, refer to the *Mentor Graphics PCB Design Tools Support* and *Cadence PCB Design Tools Support* chapters in the *I/O and PCB Tools* section in volume 2 of the *Quartus II Handbook*.

Creating a Top-Level Design File for I/O Analysis

Early in the design process, before the source code is created, the system architect typically has information about the I/O interfaces and IP cores that will be used in the design. You can use this information with the **Create/Import Megafunction** feature in the Pin Planner to specify details about the design I/O interfaces. Specifying these details allows you to create a top-level design file that includes all your I/O information, so you can analyze the I/O assignments in the Quartus II software.

The Pin Planner interfaces with the MegaWizard™ Plug-In Manager, and allows you to create or import custom megafunctions and IP cores that use I/O interfaces. Configure how they are connected to each other by specifying matching node names for selected ports in the **Set Up Top-Level Design File** dialog box. Make any other I/O-related assignments for these interfaces or other design I/O pins in the Pin Planner.

When you have entered as much information as possible, generate a top-level design file using the **Create Top-Level Design File** command. The Pin Planner creates virtual pin assignments for internal nodes, so internal nodes are not assigned to device pins during compilation. After analysis and synthesis of the newly generated top-level wrapper file, use the generated netlist to perform I/O Analysis with the **Start I/O Assignment Analysis** command.

You can use the I/O analysis results to change pin assignments or IP parameters and repeat the checking process until the I/O interface meets your design requirements and passes the pin checks in the Quartus II software. When this initial pin planning is complete, you can create a Quartus II Revision based on the Quartus II-generated netlist. You then have a choice for how to proceed: you can use the generated netlist to develop the top-level file for the actual design, or disregard the generated netlist and use the generated Quartus II Settings File (**.qsf**) with the actual design.

Selecting Third-Party EDA Tool Flows

Your complete FPGA design flow may include third-party EDA tools in addition to the Quartus II software. Determine which tools you want to use with the Quartus II software to ensure that they are supported and set up correctly, and that you are aware of any useful features or undesired limitations.

Synthesis Tools

You can synthesize your design using the Quartus II software's integrated synthesis tool or your preferred third-party synthesis tool. Different synthesis tools may give different results. If you want to select the best-performing tool for your application, you can experiment by synthesizing typical designs for your application and coding style and comparing the results. Be sure to perform placement and routing in the Quartus II software to get accurate timing analysis and logic utilization results. Results from synthesis are estimates before place-and-route and do not include logic that is treated as a black box for synthesis (such as megafunctions or Altera IP cores in some synthesis tools). In addition, these estimates do not take into account logic usage reduction achieved in the Quartus II Fitter through register packing or other Quartus II optimizations, such as Physical Synthesis, that may change both timing and resource utilization results.

Altera recommends that you use the most recent version of third-party synthesis tools, because tool vendors frequently add new features, fix tool issues, and enhance performance for Altera devices. The *Quartus II Software Release Notes* lists the version of each synthesis tool that is officially supported by that version of the Quartus II software.

Specify your synthesis tool in the New Project Wizard or the **EDA Tools Settings** page of the **Settings** dialog box to use the correct Library Mapping File (**.lmf**) for your synthesis netlist.

Synthesis tools may offer the capability to create a Quartus II project and pass constraints, such as the EDA tool setting, device selection, and timing requirements that you specified in your synthesis project. You can use this capability to save time when setting up your Quartus II project for placement and routing.

If you want to take advantage of an incremental compilation methodology, you should partition your design for synthesis and generate multiple output netlist files. Refer to "Incremental Compilation with Design Partitions" on page 1–13 for more information.

For more information about synthesis tool flows, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Simulation Tools

Altera provides the ModelSim-Altera simulator with Quartus II license subscriptions, and the Quartus II software can generate timing netlist files to support other third-party simulation tools.

If you use a third-party simulation tool, ensure that you use the software version that is supported with your Quartus II version. The *Quartus II Software Release Notes* list the version of each simulation tool that is officially supported with that particular version of the Quartus II software. Also ensure that you use the model libraries provided with your Quartus II software version. Libraries can change between versions, which might cause a mismatch with your simulation netlist.

Specify your simulation tool in the **EDA Tools Settings** page of the **Settings** dialog box to generate the appropriate output simulation netlist.

For more information about simulation tool flows, refer to the appropriate chapter in the *Simulation* section in volume 3 of the *Quartus II Handbook*.

Formal Verification Tools

The Quartus II software supports some formal verification flows. Consider whether your desired formal verification flow impacts the design and compilation stages of your design.

Using a formal verification flow can impact performance results because it requires that certain logic optimizations be turned off, such as register retiming, and forces hierarchy blocks to be preserved, which can restrict optimization. Formal verification treats memory blocks as black boxes. Therefore, it is best to keep memory in a separate hierarchy block so other logic does not get incorporated into the black box for verification. There are other restrictions that may also limit your design, so consult the documentation for details. If formal verification is important to your design, it is easier to plan for limitations and restrictions in the beginning than to make changes later in the design flow.

Specify your formal verification tool in the **EDA Tools Settings** page of the **Settings** dialog box to generate the appropriate output netlist.

For more information about formal verification flows, refer to the appropriate chapter in the *Formal Verification* section in volume 3 of the *Quartus II Handbook*.

Planning for On-Chip Debugging Options

Altera’s in-system debugging tools offer different advantages and trade-offs, so a particular debugging tool may work better for different systems and designers. It is beneficial to evaluate on-chip debugging options early in your design process, to ensure that your system board, Quartus II project, and design are all set up to support the appropriate options. Planning can reduce time spent during debugging and eliminates having to make changes later to accommodate your preferred debugging methodologies.

The Quartus II portfolio of verification tools includes the following in-system debugging features:

- SignalProbe incremental routing—This feature makes design verification more efficient by quickly routing internal signals to I/O pins without affecting the design. Starting with a fully routed design, you can select and route signals for debugging to either previously reserved or currently unused I/O pins.

- SignalTap® II Embedded Logic Analyzer—This logic analyzer helps you debug an FPGA design by probing the state of the internal signals in the design without the use of external equipment or extra I/O pins, while the design is running at full speed in an FPGA device. Defining custom trigger-condition logic provides greater accuracy and improves the ability to isolate problems. The SignalTap II Embedded Logic Analyzer does not require external probes or changes to the design files to capture the state of the internal nodes or I/O pins in the design; all captured signal data is conveniently stored in device memory until you are ready to read and analyze the data.
- Logic Analyzer Interface (LAI)—This interface enables you to connect and transmit internal FPGA signals to an external logic analyzer for analysis. You can use this feature to connect a large set of internal device signals to a small number of output pins for debugging purposes, and allows you to take advantage of advanced features in your external logic analyzer or mixed signal oscilloscope.
- In-System Memory Content Editor—This feature provides read and write access to in-system FPGA memories and constants through the JTAG interface, making it easy to test changes to memory contents and constant values in the FPGA while the device is functioning in a system.
- In-System Sources and Probes—This feature sets up customized register chains to drive or sample the instrumented nodes in your logic design, providing an easy way to input simple virtual stimuli and capture the current value of instrumented nodes. You can force trigger conditions set up using the SignalTap II Logic Analyzer, create simple test vectors to exercise your design without the use of external test equipment, and dynamically control run-time control signals with the JTAG chain.
- Virtual JTAG Megafunction—The sld_virtual_jtag megafunction allows you to build your own system-level debugging infrastructure, including both processor-based debugging solutions and debugging tools in software for system-level debugging. The sld_virtual_jtag megafunction can be instantiated directly in your HDL code to provide one or more transparent communication channels to access parts of your FPGA design using the JTAG interface of the device.

For more information about debugging tools, refer to the appropriate "Referenced Documents" on page 1–18. For an overview of debugging options that will help you decide which option to use, refer to the introduction in *Section V. In-System Design Debugging* in volume 3 of the *Quartus II Handbook*.

If you intend to use any of these features, you may have to plan for the features when developing your system board, Quartus II project, and design. The following paragraphs describe various factors to consider during your design planning stages.

The SignalTap II Embedded Logic Analyzer, Logic Analyzer Interface, In-System Memory Content Editor, In-System Sources and Probes, and Virtual JTAG megafunction require JTAG connections to perform in-system debugging. Plan your system and board with JTAG ports that are available for debugging.

The JTAG debugging features also require a small amount of additional logic resources to implement the JTAG hub logic. If you set up the appropriate feature early in your design cycle, you can include these device resources in your early resource estimations to ensure you do not overfill the device with logic.

The SignalTap II Embedded Logic Analyzer uses device memory to capture data during system operation. To ensure that you have enough memory resources to take advantage of this debugging technique, consider reserving device memory to be used during debugging.

To use incremental debugging with the SignalTap II Embedded Logic Analyzer, the **Full incremental compilation** option must be turned on. This option is on by default for projects created in the Quartus II software version 6.1 or later, but is not turned on automatically for existing projects. If incremental compilation is not enabled, you must recompile the entire design when you want to add debugging functions, or when you make certain changes to SignalTap II settings. Using incremental compilation with the SignalTap II Embedded Logic Analyzer greatly reduces the compilation time required for debugging.

SignalProbe and the Logic Analyzer Interface require I/O pins for debugging. Consider reserving I/O pins for debugging so that you do not have to change the design or board to accommodate debugging signals later. Keep in mind that the Logic Analyzer Interface can multiplex signals with design I/O pins if required. Ensure that your board supports some kind of debugging mode, where debugging signals do not affect system operation.

If you want to use the Virtual JTAG megafunction for custom debugging applications, you must instantiate it and incorporate it as part of the design process.

The In-System Sources and Probes feature also requires that you instantiate a megafunction in your HDL code. In addition, you have the option to instantiate the SignalTap II Embedded Logic Analyzer as a megafunction, so you can connect it to nodes in your design manually and ensure that the tapped node names do not change during synthesis. You can add the debugging block as a separate design partition for incremental compilation to minimize recompilation times.

To use the In-System Memory Content Editor for RAM or ROM blocks or the lpm_constant megafunction, ensure that you turn on the **Allow In-System Memory Content Editor** to capture and update content independently of the system clock option when you create the memory block in the MegaWizard Plug-In Manager.

Design Practices and HDL Coding Styles

In the development of complex FPGA designs, design practices and coding styles have an enormous impact on your device's timing performance, logic utilization, and system reliability. Follow Altera's recommendations to achieve the best synthesis and fitting results.

Design Recommendations

Use synchronous design practices to consistently meet your design goals. Problems with other design techniques include reliance on propagation delays in a device, incomplete timing analysis, and possible glitches. In a synchronous design, a clock signal triggers all events. As long as all of the registers' timing requirements are met, a synchronous design behaves in a predictable and reliable manner for all process, voltage, and temperature (PVT) conditions. You can easily target synchronous designs to different device families or speed grades.

Pay particular attention to clock signals, because they have a large effect on your design's timing accuracy, performance, and reliability. Problems with clock signals can cause functional and timing problems in your design. Use dedicated clock pins and clock routing for best results, and if PLLs are available in your target device, use the PLLs for clock inversion, multiplication, and division. For clock multiplexing and gating, use the dedicated clock control block or PLL clock switchover feature instead of combinational logic if these features are available in your device. If you must use internally-generated clock signals, register the output of any combinational logic used as a clock signal to reduce glitches.

The Design Assistant in the Quartus II software is a design-rule checking tool that enables you to check for design issues early in the design flow. The Design Assistant checks your design for adherence to Altera-recommended design guidelines or design rules. To run the Design Assistant, on the Processing menu, point to **Start** and click **Start Design Assistant**. To set the Design Assistant to run automatically during compilation, turn on **Run Design Assistant during compilation** in the **Settings** dialog box. You can also use third-party "lint" tools to check your coding style.

It is also helpful to understand your device's target architecture, so you can target your design to take advantage of those features. For example, it is important that control signals use the dedicated control signals in the device architecture, so in some cases you might be required to limit the number of different control signals used in your design to achieve the best results.

For more information about design recommendations and using the Design Assistant, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*. You can also refer to industry papers for more information about multiple clock design. For a good analysis, refer to *Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs* under **Papers** at www.sunburst-design.com.

Recommended HDL Coding Styles

HDL coding styles can have a significant effect on the quality of results for programmable logic designs. Use Altera's recommended coding styles to achieve optimal synthesis results. When designing memory and DSP functions, it is helpful to understand your device's target architecture so you can take advantage of the dedicated logic block sizes and configurations. Follow the coding guidelines for inferring megafunctions and targeting dedicated device hardware, such as memory and DSP blocks.

For specific HDL coding examples and recommendations, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. Refer to your synthesis tool's documentation for any additional tool-specific guidelines. In the Quartus II software, you can use the HDL examples in the Language Templates available from the right-click menu in the text editor.

Planning for Hierarchical and Team-Based Design

If you want to create a hierarchical design that can take advantage of the compilation-time savings and performance preservation of Quartus II incremental compilation, plan for an incremental compilation flow from the beginning of your design cycle. The following subsections describe the flat compilation flow, in which the design hierarchy is flattened without design partitions, and then the incremental compilation flows that use design partitions in top-down, bottom-up, or mixed design methodologies. Incremental compilation flows offer several advantages but require more design planning to ensure good quality of results. The last subsections discuss factors to consider when planning an incremental compilation flow: planning design partitions and creating a design floorplan.

For details about using the incremental compilation flows in the Quartus II software, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Flat Compilation Flow with No Design Partitions

In this compilation flow in the Quartus II software, the entire design is compiled together in a "flat" netlist. This flow is used if you do not create any design partitions. Your source code can have hierarchy, but the design is flattened during compilation and all of the design source code is synthesized and fit in the target device whenever the design is recompiled after any change in the design. By processing the entire design, the software performs all available logic and placement optimizations on the entire design to improve area and performance. You can use debugging tools in an incremental design flow, such as the SignalTap II Logic Analyzer, but you do not specify any design partitions to preserve design hierarchy during compilation.

The flat compilation flow is easy to use; you do not have to plan any design partitions. However, because the entire design is recompiled whenever there are any changes to the design, compilation times can be relatively long for large devices. In addition, you may find that the results for one part of the design change when you change a different part of your design.

The full incremental compilation option is turned on by default in the Quartus II software, so the project is ready for you to create design partitions for incremental compilation. If you do not create any lower-level design partitions, the entire design is considered as a single design partition, and the software uses a flat compilation flow.

Incremental Compilation with Design Partitions

In an incremental compilation flow, the system architect splits a large design into smaller partitions which can be designed separately. Team members can work on partitions independently, which can simplify the design process and reduce compilation time.

When hierarchical design partitions are well chosen and placed in the device floorplan, you can speed up your design compilation time while maintaining or even improving the quality of results.

You may want to use incremental compilation later in the design cycle when you are not interested in improving the majority of the design any further, and want to make changes to, or optimize, one specific block. In this case, you may want to preserve the performance of modules that are unmodified and reduce compilation time on subsequent iterations.

Incremental compilation may also be useful for both reducing compilation time and achieving timing closure. For example, you may want to specify which partitions should be preserved in subsequent incremental compilations and then recompile the other partitions with advanced optimizations turned on.

If a part of your design is not yet complete, you can create an empty partition for the incomplete part of the design while compiling the completed partitions. Then, save the results for the complete partitions while you work on the new part of the design.

Alternately, different designers or IP providers may be working on different blocks of the design using a team-based methodology, and you may want to combine these blocks in a bottom-up compilation flow.

When planning your design code and hierarchy, ensure that each design entity is created in a separate file so the entities remain independent when you make source code changes in the file. If you use a third-party synthesis tool, create separate Verilog Quartus Mapping (VQM) or EDIF netlists for each design partition in your synthesis tool. You may have to create separate projects within your synthesis tool, so the tool synthesizes each partition separately and generates separate output netlist files. Refer to your synthesis tool documentation for information about support for Quartus II incremental compilation. The netlists are then considered the source files for incremental compilation.

Top-Down Versus Bottom-Up Incremental Flows

The Quartus II incremental compilation feature supports both top-down and bottom-up compilation flows that are suitable for different design methodologies. You can also combine these flows in a mixed compilation flow. The following subsections briefly describe each of these compilation flows so that you can choose the flow that best meets your design requirements.

Top-Down Incremental Compilation Flow

With top-down compilation, one designer or project lead compiles the entire design in the software. Different designers or IP providers can design and verify different parts of the design, and the project lead can add design entities to the project as they are completed. You can also target optimizations on one part of the design while designating the rest of the design as “empty.” Regardless of the source for all the design logic, the project lead compiles and optimizes the top-level project as a whole.

Incremental compilation preserves the compilation results and performance of unchanged partitions in your design, greatly reducing design iteration time by focusing new compilations on changed design partitions only. New compilation results are then merged with the previous compilation results from unchanged design partitions. Additionally, you can target optimization techniques, such as physical synthesis, to specific design partitions while leaving other partitions untouched. You can also use this flow with empty partitions if parts of your design are incomplete or missing.

Bottom-Up and Team-Based Incremental Compilation Flow

Bottom-up design flows allow individual designers to complete the optimization of their design in separate projects and then integrate each lower-level project into one top-level project. Bottom-up methodologies include team-based design flows in which design partitions are created by team members in another location or by third-party IP providers.

Incremental compilation provides export and import features to enable bottom-up design methodologies. Designers of lower-level blocks can export the optimized netlist for their design, along with a set of assignments, such as LogicLock™ regions. The system architect then imports each design block as a design partition in a top-level project.

In bottom-up design flows, it is very important that the system architect provide guidance to designers of lower-level blocks to ensure that each partition uses the appropriate device resources. Because the designs are developed independently, each lower-level designer has no information about the overall design or how their partition connects with other partitions. This lack of information can lead to problems during system integration. The top-level project information, including pin locations, physical constraints, and timing requirements, should be communicated to the designers of lower-level partitions before they start their design.

The system architect can plan design partitions at the top level and use Quartus II incremental compilation to communicate information to lower-level designers through automatically-generated scripts. The **Generate bottom-up design partition scripts** option automates the process of transferring top-level project information to lower-level modules. The software provides a project manager interface for managing project information in the top-level design.

The scripts can create Quartus II projects for all the lower-level design blocks and pass all the relevant project assignments. Using these scripts makes it easier for designers of lower-level modules to implement the instructions from the project lead, and avoid conflicts between projects when importing and incorporating the projects into the top-level design. You can use this methodology to help reduce the need to further optimize the designs after integration and improve overall designer productivity and team collaboration.

Mixed Incremental Compilation Flow

You can combine top-down and bottom-up compilation flows to take advantage of top-down flows for part of your design, while importing parts of the design that are developed independently.

The top-down flow is generally simpler to perform than its bottom-up counterpart. For example, having to export and import lower-level designs is eliminated. A top-down approach also provides the design software with information about the entire design, so it can perform global placement optimizations when no part of the design is locked down to a specific location.

In a bottom-up design methodology, you must perform resource balancing and time-budgeting very carefully, because the software does not have any information about the other partitions in the top-level design when it compiles individual lower-level partitions. Using bottom-up compilation flows where required, in combination with top-down compilation flows to reduce compilation time and preserve results for other parts of the design, can be an effective way to improve your productivity.

Planning Design Partitions

Partitioning a design for an FPGA requires planning to ensure optimal results when the partitions are integrated, and ensure that each partition is placed well relative to other partitions in the device. Following Altera's recommendations for creating design partitions improves the overall quality of results. For example, registering partition I/O boundaries keeps critical timing paths inside one partition that can be optimized independently. When the design partitions are specified, you can use the Incremental Compilation Advisor to ensure that partitions meet Altera's recommendations.

Determining a timing budget before designers develop their individual blocks reduces the chance of timing problems during system integration. If you optimize lower-level partitions separately, any unregistered paths that cross between partitions are not optimized as an entire path. To ensure that the software correctly optimizes the input and output logic in each partition, you may be required to perform some manual timing budgeting. For each unregistered timing path that crosses between partitions, you should make timing assignments on the corresponding I/O path in each partition to constrain both ends of the path to the budgeted timing delay. Assigning a timing budget for each part of the connection ensures that the software optimizes paths appropriately so they meet the top-level design requirements.

It is important to plan and balance resource utilization. When performing incremental compilation, the software synthesizes each partition separately, with no data about the resources used in other partitions. Therefore, device resources can be overused in the individual partitions during synthesis, and the design may not fit in the target device when the partitions are merged.

In a bottom-up design flow in which designers optimize their lower-level designs and export them to a top-level design, the software also places and routes each partition separately. In some cases, partitions can use conflicting resources when combined at the top level. Balancing resource utilization between the design partitions avoids any problems with conflicting resources when all the partitions are integrated.

For guidelines on creating design partitions and organizing your source code, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan* chapter in volume 1 of the *Quartus II Handbook*.

Creating a Design Floorplan

To take full advantage of incremental compilation, you may be required to create a design floorplan to avoid conflicts between design partitions, and to ensure that each partition is placed well relative to other partitions. Creating location assignments for each partition ensures that no conflicts occur for locations between different partitions. In addition, a design floorplan helps to avoid a situation in which the Fitter is directed to place or replace a portion of the design in an area of the device where most resources have already been claimed. Without floorplan assignments, this situation can lead to increased compilation time and reduced quality of results.

You can use the Quartus II Timing Closure Floorplan or Chip Planner, depending on your target device, to create a design floorplan using LogicLock region assignments for each design partition. With a basic design framework for the top-level design, these floorplan editors allow you to view connections between regions, estimate physical timing delays on the chip, and move regions around the device floorplan. When you have compiled the full design, you can also view logic placement and locate areas of routing congestion to improve the floorplan assignments.

Good partition and floorplan design helps lower-level designs meet top-level design requirements when integrated with the rest of the design, reducing the time spent integrating and verifying the timing of the top-level design.

For details about creating placement assignments in the design floorplan, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*. For guidelines on creating a design floorplan for incremental compilation, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.

Fast Synthesis and Early Timing Estimation

It is much less costly to find design issues early in the design cycle than to find problems in the final timing closure stages. When the first version of the design source code is complete, you may want to perform a quick compilation to create a kind of silicon virtual prototype (SVP) that you can use to perform timing analysis.

If you synthesize with the Quartus II software, you can choose to perform a **Fast** synthesis, which reduces the compilation time but may give reduced quality of results. On the Assignments menu, click **Settings**. On the **Analysis & Synthesis Settings** tab, click **More Settings** and set the **Synthesis Effort**.

Regardless of your compilation flow, you can use the an Early Timing Estimate to perform a quick placement and routing, and a timing analysis of your design. On the Processing menu, point to **Start**, and click **Start Early Timing Estimate**. The software chooses a device automatically if required, places any LogicLock regions used to create a floorplan, finds a quick initial placement for all the design logic, and provides a useful estimate of the final design performance. If you have entered timing constraints, timing analysis reports on these constraints.

Early Timing Estimation is supported with both the TimeQuest and Classic Timing Analyzers. Use the TimeQuest Timing Analyzer with Synopsys Design Constraint (SDC) format constraints to enable advanced timing analysis capabilities that are not available in the Classic Timing Analyzer.

Designers of individual blocks in bottom-up design flows can use these features as they develop the design. Any issues highlighted in the lower level design blocks can be communicated to the system architect. Resolving these issues may require allocating additional device resources to the individual block or changing its timing budget.

A top-level designer can also use fast synthesis and early timing estimation to prototype the entire design. Incomplete partitions can be marked as empty in an incremental compilation flow, while the rest of the design is compiled to get an early timing estimate and detect any problems with design integration.

A system architect can use early timing estimation along with design partition scripts (as described in "Bottom-Up and Team-Based Incremental Compilation Flow" on page 1–15) to pass additional constraints to lower-level designers, and provide more information about the other partitions in the design. This information can be especially useful to optimize cross-partition paths. Running early timing estimations helps designers find and resolve design problems during the early design stages.

Conclusion

Modern FPGAs support large, complex designs with fast timing performance. By planning several aspects of your design early in the process, you can reduce unnecessary time spent handling issues in later stages of the process. You can use various features of the Quartus II software to quickly plan your design and achieve the best possible results. Following the guidelines presented in this chapter can improve productivity, which reduces the design cost and improves the final product's time to market.

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *AN 386: Using the MAX II Parallel Flash Loader with the Quartus II Software*
- *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*
- *Cadence PCB Design Tools* chapter in volume 2 of the *Quartus II Handbook*
- *Configuration Handbook*
- *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Design Debugging Using In-System Sources and Probes* chapter in volume 3 of the *Quartus II Handbook*
- *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*
- *Formal Verification* section in volume 3 of the *Quartus II Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*

- *In-System Debugging Using External Logic Analyzers* chapter in volume 3 of the *Quartus II Handbook*
- *In-System Updating of Memory and Constants* chapter in volume 3 of the *Quartus II Handbook*
- *Introduction to the Quartus II Software*
- *Mentor Graphics PCB Design Tools Support* chapter in volume 2 of the *Quartus II Handbook*
- *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*
- *Simulation* section in volume 3 of the *Quartus II Handbook*
- *sld_virtual_jtag Megafunction User Guide*
- *Synthesis* section in volume 1 of the *Quartus II Handbook*

Document Revision History

Table 1–1 shows the revision history for this chapter.

Table 1–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v.9.0.0	No change to content	Updated for the Quartus II 9.0 software release.
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	Updated for the Quartus II 8.1 software release.
May 2008 v8.0.0	■ Organization changes ■ Added “Creating Design Specifications” section ■ Added reference to new details in the *In-System Design Debugging* section of volume 3 ■ Added more details to the “Design Practices and HDL Coding Styles” section ■ Added references to the new *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter ■ Added reference to the Quartus II Language Templates	Updated for the Quartus II 8.0 software release and related documentation; expanded and improved organization of topic coverage.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

2. Quartus II Incremental Compilation for Hierarchical and Team-Based Design

QII51015-9.0.0

Introduction

The ability to iterate rapidly through FPGA design and debugging stages is critical. The Quartus® II software introduced the FPGA industry's first true incremental design and compilation flow, with the following benefits:

- Preserves the results and performance for unchanged logic in your design as you make changes elsewhere
- Reduces design iteration time by up to 70%, so you can perform more design iterations per day and achieve timing closure efficiently
- Integrates with third-party synthesis software's incremental synthesis flows
- Facilitates modular hierarchical and team-based design flows

"Deciding Whether to Use an Incremental Compilation Flow" on page 2–2 provides an overview of the Quartus II design flow with and without incremental compilation to help you decide if you should take advantage of optional incremental flows for your project. The remainder of the chapter includes the following sections:

- "Quick Start Guide—Summary of Incremental Compilation" on page 2–7
- "Deciding which Design Blocks Should Be Design Partitions" on page 2–9, including integration with third-party synthesis tools
- "Creating Design Partition Assignments" on page 2–16, including using the Design Partition Planner
- "Setting the Netlist Type for Design Partitions" on page 2–19
- "Creating a Design Floorplan with LogicLock Location Assignments" on page 2–25
- "Exporting and Importing Partitions" on page 2–29
- "Importing SDC Constraints from Lower-Level Partitions" on page 2–45
- "Recommended Design Flows and Compilation Application Examples" on page 2–49, including the steps required to carry out the following tasks:
  - "Reducing Compilation Time When Changing a Source File for One Partition"
  - "Preserving Results for Some Partitions Before Adding Other Partitions"
  - "Optimizing a Timing-Critical Partition to Achieve Timing Closure"
  - "Debugging Incrementally with the SignalTap II Logic Analyzer"
  - "Implementing a Team-Based Bottom-Up Design Flow"
  - "Performing Design Iteration in a Bottom-Up Design Flow"
  - "Creating Hard-Wired Macros (or Precompiled Design Blocks) for IP Reuse"
  - "Using an Exported Partition to Send a Design without Including Source Files"
- "Incremental Compilation Restrictions" on page 2–60
- "Scripting Support" on page 2–69

Quartus II incremental compilation supports the Arria® GX, Stratix®, and Cyclone® series of devices, with limited support for HardCopy® ASICs (for details, refer to "HardCopy Compilation and Migration Flows" on page 2–64.

Deciding Whether to Use an Incremental Compilation Flow

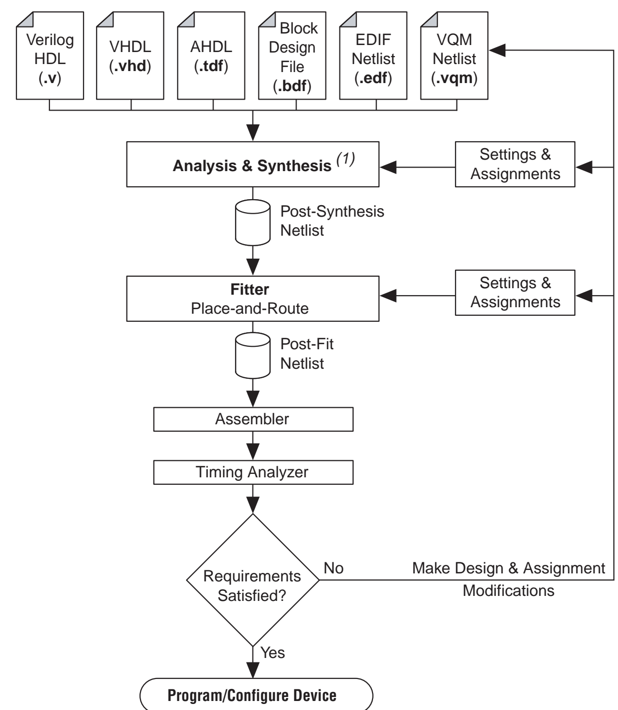
Quartus II incremental compilation enhances the standard Quartus II design flow by allowing you to preserve the satisfactory compilation results and performance of unchanged parts of your design. This section outlines the flat compilation flow with no design partitions and the incremental flow when you divide the design into partitions, and explains the differences. The section also explains when a flat compilation flow is satisfactory, and highlights some of the reasons you might want to create design partitions and use the incremental flow.

The full incremental compilation option is turned on by default in the Quartus II software, so the project is ready for you to create design partitions for incremental compilation. If you do not create any design partitions, the software uses a flat compilation flow, and you cannot make incremental changes within the design. Synthesizing and fitting the design with no partitions is similar to compiling with the Quartus II full incremental compilation option turned off. However, Altera recommends leaving the option turned on because doing so enables some incremental flows, such as debugging with the SignalTap® II Embedded Logic Analyzer, even if you do not create design partitions.

Flat Compilation Flow with No Design Partitions

Figure 2–1 shows the compilation flow with no design partitions.

Figure 2–1. Quartus II Compilation Flow with No Design Partitions



Note to Figure 2–1:

(1) When you use EDIF or VQM netlists created by third-party EDA synthesis tools, Analysis and Synthesis creates the design database, but logic synthesis and technology mapping are performed only for black boxes.

In the default flat compilation flow with no design partitions, all the source code is processed with the Analysis & Synthesis module, and all the logic is placed and routed by the Fitter module whenever the design is recompiled after a change in any part of the design. One reason for this behavior is to obtain optimal quality of results. By processing the entire design, the compiler can perform global optimizations to improve area and performance.

You can use a flat compilation flow for small designs, such as designs in CPLD devices or low-density FPGA devices, when the timing requirements are met easily with a single compilation. A flat design is satisfactory when compilation time and preserving results for timing closure are not concerns.

Quartus II Smart Compilation

The Quartus II software also includes a feature called Smart Compilation, which should not be confused with incremental compilation. In any Quartus II compilation flow, you can use Smart Compilation to allow the compiler to determine which compiler stages are required, based on the changes made to the design since the last smart compilation, and then skip any stages that are not required. For example, when Smart Compilation is on, the compiler skips the Analysis & Synthesis module if all the design source files are unchanged. Smart Compilation skips only entire compiler stages. It cannot make incremental changes within a given stage of the compilation flow. To turn on Smart Compilation, on the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Process Settings** and click **Use Smart Compilation**.

Incremental Compilation Flow with Design Partitions

Using design partitions allows you to preserve the results and performance for unchanged logic in your design as you make changes elsewhere, and reduce compilation time. Incremental compilation is recommended for large designs and high resource densities, as well as designs that require high performance relative to the speed of the device architecture. The feature also facilitates team-based design environments, allowing designers to create and optimize design blocks independently.

Incremental compilation supports top-down incremental compilation, in which one designer manages a single project for the entire design, and bottom-up incremental compilation, in which each design block can be developed and optimized independently, and combinations of the two. See "Top-Down versus Bottom-Up Compilation Flows" on page 2–6 for more details. To take advantage of the incremental compilation flow, start by splitting the design along any of its hierarchical boundaries into blocks called design partitions. Refer to "Deciding which Design Blocks Should Be Design Partitions" on page 2–9 and "Creating Design Partition Assignments" on page 2–16 for more details. The Quartus II software synthesizes each individual hierarchical design partition separately, then merges the partitions into a complete netlist for subsequent stages of the compilation flow. When recompiling the design, you can use source code, post-synthesis results, or post-fitting results for each partition. If you want to preserve the Fitter results, you can keep just the placement results, or keep both the placement and routing results.

You can use incremental compilation toward the end of your design cycle when you do not have to improve the majority of the design any further and want to make changes to or optimize one specific block. In this case, you can preserve the performance of modules that meet their requirements to make timing closure easier and reduce compilation time on subsequent iterations. You can also recompile the other partitions with advanced optimizations turned on to improve their performance without affecting the preserved partitions.

Part of your design may be incomplete or developed by a different designer or IP provider. You can create an empty partition for this part of the design while compiling the completed partitions, and then save the results for the complete partitions while you optimize the imported part of the design. Alternatively, different designers or IP providers may want to develop and optimize different blocks of the design independently, and you can combine them in a bottom-up compilation flow.

For more detailed examples that describe recommended design flows to take advantage of the incremental compilation features, refer to “Recommended Design Flows and Compilation Application Examples” on page 2–49.

Table 2–1 shows a summary of the impact of incremental compilation on your compilation results.

Table 2–1. Impact Summary of Using Incremental Compilation

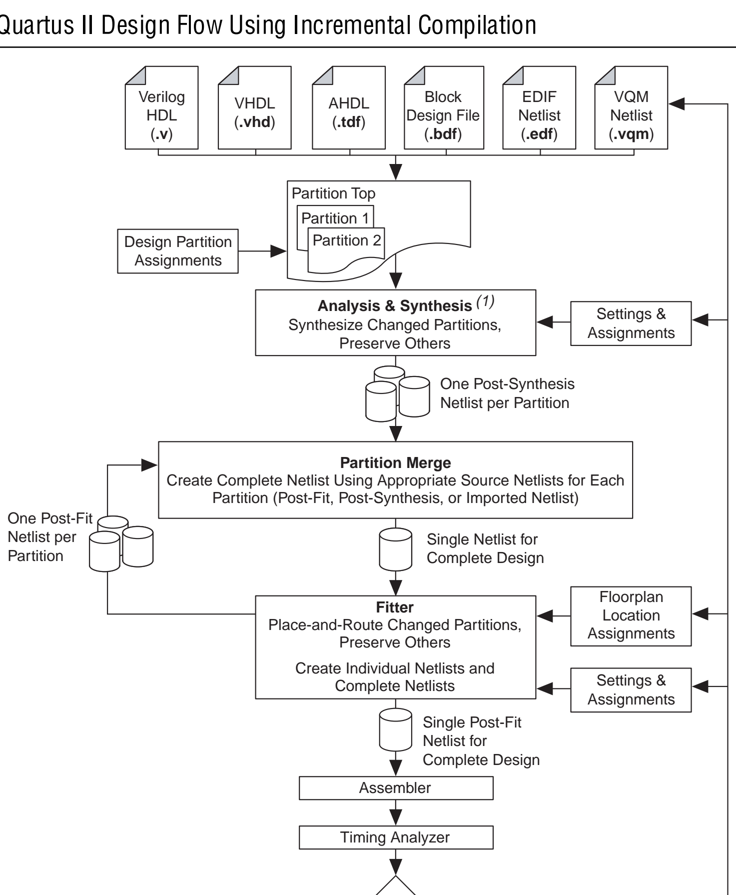
Characteristic	Impact of Incremental Compilation with Design Partitions
Compilation Time Savings	Typically saves 50-70% of compilation time when post-fit netlists are preserved; there are savings in both Quartus II integrated synthesis and the Fitter.
Performance Preservation	Excellent when critical paths are contained within a partition, because you can preserve post-fitting information for unchanged partitions.
Node Name Preservation	Preserves post-fitting node names for unchanged partitions.
Area Changes	The area (logic resource utilization) might increase because cross-boundary optimizations are no longer possible, and placement and register packing are restricted.
f_{MAX} Changes	The design’s maximum frequency might be reduced because cross-boundary optimizations are no longer possible. If the design is partitioned and the floorplan location assignments are created appropriately, there is no negative impact on f_{MAX}.
Floorplan Creation	Recommended for critical partitions to ensure the best quality of results when making design changes. Required when importing partitions from another Quartus II project to avoid placement conflicts.

If you use the incremental compilation feature at any point in your design flow, you should start planning for incremental compilation from the start of your design development. It is easier to accommodate the guidelines for partitioning and creating a floorplan if you start planning at the beginning of your design cycle.

Refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook* for more information and recommendations.

Figure 2–2 shows a block diagram of the Quartus II design flow using incremental compilation with design partitions.

Figure 2–2. Quartus II Design Flow Using Incremental Compilation



Note to Figure 2–2:

(1) When you use EDIF or VQM netlists created by third-party EDA synthesis tools, Analysis and Synthesis creates the design database, but logic synthesis and technology mapping are performed only for black boxes.

The diagram in Figure 2–2 shows a top-level partition and two lower-level partitions. If any part of the design changes, Analysis and Synthesis processes the changed partitions and keeps the existing netlists for the unchanged partitions. After completion of Analysis and Synthesis, there is one post-synthesis netlist for each partition.

The Partition Merge step creates a single, complete netlist that consists of post-synthesis netlists, post-fit netlists, and netlists imported from other Quartus II projects, depending on the netlist type you specify for each partition.

The Fitter then processes the merged netlist, preserving the placement or placement and routing of unchanged partitions, refitting only those partitions that have changed. The Fitter generates the complete netlist for use in further stages of the compilation flow, including timing analysis and programming file generation. It also generates individual netlists for each partition so the Partition Merge stage can use the post-fit netlist to preserve the placement and routing of a partition if you specify to do so in future compilations. The Quartus II software does not resynthesize or refit unchanged partitions that have a netlist type assignment that specifies the use of a post-synthesis or post-fit netlist, respectively.

For more information about using the incremental compilation feature, refer to the “Quick Start Guide—Summary of Incremental Compilation” on page 2–7.

Quartus II software versions before version 8.1 included an “incremental synthesis only” option that did not preserve placement results. This option was removed beginning with version 8.1. You can use a post-synthesis netlist to preserve synthesis results with full incremental compilation.

Top-Down versus Bottom-Up Compilation Flows

With top-down compilation, one designer compiles the entire design in the software. You can use a top-down flow to optimize all blocks of the design together, or to optimize one or more critical design blocks or IP cores before adding the rest of the design. You can preserve fitting results and performance for completed blocks while other parts of the design are changing, which also reduces the compilation times for each design iteration. Different designers or IP providers can create and verify HDL code separately, but one person (generally the project lead or system architect) compiles and optimizes the design as a single top-level Quartus II project.

With bottom-up design flows, individual designers or IP providers can complete the development and optimization of their design in separate Quartus II projects and then integrate each lower-level project into one top-level project. Incremental compilation provides export and import features to enable this design methodology. Designers of lower-level blocks can export the optimized placed and routed netlist for their design, along with a set of assignments such as LogicLock™ regions. The project lead then imports each design block as a design partition in a top-level project.

A bottom-up design flow permits the reuse of compilation results from another project, with the ultimate goals of performance preservation and compilation time reduction in a multiple-designer environment.

A bottom-up design flow also has the following potential drawbacks that require careful planning:

- Achieving timing closure for the full design may be more difficult if you compile the lower-level blocks independently without any information about each other. This problem may be avoided by careful timing budgeting and special design rules, such as always registering the ports at the module boundaries.
- For the same reason, resource budgeting and allocation may be required to avoid resource conflicts and overuse. Floorplan creation is typically very important in a bottom-up flow.

 © March 2009 Altera Corporation

In a bottom-up design flow, the top-level project lead can do much of the design planning, and then pass constraints on to the designers of lower-level blocks using Quartus II-generated design partition scripts.

When using the full incremental compilation flow, users who traditionally relied on a bottom-up approach for the sole reason of performance preservation can now employ a top-down approach to achieve the same goal. This ability is important for the following two reasons:

- A top-down flow is generally simpler to perform than its bottom-up counterpart. For example, having to export and import lower-level designs is eliminated.
- A top-down approach provides the design software with information about the entire design so it can perform some global placement and routing optimizations. Therefore, it is often easier to ensure good quality of results with a top-down flow than with a bottom-up flow.

The Quartus II incremental compilation feature is very flexible and supports numerous design methodologies. You can mix top-down and bottom-up flows within a single project. If the top-level design includes one or more design blocks that are optimized by different designers or IP providers, you can import those blocks (using a bottom-up methodology) into a project that also includes partitions for a top-down incremental methodology. In addition, as you perform timing closure for a design, you can create a subproject for one block of the design to be optimized by another designer in a separate Quartus II project, and pass information about the rest of the design to the subproject to obtain the best results. By following a mixed design methodology, you can take advantage of the team-based capabilities of a bottom-up flow while maintaining the advantages of a top-down flow for most of the design logic.

Importing partitions is not supported in HardCopy or FPGA companion device compilations when there is a migration device setting. You cannot use a bottom-up methodology if you migrate to a HardCopy ASIC. For details, refer to “HardCopy Compilation and Migration Flows” on page 2–64.

The following Quick Start Guide describes the top-down compilation flow because it is the more commonly used and easy-to-use incremental compilation flow. For more information about exporting design partitions from separate Quartus II projects and importing them as part of a bottom-up design flow, refer to “Exporting and Importing Partitions” on page 2–29.

Quick Start Guide—Summary of Incremental Compilation

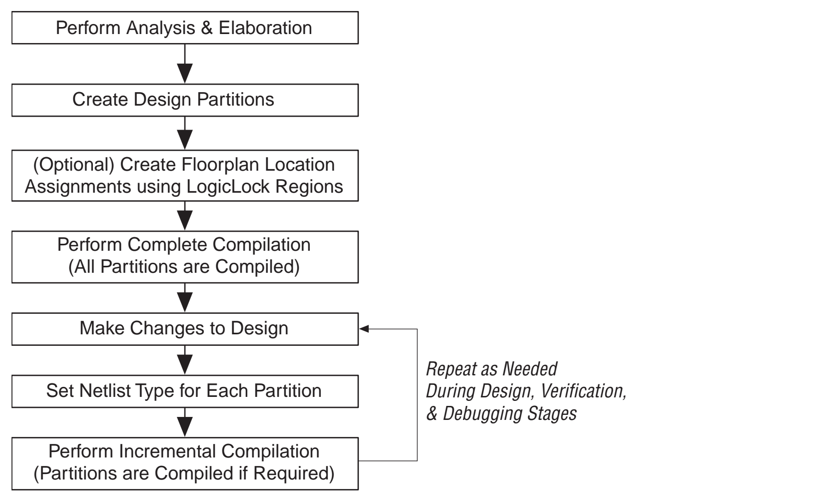
This section provides a summary of the steps required to perform a simple incremental compilation flow. Detailed descriptions for many of these steps are included in later sections of this chapter. For specific examples of design flows that take advantage of the incremental compilation features, refer to “Recommended Design Flows and Compilation Application Examples” on page 2–49.

For a step-by-step introduction to implementing an incremental compilation flow in the Quartus II software, on the Help menu, click **Tutorial**. After the introduction, choose **Module 7: Incremental Compilation** to view design flows for incremental compilation.

The flow chart in Figure 2–3 illustrates the incremental compilation flow when all partitions are contained in one top-level project. The following subsections describe the steps in the flow.

First, prepare the design for incremental compilation and perform a full compilation. Then proceed to verify or debug your design and make design changes as required. When you perform additional design iterations, choose which netlists to reuse and perform incremental compilations.

Figure 2–3. Summary of Incremental Compilation Flow



Preparing a Design for Incremental Compilation

To set up your design for incremental compilation, perform the following steps:

1. Elaborate the design. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration**, or run any compilation flow (such as a full compilation) that includes this step. Elaboration is the part of the synthesis process that identifies your design's hierarchy.
2. Create partitions in your design by designating specific instances as design partitions. In the user interface, you can right-click an instance in the Project Navigator, point to **Design Partition**, and click **Set as Design Partition**. Alternatively, on the Tools menu you can open the Design Partition Planner and right-click on a design block to use the Auto-Partition feature that creates partitions based on the size and connectivity of the hierarchical design blocks. Refer to "Creating Design Partition Assignments" on page 2–16 for details.

   Refer to "Deciding which Design Blocks Should Be Design Partitions" on page 2–9 for an explanation of design partitions and what part of your design can be specified as a design partition.
3. If required for your design flow, use LogicLock regions to make location assignments for each partition to create a design floorplan. If timing-critical design blocks change with future compilations, assigning the partition to a physical region on the device can improve results. Refer to the section "Creating a Design Floorplan with LogicLock Location Assignments" on page 2–25 for details about these assignments.

4. Compile the design. The first compilation after making partition assignments is a complete compilation that prepares the design for subsequent incremental compilations.

Compiling a Design Using Incremental Compilation

After compiling the design once and then making changes, take advantage of incremental compilation to recompile only the changed parts of the design. To do this, perform the following general steps:

1. Choose which of the following compilation results you intend to reuse for each partition.
   - To preserve previous placement results for a partition, set the Netlist Type assignment for that partition to **Post-Fit**.
     - To preserve routing information as well, set the **Fitter Preservation Level** to **Placement and Routing**.
   - To save only the synthesis results, set the Netlist Type assignment for that partition to **Post-Synthesis**.

   Partitions with design changes are recompiled automatically with these Netlist Type settings. You can also direct the software to recompile from the source code by choosing the **Source File** netlist type.

   If you do not want to compile a specific partition at all, set its Netlist Type to **Empty**.

   For details about setting these partition properties, refer to "Setting the Netlist Type for Design Partitions" on page 2–19.

2. Compile the design. The Quartus II software preserves the results you specified in step 1.

Deciding which Design Blocks Should Be Design Partitions

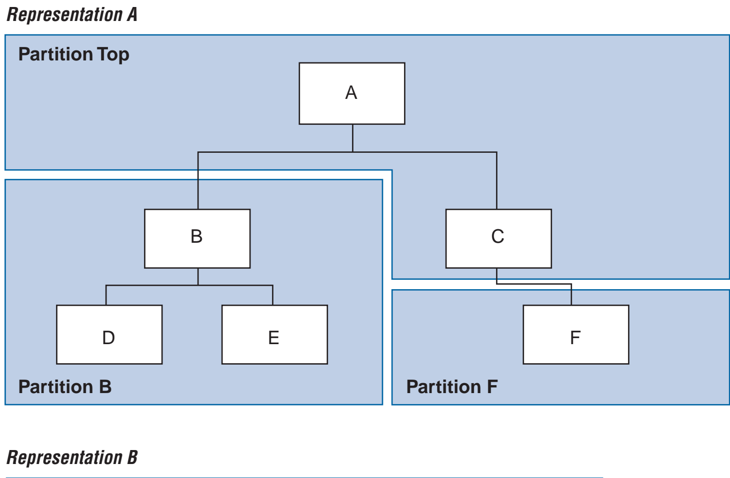
It is a common design practice to create modular or hierarchical designs in which you develop each design entity separately, and then instantiate them in a higher-level entity, forming a complete design. The Quartus II software does not consider each design entity or instance to be a design partition for incremental compilation automatically; instead, you must designate one or more design hierarchies below the top-level project as a design partition. Creating partitions prevents the compiler from performing optimizations across partition boundaries, as discussed in "Impact of Design Partitions on Design Optimization" on page 2–11. However, this allows for separate synthesis and placement for each partition, making incremental compilation possible.

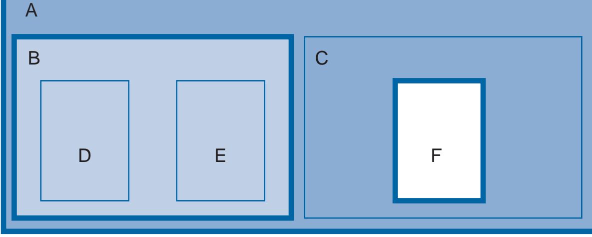
Partitions must have the same boundaries as hierarchical blocks in the design because a partition cannot be a portion of the logic within a hierarchical entity. When you declare a partition, every hierarchical instance within that partition becomes part of the same partition. You can create new partitions for hierarchical instances within an existing partition, in which case the instances within the new partition are no longer included in the higher-level partition, as described in the following example.

In Figure 2–5, hierarchical instances **B** and **F** form partitions in the complete design, which is made up of instances **A**, **B**, **C**, **D**, **E**, and **F**. The shaded boxes in Representation A indicate design partitions in a "tree" representation of the hierarchy. In Representation B, the lower-level instances are represented inside the higher-level instances, and the partitions are illustrated with different colored shading. The top-level partition, called Top, automatically contains the top-level entity in the design, and contains any logic not defined as part of another partition. The design file for the top level may be just a wrapper for the hierarchical instances below it, or it may contain its own logic. In this example, the partition for top-level entity **A** also includes the logic in one of its lower-level instances, **C**. Because instance **F** is contained in its own partition, it is not treated as part of the top-level partition. Another separate partition, **B**, contains the logic in instances **B**, **D**, and **E**.

Figure 2–4. Partitions in a Hierarchical Design

Representation A





You can make partition assignments to any design instance. The instance can be defined in HDL or schematic design, or come from a third-party synthesis tool as a VQM or EDIF netlist instance.

To take advantage of incremental compilation when source files change, create separate design files for each partition. If you define two different entities as separate partitions but they are in the same design file, you cannot maintain incremental compilation because the software would have to recompile both partitions if you changed either entity in the design file. Similarly, if two partitions rely on the same lower-level entity definition, changes in that lower-level affect both partitions.

The remainder of this section provides information to help you choose which design blocks you should assign as partitions.

Impact of Design Partitions on Design Optimization

The boundaries of your design partitions can impact the design's quality of results. Creating partitions prevents the compiler from performing logic optimizations across partition boundaries, which allows the software to synthesize and place each partition separately in an incremental flow. Therefore, consider partitioning guidelines to help reduce the effect of partition boundaries.

Whenever possible, register all inputs and outputs of each partition. This helps avoid any delay penalty on signals that cross partition boundaries and keeps each register-to-register timing path within one partition for optimization. In addition, minimize the number of paths that cross partition boundaries. If there are timing-critical paths that cross partition boundaries, rework the partitions to avoid these inter-partition paths. Including as many of the timing-critical connections as possible inside a partition allows you to effectively apply optimizations to that partition to improve timing, while leaving the rest of the design unchanged. In addition, avoid constant partition inputs and outputs, because to maintain incremental behavior, the software cannot use the constants to optimize logic on either side of the partition boundary.

The Design Partition Planner can help you make good assignments, as described in "Creating Design Partition Assignments" on page 2–16. The following sections describe tools you can use after compilation to analyze the partition assignments.You can view "Partition Statistics Reports", including information about the number of I/O connections and how many are unregistered or driven by a constant value, in the partition statistics reports. You can also create "Partition Timing Reports" and refer to the "Incremental Compilation Advisor"for analysis and guidelines.

If critical timing paths cross partition boundaries, you can perform timing budgeting and make timing assignments to constrain the logic in each partition so the entire timing path meets its requirements. In addition, because each partition is optimized independently during synthesis, you may have to perform some resource balancing to ensure that each partition uses an appropriate number of device resources. With bottom-up compilation flows in which design partitions are compiled separately from the top level, there may be conflicts related to global routing resources for clock signals when the design is imported into the top. You can use the ALTCLK_CTRL megafunction to instantiate a clock control block and connect it appropriately in both the bottom and top-level projects, or find the compiler-generated clock control node in your design and make clock control location assignments in the Assignment Editor.

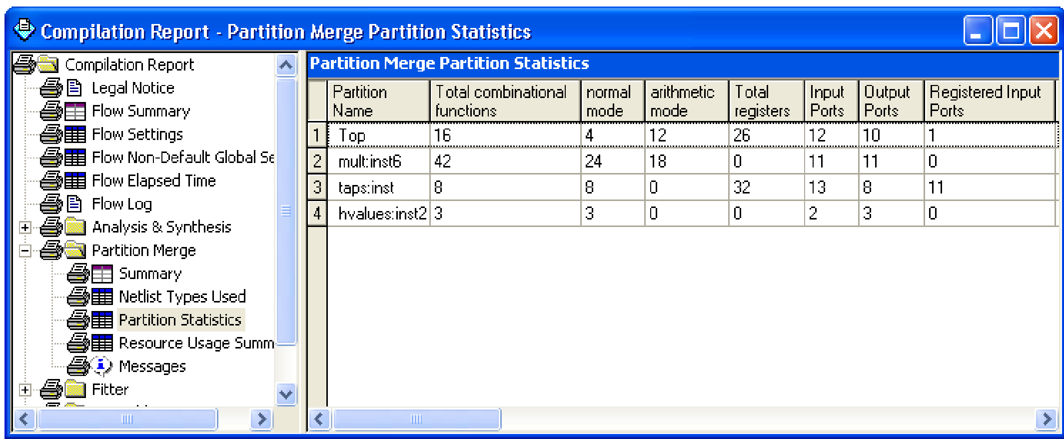
For more partitioning guidelines and specific recommendations for fixing common design issues, as well as information on resource balancing, global signal usage, and timing budgeting, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.

Partition Statistics Reports

After compilation, you can view statistics about design partitions in the Partition Merge Partition Statistics compilation report and the **Statistics** tab in the **Design Partitions Properties** dialog box.

The **Partition Statistics** page under the **Partition Merge** folder of the Compilation Report lists statistics about each partition. The statistics for each partition (each row in the table) include the number of logic cells it contains, as well as the number of input and output pins it contains and how many are registered or unconnected. This report is useful when optimizing your design partitions in a top-down compilation flow, or when you are compiling the top-level design in a bottom-up compilation flow, ensuring that the partitions meet the guidelines presented in the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*. Figure 2–5 shows the report window.

Figure 2–5. Partition Merge Partition Statistics Report

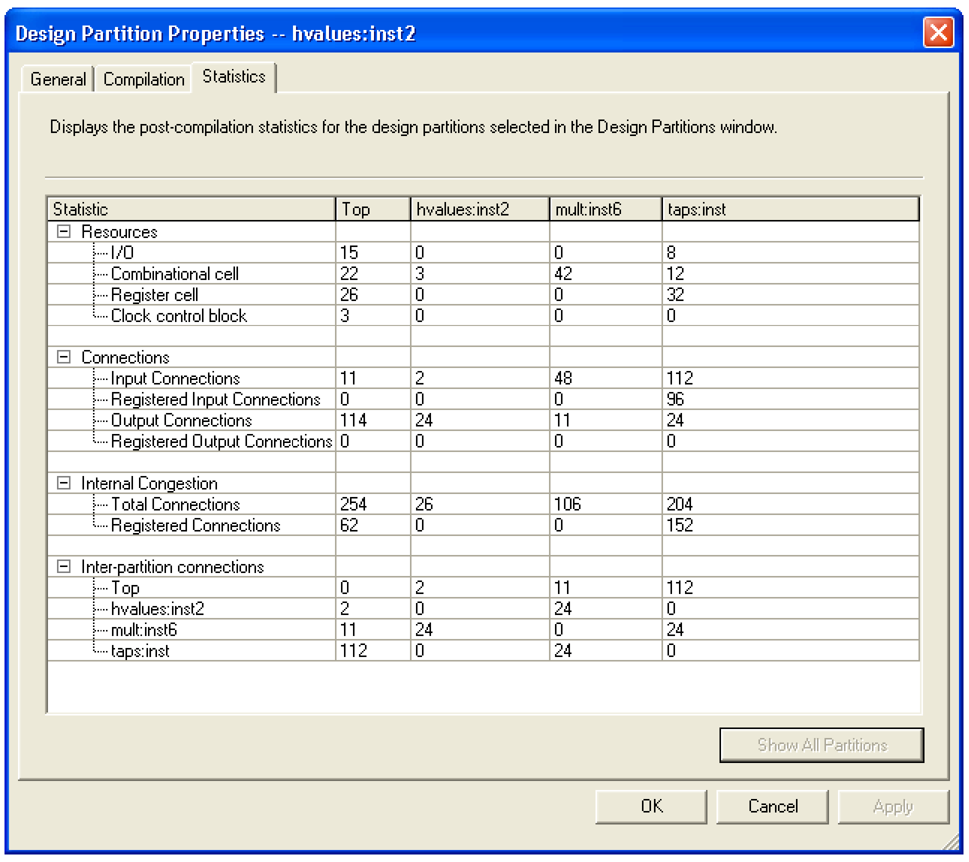


	Partition Name	Total combinational functions	normal mode	arithmetic mode	Total registers	Input Ports	Output Ports	Registered Input Ports
1	Top	16	4	12	26	12	10	1
2	mult:inst6	42	24	18	0	11	11	0
3	taps:inst	8	8	0	32	13	8	11
4	hvalues:inst2	3	3	0	0	2	3	0

You can also view statistics about the resource and port connections for a particular partition on the **Statistics** tab of the **Design Partition Properties** dialog box. On the Assignments menu, click **Design Partitions Window**. Right-click on a partition and click **Properties** to open the dialog box. Click **Show All Partitions** to view all the partitions in the same report (Figure 2–6).

 © March 2009 Altera Corporation

Figure 2–6. Statistics Tab in the Design Partitions Properties Dialog Box



Partition Timing Reports

You can generate a Partition Timing Overview report and a Partition Timing Details report by clicking **Report Partitions** in the Tasks pane in the TimeQuest Timing Analyzer or using the `report_partitions` Tcl command.

The Partition Timing Overview report shows the total number of failing paths for each partition and the worst-case slack for any path involving the partition.

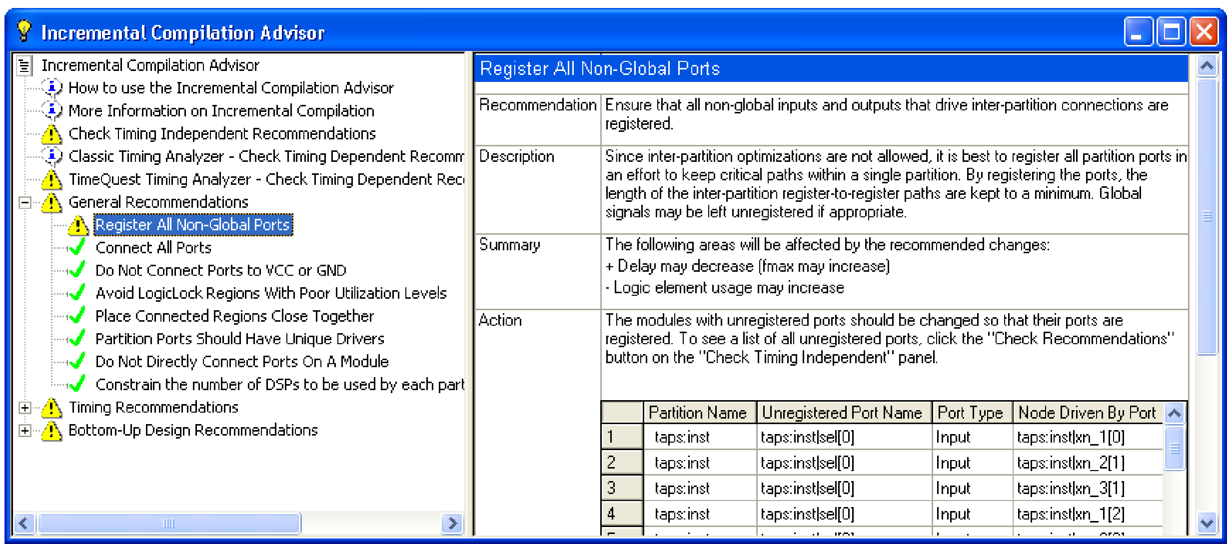
The Partition Timing Details report shows the number of failing partition-to-partition paths and worst-case slack for partition-to partition paths to provide a more detailed breakdown of where the critical paths in the design are with respect to design partitions.

Incremental Compilation Advisor

You can use the Incremental Compilation Advisor to check that your design follows Altera's recommendations presented for creating design partitions and floorplan location assignments. On the Tools menu, point to **Advisors** and click **Incremental Compilation Advisor**.

As shown in Figure 2–7, recommendations are split into **General Recommendations** that apply to all compilation flows and **Bottom-Up Design Recommendations** that apply to advanced bottom-up design methodologies. Each recommendation provides an explanation, describes the effect of the recommendation, and provides the action required to make the suggested change. In some cases, there is a link to the appropriate Quartus II settings page where you can make a suggested change to assignments or settings. The relevant timing-independent recommendations for the design are also listed in the Design Partitions Window and the LogicLock Regions Window.

Figure 2–7. Incremental Compilation Advisor



To check whether the design follows the recommendations, go to the **Timing Independent Recommendations** page or the **Timing Dependent Recommendations** page, and click **Check Recommendations**. For large designs, these operations can take a few minutes. After you perform a check operation, symbols appear next to each recommendation, as shown in Figure 2–8, to indicate whether the design or project setting follows the recommendations, or if some or all of the design or project settings do not follow the recommendations. Refer to the Legend on the **How to use the Incremental Compilation Advisor** page in the advisor for more information.

For some items in the Advisor, if your design does not follow the recommendation, the **Check Recommendations** operation lists any parts of the design that could be improved. For example, if not all of the partition I/O ports follow the **Register All Ports** recommendation, the advisor displays a list of unregistered ports with the partition name and the node name associated with for the port.

When the advisor provides a list of nodes, you can right-click on a node and click **Locate** to cross-probe to other Quartus II features such as the RTL Viewer, Chip Planner, or the design source code in the text editor.

Opening a new TimeQuest report resets the Incremental Compilation Advisor results, so you must rerun the Check Recommendations process.

Using Partitions with Third-Party Synthesis Tools

If you are using a third-party synthesis tool, set up your tool to create a separate VQM or EDIF netlist for each hierarchical partition. In the Quartus II software, assign the top-level entity from each netlist to be a design partition. The VQM or EDIF netlist file is treated as the source file for the partition in the Quartus II software.

Synopsys Synplify Pro/Premier and Mentor Graphics Precision RTL Plus

The Synplify Pro and Synplify Premier software include the MultiPoint synthesis feature to perform incremental synthesis for each design block assigned as a Compile Point in the user interface or a script. The Precision RTL Plus software includes an incremental synthesis feature that performs block-based synthesis based on Partition assignments in the source HDL code. These features provide automated block-based incremental synthesis flows and create different output netlist files for each block when set up for an Altera device.

Using incremental synthesis within your synthesis tool ensures that only those sections of a design that have been updated are resynthesized when the design is compiled, reducing synthesis run time and preserving the results for the unchanged blocks. You can change and resynthesize one section of a design without affecting other sections of the design.

For more information about these incremental synthesis flows, refer to your tool vendor's documentation, or the appropriate chapter in volume 1 of the *Quartus II Handbook*: *Synopsys Synplify Support* or *Mentor Graphics Precision Synthesis Support*.

Other Synthesis Tools

You can also partition your design and create different netlist files manually with the basic Synplify software (non-Pro/Premier), the basic Precision RTL software (non-Plus), or any other supported synthesis tool by creating a separate project or implementation for each partition, including the top level. Set up each higher-level project to instantiate the lower-level VQM/EDIF netlists as black boxes. Synplify, Precision, and most synthesis tools automatically treat a design block as a black box if the logic definition is missing from the project. Each tool also includes options or attributes to specify that the design block should be treated as a black box, which you can use to avoid warnings about the missing logic.

Design Partition Assignments Compared to Physical Placement Assignments

Design partitions for incremental compilation are logical partitions, different from physical placement assignments in the device floorplan. A logical design partition does not refer to a physical area of the device and does not directly control the placement of instances. A logical design partition sets up a virtual boundary between design hierarchies so each is compiled separately, preventing logical optimizations from occurring between them. When the software compiles the design source code, the logic in each partition can be placed anywhere in the device unless you make additional placement assignments. The software creates a separate post-synthesis and post-fitting netlist for each partition, which allows the software to reuse the synthesis results or reuse the fitting results (including placement and routing information) in subsequent compilations.

If you preserve the compilation results using a Post-Fit netlist, it is not necessary for you to back-annotate or make any location assignments for specific logic nodes. You should not use the incremental compilation and assignment back-annotation features in the same Quartus II project. The incremental compilation feature does not use placement "assignments" to preserve placement results; it simply reuses the netlist database that includes the placement information.

You can assign design partitions to physical regions in the device floorplan using LogicLock assignments. In the Quartus II software, LogicLock regions are used to constrain blocks of a design to a particular region of the device. Altera recommends using LogicLock regions to improve the quality of results and avoid placement conflicts in some cases when performing incremental compilation. Creating floorplan location assignments for design partitions using LogicLock regions is discussed in "Creating a Design Floorplan with LogicLock Location Assignments" on page 2–25.

For more information about when and why to create a design floorplan, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.

Creating Design Partition Assignments

There are several ways to designate a design instance as a design partition, as described in the following subsections. If the full incremental compilation option is not turned on when you specify your first design partition, a dialog box appears that asks whether you want to enable incremental compilation.

To turn on incremental compilation, on the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Process Settings**. Under **Compilation Process Settings**, select **Incremental Compilation**. On the **Incremental Compilation** page, turn on **Full incremental compilation**. Turning off the **Full incremental compilation** option does not remove any partition assignments. Partition assignments have no effect on the design if incremental compilation is turned off.

Creating Design Partitions with the Design Partition Planner

The Design Partition Planner allows you to view design connectivity and hierarchy, and can assist you in creating effective design partitions that follow Altera's guidelines.

To view a design and create design partitions, first compile the design, or perform at least Analysis and Synthesis. On the Tools menu, click **Design Partition Planner**. The design is displayed as a single top-level design block, containing its lower-level instances as boxes.

To show connectivity between blocks, extract instances from the top-level design block. Click on a design block and drag it into the surrounding white space, or right-click an entity and click **Extract from Parent** on the Shortcut menu. When you extract entities, connection bundles are drawn between entities, showing the number of connections existing between pairs of entities. When you have extracted a design block that you want to set as a design partition, right-click on that design block and choose **Create Design Partition**.

The Design Partition Planner also has an **Auto-Partition** feature that creates partitions based on the size and connectivity of the hierarchical design blocks. Right-click on the design block you want to partition (such as the top-level design hierarchy), and choose **Auto-Partition**. You can then analyze and adjust the partition assignments as required.

Figure 2–8 shows the Design Partition Planner after making a design partition assignment to one instance (in the pale red shaded box), and dragging another instance away from the top-level block within the same partition (two design blocks in the pale blue shaded box). The figure shows the number of connections between each partition and information about the size of each design instance.

Figure 2–8. Design Partition Planner

To switch between connectivity display mode and hierarchical display mode, click **Hierarchy Display** on the View menu. Alternately, to switch temporarily to a view-only hierarchy display, click and hold the hierarchy icon in the top-left corner of any entity.

To control the way the connection bundles are displayed, right-click in the white space and choose **Bundle Configuration**. For example, you can remove the connection lines between partitions and I/O banks by turning off **Display connections to I/O banks**. You can also use the settings on the **Connection Counting** tab to adjust how the connections are counted in the bundles.

For more details about how to use the Design Partition Planner, refer to Using the Design Partition Planner in the Quartus II Help.

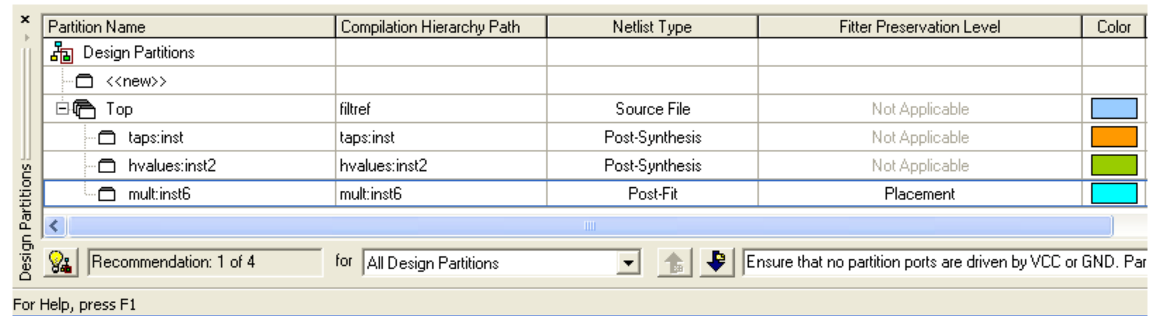
Creating Design Partitions In the Design Partitions Window

The Design Partitions Window allows you to create and delete partitions, and is the main window for setting the Netlist Type and Fitter Preservation Level described in "Setting the Netlist Type for Design Partitions" on page 2–19. First, perform Analysis and Elaboration, or any compilation flow that includes this step. Elaboration is the part of the synthesis process that identifies your design's hierarchy. On the Assignments menu, click **Design Partitions Window** (Figure 2–9). In this window, you can create your partitions in one of the following ways:

- Create new partitions for one or more instances by dragging and dropping them from the **Hierarchy** tab of the **Project Navigator** into the **Design Partitions Window**. Using this method, you can create multiple partitions at once.
- Create new partitions by double-clicking the <<*new*>> cell in the **Partition Name** column. In the **Create New Partitions** dialog box, select the design instance and click **OK**.

To delete partitions in the Design Partitions Window, right-click a partition and click **Delete**, or select the partition and press the Delete key.

Figure 2–9. Design Partitions Window



The Design Partitions Window lists recommendations at the bottom of the window, as well as a link to the Incremental Compilation Advisor, where you can view additional recommendations about the partitions.

The **Color** column indicates the color of each partition in the Partition Planner view of the Chip Planner floorplan. The **Source File Status** column lists the date that the source code was changed and bold font indicates that it has changed since the last compilation. The other status columns indicate which post-compilation netlists are available.

Creating Design Partitions in the Project Navigator

You can use the list of instances under the **Hierarchy** tab in the **Project Navigator** to create and delete design partitions. First, elaborate the design or run any compilation flow that includes this step. Right-click an instance in the **Project Navigator**, point to **Design Partition,** and click **Set as Design Partition**. A design partition icon appears next to each instance that is set as a partition (Figure 2–10).

To remove an existing partition assignment, click **Set as Design Partition** again. (This process turns off the option.)

Figure 2–10. Project Navigator Showing Design Partitions

Creating Design Partitions with Tcl Scripting

You can also create partitions with Tcl scripting commands. For details about the command line and scripting flow, refer to “Scripting Support” on page 2–69.

Partition Name

When you create a partition, the Quartus II software automatically generates a name based on the instance name and hierarchy path. To change the name, double-click on the partition name in the Design Partitions window, or right-click the partition and click **Rename**. Alternatively, right-click the partition in the Design Partitions window and click **Properties** to open the **Design Partition Properties** dialog box. On the **General** tab, enter the new name in the **Name** field.

By renaming your partitions, you can avoid referring to them by their hierarchy path, which can sometimes be long. This is especially important when using command-line commands or assignments. Partition names can be from 1 to 1024 characters in length and must be unique. The name can only contain alphanumeric characters and the pipe (|), colon (:), and underscore (_) characters.

Setting the Netlist Type for Design Partitions

The Netlist Type is a property of each design partition that allows you to specify the type of netlist or source file that the compiler should use as the input for each partition. The Netlist Type property controls the incremental compilation process, as described in “Compiling a Design Using Incremental Compilation” on page 2–9. This property determines which netlist the Partition Merge stage uses in the next compilation.

To view and modify the Netlist Type, on the Assignments menu, click **Design Partitions Window**. Double-click the Netlist Type for an entry. Alternatively, right-click on an entry, click **Design Partition Properties**, then modify the Netlist Type on the **Compilation** tab.

Table 2–2 and Table 2–3 describe the standard and advanced settings for the Netlist Type property, explains the behavior of the Quartus II software for each setting, and provides guidance on when to use each setting. For examples that describe how to use these settings to accomplish various design goals, refer to "Recommended Design Flows and Compilation Application Examples" on page 2–49.

Table 2–2. Standard Netlist Type Settings

Partition Netlist Type	Quartus II Behavior for Partition During Compilation
Source File	Always compiles the partition using the associated design source file(s). Use this netlist type to recompile a partition from the source code using new synthesis or Fitter settings.
Post-Synthesis	Preserves post-synthesis results for the partition and reuses the post-synthesis netlist as long as the following conditions are true: ■ A post-synthesis netlist is available from a previous synthesis ■ No change that triggers an automatic resynthesis has been made to the partition since the previous synthesis. For details, refer to "What Changes Trigger a Partition's Automatic Resynthesis?" on page 2–23. Compiles the partition from the source files if resynthesis is triggered or if a post-synthesis netlist is not available. Use this netlist type to preserve the synthesis results unless you make design changes, but allow the Fitter to refit the partition using any new Fitter settings.
Post-Fit	Preserves post-fit results for the partition and reuses the post-fit netlist as long as the following conditions are true: ■ A post-fit netlist is available from a previous fitting ■ No change that triggers an automatic resynthesis has been made to the partition since the previous fitting. For details, refer to "What Changes Trigger a Partition's Automatic Resynthesis?" on page 2–23. When a post-fit netlist is not available, the software reuses the post-synthesis netlist if it is available, or otherwise compiles from the source files. Compiles the partition from the source files if resynthesis is triggered. The Fitter Preservation Level specifies what level of information is preserved from the post-fit netlist. For details, refer to "Fitter Preservation Level" on page 2–21. Use this netlist type to preserve the Fitter results unless you make design changes. You can also use this netlist type to apply global optimizations, such as Physical Synthesis optimizations that occur in the Fitter, to certain partitions while preserving the fitting results for other partitions.

Table 2–3. Advanced Netlist Type Settings

Partition Netlist Type	Quartus II Behavior for Partition During Compilation
Post-Fit (Strict)	Preserves post-fit results for the partition even if changes have been made to the associated source files since the previous fitting. Misuse of the **Post-Fit (Strict)** Netlist Type can result in the generation of a functionally incorrect netlist when source design files change. Use caution when applying this assignment. For more information, refer to “Forcing Use of the Post-Fitting Netlist When a Partition has Changed” on page 2–25. When a post-fit netlist is not available, the software reuses the post-synthesis netlist if it is available, or otherwise compiles from the source files. The Fitter Preservation Level specifies what level of information is preserved from the post-fit netlist. For details, refer to “Fitter Preservation Level” on page 2–21.
Empty	Uses an empty placeholder netlist for the partition and automatically adds virtual pins at the partition boundaries. You can use this netlist type to skip the compilation of a partition. For more details on the **Empty** setting, refer to “Empty Partitions” on page 2–22.

Fitter Preservation Level

The **Fitter Preservation Level** property specifies which information the compiler uses from a post-fit netlist.

On the Assignments menu, click **Design Partitions Window**. To view and modify the Fitter Preservation Level, double-click an entry. Alternatively, right-click and click **Properties**, then edit the F**itter Preservation Level** on the **Compilation** tab.

Table 2–4 describes the Fitter Preservation Level settings.

Table 2–4. Fitter Preservation Level Settings (Part 1 of 2)

Fitter Preservation Level	Quartus II Behavior for Partition During Compilation
Placement	Preserves the netlist atoms and their placement in the design partition. Re-routes the design partition. This setting saves significant compilation time because the Fitter does not need to re-fit the nodes in the partition.
Placement and Routing	Preserves the design partition’s netlist atoms and their placement and routing. The minimum preservation level required to preserve Engineering Change Order (ECO) changes made to the post-fitting netlist and SignalProbe pins added to the design. This setting reduces compilation times compared to Placement only, but provides less flexibility to the router to make changes if there are changes in other parts of the design.

Table 2–4. Fitter Preservation Level Settings (Part 2 of 2)

Fitter Preservation Level	Quartus II Behavior for Partition During Compilation
Placement, Routing, and High-Speed Tiles	Preserves the design partition's netlist atoms and their placement and routing in the design partition, as well as the high-speed power tile settings. This setting maximizes performance preservation for timing-critical paths, while allowing low-power tiles to be switched to high-speed if required as the rest of the design is changed. This setting is available only for devices with configurable power tiles.
Netlist Only	Preserves the netlist atoms of the design partition, but replaces and reroutes the design partition. A post-fit netlist with the atoms preserved can be different than the Post-Synthesis netlist because it contains Fitter optimizations; for example, Physical Synthesis changes made during a previous Fitting. You can use this setting to: ■ Preserve Fitter optimizations but allow the software to perform placement and routing again ■ Reapply certain Fitter optimizations (such as Fitter, physical synthesis) that would otherwise be impossible when the placement is locked down ■ Resolve resource conflicts between two imported partitions in a bottom-up design flow

Empty Partitions

You can use the **Empty** setting to skip the compilation of a partition that is incomplete or missing from the top-level design. You can also use it if you want to compile only some partitions in the design, such as during optimization or if the compilation time is large for one partition and you want to exclude it. This is useful if you want to optimize the placement of a timing-critical block such as an IP core, and then lock its placement before adding the rest of your custom logic in a top-down design flow.

To set the Netlist Type to **Empty**, on the Assignments menu, click **Design Partitions Window**, double-click an entry, or right-click an entry and click **Design Partition Properties** and select **Empty**. This setting specifies that the Quartus II Compiler should use an empty placeholder netlist for the partition.

When a partition Netlist Type is defined as **Empty**, virtual pins are automatically created at the boundary of the partition. This means that the software temporarily maps I/O pins in the lower-level design entity to internal cells instead of pins during compilation.

Any child partitions below an empty partition in the design hierarchy are also automatically treated as empty, regardless of their settings.

If you plan to take full advantage of the **Empty** setting, it is important to keep the design logic in the leaves of the hierarchy tree to provide the most flexibility. If you have logic at one hierarchy level and additional logic in a child hierarchy, you cannot isolate the top-level logic in an empty partition without the lower-level partition being treated as empty as well.

You can use a design flow in which some partitions are set to **Empty** in a variation of a bottom-up design flow, where you develop pieces of the design separately and then combine them at the top level at a later time.

When you implement part of the design without information about the rest of the project, it is impossible for the Compiler to perform global placement optimizations. To reduce this effect, follow good partitioning guidelines by ensuring the input and output ports of the partitions are registered whenever possible, and minimizing cross-partition I/O.

When you set a design partition to **Empty**, a design file is required in Analysis and Synthesis to specify the port interface information so it can connect the partition correctly to other logic and partitions in the design. If a partition is imported from another project, the Quartus II Exported Partition (**.qxp**) file contains this information. For more information about these files, refer to "Quartus II Exported Partition Files (.qxp)" on page 2–30. If there is no **.qxp** file or design file to represent the design entity, you must create a wrapper file (called a black box, stub, or hollow-body file) that defines the design block and specifies the input, output, and bidirectional ports. For example, in Verilog HDL, you should include a module declaration, and in VHDL, you should include an entity and architecture declaration.

If the project database includes a previously generated post-synthesis or post-fit netlist for an unchanged **Empty** partition, you can set the Netlist Type from **Empty** directly to **Post-Synthesis** or **Post-Fit**. In this case, the software reuses the previous netlist information and does not have to recompile from the source code.

Where Are the Netlist Databases Stored?

The incremental compilation database folder (**\incremental_db**) includes all the netlist information from previous compilations. To avoid unnecessary recompilations, these database files must not be altered or deleted.

If you archive or reproduce the project in another location, you can use a Quartus II Archive File (**.qar)**. Include the compilation database to preserve post-synthesis or post-fit compilation results. For details, refer to "Using Incremental Compilation with Quartus II Archive Files" on page 2–61.

To manually create a project archive that preserves compilation results without keeping the incremental compilation database, you can keep all source and settings files, and create and save a **.qxp** file for each partition in the design that can be imported into the project to import the compilation results. Refer to "Exporting a Lower-Level Block within a Project" on page 2–35 for more details about how to create a **.qxp** file for a partition within your design.

What Changes Trigger a Partition's Automatic Resynthesis?

A partition is synthesized from its source files if there is no post-synthesis netlist available from a previous synthesis, or if the Netlist Type is set to **Source File**. In addition, certain changes to a design partition trigger an automatic resynthesis of the partition when the Netlist Type is **Post-Synthesis** or **Post-Fit**. The software resynthesizes the partition in these cases to ensure that the design description matches the post-place-and-route programming files. If you don't want this resynthesis to occur automatically, set the Netlist Type to **Post-Fit (Strict)**. Refer to "Forcing Use of the Post-Fitting Netlist When a Partition has Changed" on page 2–25.

The following list explains the changes that trigger a partition's automatic resynthesis when the Netlist Type is set to **Post-Synthesis** or **Post-Fit**:

- The device family setting has changed.

- Any dependent source design file has changed. Refer to "Determining Which Partitions Are Resynthesized Due to Source Code Changes" on page 2–24 for details.
- The partition boundary was changed by an addition, removal, or change to the port boundaries of a lower-level partition (that is, a partition defined for a lower-level instance within this partition).
- A dependent source file was compiled into a different library (so it has a different `-library` argument).
- A dependent source file was added or removed; that is, the partition depends on a different set of source files.
- The partition's root instance has a different entity binding. In VHDL, an instance may be bound to a specific entity and architecture. If the target entity or architecture changes, it triggers resynthesis.
- The partition has different parameters on its root hierarchy or on an internal AHDL hierarchy (AHDL automatically inherits parameters from its parent hierarchies). This occurs if you modified the parameters on the hierarchy directly, or if you modified them indirectly by changing the parameters in a parent design hierarchy.

The software reuses the post-synthesis results but re-fits the design if you change the device setting within the same device family. The software reuses the post-fitting netlist if you change only the device speed grade.

Synthesis and Fitter assignments such as optimization settings, timing assignments, or Fitter location assignments including pin assignments, do not trigger automatic recompilation in the incremental compilation flow. For details about how you can affect placement with LogicLock regions, refer to "What LogicLock Changes Trigger Refitting?" on page 2–28. To recompile a partition with new assignments, change the Netlist Type assignment for that partition to one of the following:

- **Source File** to recompile with all new settings
- **Post-Synthesis** to recompile using existing synthesis results but new Fitter settings
- **Post-Fit** with the **Fitter Preservation Level** set to **Placement** to rerun routing using existing placement results, but new routing settings (such as delay chain settings)

Determining Which Partitions Are Resynthesized Due to Source Code Changes

The Quartus II software uses an internal checksum algorithm to determine whether the contents of a source file have changed. Source files are the design files used to create the design, and consist of VHDL files, Verilog HDL files, AHDL files, Block Design Files (**.bdf**), EDIF netlists, VQM netlists, memory initialization files, as well as **.qxp** files from exported partitions. Changes in other files, such as vector waveform files for simulation, do not trigger recompilation. When design files in a partition have dependencies on other files, changing one file may trigger an automatic recompilation of another file. The Partition Dependent Files table in the Analysis and Synthesis report lists the design files that contribute to each design partition. You can use this table to determine which partitions are recompiled when a specific file is changed.

For example, if a design has file **A.v** that contains entity a, **B.v** that contains entity **B**, and **C.v** that contains entity **C**, then the Partition Dependent Files table for the partition containing entity a lists file **A.v**, the table for the partition containing entity **B** lists file **B.v**, and the table for the partition containing entity **C** lists file **C.v**. Any dependencies are transitive, so if file **A.v** depends on **B.v**, and **B.v** depends on **C.v**, the entities in file **A.v** depend on files **B.v** and **C.v**. In this case, files **B.v** and **C.v** are listed in the report table as dependent files for the partition containing entity **A**.

If you define module parameters in a higher-level module, the Quartus II software checks the parameter values when determining which partitions require resynthesis. If you change a parameter in a higher-level module that affects a lower-level module, the lower-level module is resynthesized. Parameter dependencies are tracked separately from source file dependencies; therefore, parameter definitions are not listed in the Partition Dependent Files list.

If a design contains common files, such as an **includes.v** file that is referenced in each entity by the command `` `include includes.v``, all partitions are dependent on this file. A change to **includes.v** causes the entire design to be recompiled. The VHDL statement `use work.all` also typically results in unnecessary recompilations, because it makes all entities in the work library visible in the current entity, which results in the current entity being dependent on all other entities in the design.

To avoid this type of problem, ensure that files common to all entities, such as a common include file, contain only the set of information that is truly common to all entities. Remove `use work.all` statements in your VHDL file or replace them by including only the specific design units needed for each entity.

Forcing Use of the Post-Fitting Netlist When a Partition has Changed

Forcing the use of the post-fitting netlist when the contents of a source file has changed is recommended only for advanced users who understand when a partition must be recompiled. You might use this assignment, for example, if you are making source code changes but do not want to recompile the partition until you finish debugging a different partition. To force the Fitter to use a previously generated post-fit netlist even when there are changes to the source files, you can use the **Post-Fit (Strict)** Netlist Type assignment.

Misuse of the **Post-Fit (Strict)** Netlist Type can result in the generation of a functionally incorrect netlist when source design files change. Use caution when applying this assignment.

Creating a Design Floorplan with LogicLock Location Assignments

A floorplan represents the layout of the physical resources on the device. The expressions “creating a design floorplan” and “floorplanning” describe the process of mapping the logical design hierarchy onto physical regions in the device floorplan. After you have partitioned the design, you can create floorplan location assignments for the design as discussed in this section to improve the quality of results when using the full incremental compilation flow. Creating a design floorplan is not a requirement to use an incremental compilation flow, but it is highly recommended in certain cases. Floorplan location planning can be important for a design that uses incremental compilation for the following reasons:

- To avoid resource conflicts between partitions, predominantly when partitions are imported from another Quartus II project

- To ensure a good quality of results when recompiling individual partitions in top-down flows

In top-down flows, design floorplan assignments prevent the situation in which the Fitter must place a partition in an area of the device where most resources are already used by other partitions. A physical region assignment provides a reasonable region to re-place logic after a change, so the Fitter does not have to scatter logic throughout the available space in the device.

Floorplan assignments are not required for non-critical partitions in a top-down flow. The logic for partitions that are not timing-critical (such as simple top-level glue logic) can be placed anywhere in the device on each recompilation, if that is best for your design.

The simplest way to create a floorplan for a partitioned design is to create one LogicLock region per partition (including the top-level partition). Initially, you can leave each region with the default settings of **Auto** size and **Floating** location to allow the Quartus II software to determine the optimal size and location for the regions. Then, after compilation, use the Fitter-determined size and origin location as a starting point for your design floorplan. Check the quality of results obtained for your floorplan location assignments and make changes to the regions as needed. Alternately, you can perform synthesis, and then set the regions to the required size based on resource estimates. In this case, use your knowledge of the connections between partitions to place the regions in the floorplan.

Once you have created an initial floorplan, you can refine the region using tools in the Quartus II software. You can also use advanced techniques such as creating non-rectangular regions by nesting child LogicLock regions.

For more information about when creating a design floorplan can be important, as well as guidelines for creating the floorplan, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.

You can use the Incremental Compilation Advisor to check that your LogicLock regions meet Altera's guidelines, described in "Incremental Compilation Advisor" on page 2–13.

To create a LogicLock region for each design partition, use the following general methodology:

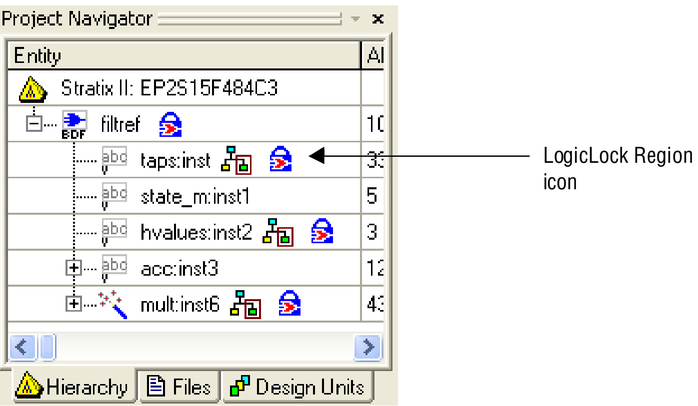
1. On the Assignments menu, click **Design Partitions Window** and ensure that all partitions have their Netlist Type set to **Source File** or **Post-Synthesis**. If the Netlist Type is set to **Post-Fit**, floorplan location assignments are not used when recompiling the design.

2. Create a LogicLock region for each partition (including the top-level entity, which is automatically considered a partition) using one of the following methods:
   - On the Tools menu, click **Design Partition Planner**. Right-click within the colored box that represents a partition and click **Create LogicLock Region**. In the Design Partitions Window, right-click on a partition and click **Create New LogicLock Region**.
   - Under **Compilation Hierarchy** in the **Project Navigator**, right-click each instance that is denoted as a partition and click **Create New LogicLock Region**. In the Design Partitions Window, right click on the row for a partition and choose **Create New LogicLock Region**.

With any of these methods, you can highlight multiple (or all) partitions by holding down the Ctrl key and clicking each partition. Then you can choose the option to create a separate LogicLock region for each highlighted partition.

A LogicLock icon appears in the Project Navigator next to each instance that is set as a LogicLock region (Figure 2–11).

Figure 2–11. Project Navigator Showing LogicLock Regions



3. To place auto-sized, floating-location LogicLock regions, on the Processing menu, point to **Start** and click **Start Early Timing Estimate**.

You must perform Analysis and Synthesis and Partition Merge before performing an Early Timing Estimate.

   To run a full compilation instead of the Early Timing Estimate, on the Processing menu, click **Start Compilation**.

4. On the Assignments menu, click **LogicLock Regions Window**, and while holding the Ctrl key, click each LogicLock region to select all regions (including the top-level region).
5. Right-click the last selected LogicLock region, and click **Set Size and Origin to Previous Fitter Results**.

Use the Fitter-chosen locations only as a starting point to make the regions of a fixed size and location. Generally, regions with fixed size and location yield better f_{MAX} than auto-sized regions.

Do not back-annotate the contents of the region, just save the size and origin. Placement is preserved using the post-fit netlist, not back-annotated content assignments.

6. If required, modify the size and location via the **LogicLock Regions Window** or the **Chip Planner**. For example, make the regions bigger to fill up the device and allow for future logic changes.
7. To estimate the timing performance of your design with these LogicLock regions, on the Processing menu, point to **Start** and click **Start Early Timing Estimate**.
8. Repeat steps 6 and 7 until you are satisfied with the quality of results for your design floorplan.
9. On the Processing menu, click **Start Compilation** to run a full compilation.

If you do not use auto-sized and floating-location regions, you can estimate the size of the regions after synthesis in steps 3–5. On the Processing menu, point to **Start** and click **Start Analysis & Synthesis**. Right-click a region in the **LogicLock Regions** dialog box, and choose **Set to Estimated Size**. Then continue with step 6 to modify the size and origin of each region as appropriate.

Taking Advantage of the Early Timing Estimator

The methodology for creating a floorplan takes advantage of the Early Timing Estimator to enable quick compilations of the design while creating assignments. The Early Timing Estimator feature provides a timing estimate for a design as much as 45 times faster than running a full compilation, yet estimates are, on average, within 11% of final design timing. You can use the Chip Planner to view the "placement estimate" created by this feature, identify critical paths by locating from the timing analyzer reports, and, if necessary, add or modify floorplan constraints. You can then rerun the Early Timing Estimator to quickly assess the impact of any floorplan location assignments or logic changes, enabling rapid iterations on design variants to help you find the best solution. This faster placement has an impact on the quality of results. If getting the best quality of results is important in a given design iteration, perform a full compilation with the Fitter instead of using the Early Timing Estimate feature.

What LogicLock Changes Trigger Refitting?

As described in "What Changes Trigger a Partition's Automatic Resynthesis?" on page 2–23, most assignment changes do not trigger recompilation of a partition if the Netlist Type and Fitter Preservation Level settings specify that Fitter results should be preserved. For example, changing a pin assignment does not trigger recompilation; therefore, the design does not use the new pin assignment unless you change the Netlist Type to **Post-Synthesis** or **Source File**.

Similarly, if a partition's placement is preserved, and the partition is assigned to a LogicLock region, the Fitter always reuses the corresponding LogicLock region size specified in the post-fit netlist. That is, changes to the LogicLock **Size** setting do not trigger refitting if a partition's placement is preserved with the **Post-Fit** Netlist Type setting or with an imported partition that includes post-fit information.

However, you can use the LogicLock **Origin** location assignment to change or fine-tune the previous Fitter results. When you change the **Origin** setting for a region, the Fitter can move the region in the following manner, depending upon how the placement is preserved for that region's members:

- When you set a new region Origin, the Fitter uses the new origin and re-places the logic, preserving the relative placement of the member logic.
- When you set the region Origin to **Floating**, the following conditions apply:
  - If the region's member placement is preserved with an Imported partition: The Fitter chooses a new Origin and re-places the logic, preserving the relative placement of the member logic within the region.
  - If the region's member placement is preserved with a **Post-Fit** Netlist Type: The Fitter does not change the Origin location, and reuses the previous placement results.

Exporting and Importing Partitions

A bottom-up compilation flow refers to the design methodology in which a project is first divided into smaller subdesigns that are implemented as separate projects, potentially by different designers. The compilation results of these lower-level projects are then exported and given to the designer (or the project lead) who is responsible for importing them into the top-level project to obtain a fully functional design. This type of design flow is required only if lower-level designers want to optimize their placement and routing independently, and pass their design to the project lead to reuse placement and routing results. Otherwise, a project lead can integrate source HDL from several designers in a single Quartus II project, and use the standard incremental compilation flow described previously.

In a bottom-up design flow, the top-level project lead can do much of the design planning, and then pass constraints on to the designers of lower-level blocks. The bottom-up design partition scripts generated by the Quartus II software can make it easier to plan a bottom-up design, and limit the difficulties that can arise when integrating separate designs. Refer to "Generating Bottom-Up Design Partition Scripts for Project Management" on page 2–40 for details.

For examples of using team-based and bottom-up design to achieve design goals, refer to "Recommended Design Flows and Compilation Application Examples" on page 2–49. There are some additional restrictions related to bottom-up flows in the Quartus II software, described in "Incremental Compilation Restrictions" on page 2–60.

This section describes the export and import features provided to support bottom-up compilation flows.

The section covers the following topics:

- "Quartus II Exported Partition Files (.qxp)" on page 2–30
- "Bottom-Up Incremental Compilation Summary"
- "Preparing a Design for Bottom-Up Incremental Compilation" on page 2–31
- "Exporting a Lower-Level Partition to be Used in a Top-Level Project" on page 2–34

- "Exporting a Lower-Level Block within a Project" on page 2–35
- "Using a .qxp File as a Source File in the Top-Level Project" on page 2–36
- "Importing a Lower-Level Partition Into the Top-Level Project" on page 2–36
- "Importing Assignments and Advanced Import Settings" on page 2–38
- "Generating Bottom-Up Design Partition Scripts for Project Management" on page 2–40
- "Importing SDC Constraints from Lower-Level Partitions" on page 2–45

Quartus II Exported Partition Files (.qxp)

The bottom-up incremental compilation flow uses a **.qxp** file to represent lower-level design partitions. A **.qxp** file is a binary file that contains compilation results describing the exported design partition and includes a post-fit or post-synthesis netlist, and a set of assignments typically including LogicLock placement constraints. The **.qxp** file does not contain the original source design files from the lower-level design.

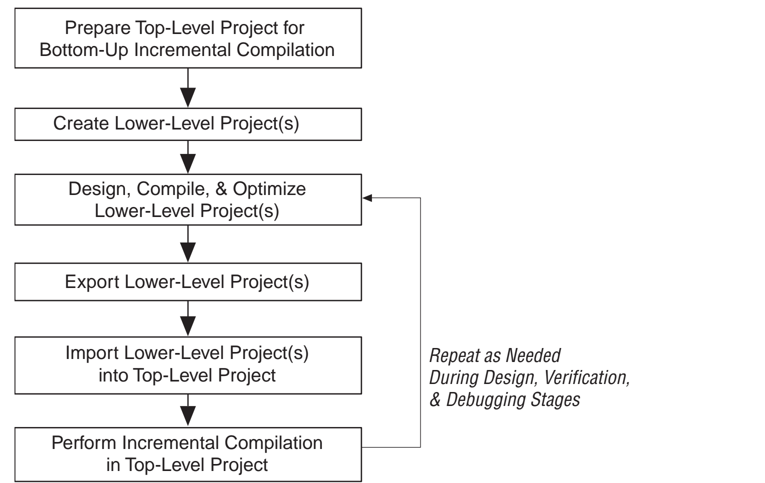
The following sections provide an overview of a bottom-up incremental compilation flow, and describe how to generate a **.qxp** file for a lower-level design partition, and how to import the **.qxp** file into the top-level project.

Bottom-Up Incremental Compilation Summary

The flow chart in Figure 2–12 illustrates the incremental compilation flow using a bottom-up methodology in which lower-level partitions are compiled separately before being imported into the top-level project. The following subsections describe the steps involved in the flow.

First, prepare the top-level design for incremental compilation. Then design, optimize, verify, and debug the lower-level projects. Export the design hierarchy of each lower-level project as a Quartus II **.qxp**, and import the **.qxp** files into the top-level design. Finally, compile the entire top-level design.

Figure 2–12. Summary of Bottom-Up Incremental Compilation Flow



Preparing a Design for Bottom-Up Incremental Compilation

To prepare the design for a bottom-up design methodology, the project lead or top-level designer should perform the following steps:

1. Create the top-level Quartus II project, and apply project-wide settings and global assignments.

   a. Create source code for a "skeleton" of the entire design that includes the hierarchy and the port interfaces for the lower-level designs. The top-level design file instantiates the lower-level blocks you plan to compile in separate Quartus II projects. If you want to compile the design with the lower-level blocks missing, create empty black box wrapper files for each design block to define the design entity and ports.

   b. Create all global assignments, including the device assignment, pin location assignments, and timing assignments, so the final design meets its requirements. Lower-level project designers can add their own constraints for their partitions as needed, and later provide them to the top-level designer, but the project-wide constraints that affect more than one lower-level project constraint should be provided by the top-level designer or project lead to avoid any conflicts and ensure that lower-level projects use the correct assignments.

2. For each lower-level design block imported to the top-level project, designate the instance as a design partition with an Empty Netlist Type. Refer to "Creating Design Partition Assignments" on page 2–16 and "Setting the Netlist Type for Design Partitions" on page 2–19 for details.

3. If the project lead plans to import placement information from the lower-level projects, create LogicLock regions for each of the lower-level partitions to create a design floorplan. Refer to "Creating a Design Floorplan with LogicLock Location Assignments" on page 2–25.

4. Optional: Perform a full compilation of the skeleton design. On the Project menu, click **Generate Bottom-Up Design Partition Scripts**. Provide each lower-level designer with the generated Tcl file to create their project with the appropriate constraints. If you use makefiles in your design environment, provide the makefile for each partition. Refer to "Generating Bottom-Up Design Partition Scripts for Project Management" on page 2–40 for details.

Creating and Compiling Lower-Level Projects

The designer of each lower-level design should create a separate Quartus II project.

If you create the project manually, create a new Quartus II project for the subdesign with all of the required settings. Create with LogicLock region assignments and global assignments (including clock settings) as specified by the project lead, as well as virtual pin assignments for ports which represent connections to core logic instead of external device pins in the top-level module.

If you have a bottom-up design partition script from the top-level designer, source the Tcl script to create the Quartus II project with all the required settings and assignments from the top-level design.

If you use makefiles, use the **make** command and the makefile provided by the project lead to create a Quartus II project with all of the required settings and assignments, and compile the project. Specify the dependencies in the makefile to indicate which source file should be associated with which partition.

Compile and optimize each lower-level design as a separate Quartus II project.

Exporting Lower-Level Projects

When you achieve the design requirements for the lower-level design, export each design as a partition for the top-level design.

If you are not using makefiles, on the Project menu, use the **Export Design Partition** dialog box to export each lower-level design. Refer to "Exporting a Lower-Level Partition to be Used in a Top-Level Project" on page 2–34. If you want to export only a portion of the design in the lower-level project, refer to "Exporting a Lower-Level Block within a Project" on page 2–35 for instructions. Each lower-level designer must provide the **.qxp** file to the project lead.

If your design team uses makefiles, the project lead can use the **make** command with the **master_makefile** to export the lower-level partitions and create **.qxp** files, and then import them into the top-level design.

Including or Importing Lower-Level Projects into the Top-Level Project

After exporting lower-level projects, the project lead then incorporates the **.qxp** files sent in by the designers of each lower-level subdesign partition.

If you want to use the exported **.qxp** file information as a design file in the top-level design, simply add the **.qxp** file as a source file in the project. In this case, the instance in the **.qxp** file does not have to be a partition in the top-level design. Refer to "Using a .qxp File as a Source File in the Top-Level Project" on page 2–36 for details.

If you want to import any placement information, on the Project menu, click **Import Design Partition** and specify the partition in the top-level project that is represented by the subdesign **.qxp** file. Refer to "Importing a Lower-Level Partition Into the Top-Level Project" on page 2–36 for details. Repeat the process for each partition in the design that you want to import.

You can automate the import process by using makefiles: the **master_makefile** command imports each partition into the top-level design. Be sure to specify which source files should be associated with which partition so that the software can rebuild the project if source files change.

For details about which assignments are imported and how to avoid conflicts, refer to "Importing Assignments and Advanced Import Settings" on page 2–38.

Performing an Incremental Compilation in the Top-Level Project

After you have imported the design partitions that make up the top-level project, you can perform a full compilation. The software compiles imported partitions in the same way as partitions defined in the top-level project. The software recompiles an imported partition only if it has been imported since the last compilation.

Netlist Types for Imported Partitions

Partitions that are imported from another project use two additional netlist types and the top-level project uses the **Empty** Netlist Type to create a placeholder for partitions until the associated **.qxp** files from other designers have been imported. These Netlist Types are described in Table 2–5.

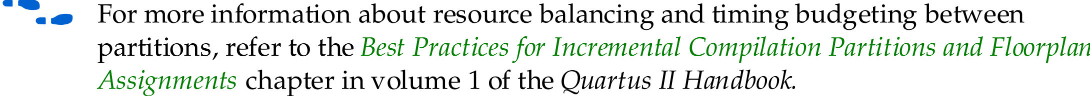
Table 2–5. Netlist Types for Imported Partitions

Partition Netlist Type	Quartus II Behavior for Partition During Compilation
Imported	Compiles the partition using a netlist imported from a **.qxp** file. The software does not modify or overwrite the original imported netlist during compilation. To preserve changes made to the imported netlist (such as movement of an imported LogicLock region), use the **Post-Fit (Import-based)** setting following a successful compilation with the imported netlist. For additional details, refer to "Exporting and Importing Partitions" on page 2–29. To remove the imported netlist and recompile from the source code, set the Netlist Type to **Source File**. The Fitter Preservation Level specifies what level of information is preserved from the imported netlist. Refer to "Fitter Preservation Level" on page 2–21 for details. If you have not imported a netlist for this partition using the **Import Design Partition** command, this setting is not available.
Post-Fit (Import-based)	Preserves post-fit results for the partition and reuses the post-fit netlist as long as the following conditions are true: ■ A post-fit netlist is available from a previous fitting ■ No change has been made to the associated imported netlist since the previous fitting Compiles the partition from the imported netlist if the imported netlist changes (which means it has been reimported) or if a post-fit netlist is not available. Changes to assignments do not cause recompilation. The Fitter Preservation Level specifies what level of information is preserved from the post-fit netlist. Refer to "Fitter Preservation Level" on page 2–21 for details. You can use this netlist type to preserve changes to the placement and routing of an imported netlist. If you have not imported a netlist for this partition using the **Import Design Partition** command, this setting is not available.
Empty	Uses an empty placeholder netlist for the partition and automatically adds virtual pins at the partition boundaries. You can use this netlist type to skip the compilation of a lower-level partition to be imported later. For more details on the Empty setting, refer to "Empty Partitions" on page 2–22.

Exporting a Lower-Level Partition to be Used in a Top-Level Project

Each lower-level subdesign is compiled as a separate Quartus II project. In each project, use the following guidelines to improve the exporting and importing process:

- If you have a bottom-up design partition script from the top level, source the Tcl script to create the project and all the assignments from the top-level design. Doing so may create many of the assignments described below. Ensure that the LogicLock region uses only the resources allocated by the top-level project lead.
- Ensure that you know which clocks should be allocated to global routing resources so that there are no resource conflicts in the top-level design.
  - Set the **Global Signal** assignment to **On** for the high fan-out signals that should be routed on global routing lines.
  - To avoid other signals being placed on global routing lines, on the Assignments menu, click **Settings** and turn off **Auto Global Clock and Auto Global Register Controls** under **More Settings** on the Fitter page of the **Settings** dialog box.
  - Alternatively, you can set the **Global Signal** assignment to **Off** for signals that should not be placed on global routing lines. Placement for LABs depends on whether the inputs to the logic cells within the LAB use a global clock. You may encounter problems if signals do not use global lines in the lower-level design, but use global routing in the top level.
- Use the Virtual Pin assignment to indicate pins of a subdesign that do not drive pins in the top-level design. This is critical when a subdesign has more output ports than the number of pins available in the target device. Using virtual pins also helps optimize cross-partition paths for a complete design by enabling you to provide more information about the subdesign ports, such as location and timing assignments.
- Because subdesigns are compiled independently without any information about each other, you should provide more information about the timing paths that may be affected by other partitions in the top-level design. You can apply location assignments for each pin to indicate the port location after incorporation in the top-level design. You can also apply timing assignments to the I/O ports of the subdesign to perform timing budgeting.



For more information about resource balancing and timing budgeting between partitions, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook.*

When your subdesign partition has been compiled using these guidelines, and is ready to be incorporated into the top-level design, export a subdesign as a partition using the following steps:

1. In the subdesign project, use one of the following methods to open the **Export Design Partition** dialog box.
   - In the Design Partition Planner (available from the Tools menu), right-click within the colored box that represents a partition and click **Export Design Partition**.
   - On the Project menu, click **Export Design Partition**.

2. In the **Export file** box, type the name of the **.qxp** file. By default, the directory path and file name are the same as the current project.
3. You can also select the **Partition hierarchy to export**. By default, the Top partition (the entire project) is exported, but you can choose to export the compilation result of any partition hierarchy in the project, as described in "Exporting a Lower-Level Block within a Project" on page 2–35. Choose the partition hierarchy from the pull-down list.
4. Under **Netlist to export**, select either **Post-fit netlist** or **Post-synthesis netlist**. The default is **Post-fit netlist**. For post-fit netlists, turn on or off the **Export routing** option as required.
5. Click **OK**. The Quartus II software creates the **.qxp** file in the specified directory.

Alternatively, you can set up your project so that the export process is performed every time you compile the design:

1. On the Assignments menu, click **Settings**.
2. In the **Settings** dialog box, under **Compilation Process Settings**, select the **Incremental Compilation** page.
3. Turn on **Automatically export design partition after compilation**.
4. If you want to view or change the default export settings, click the **Export Design Partition Settings** button.
5. In the **Export Design Partition Settings** dialog box, change the settings, if required, as in steps 2-4 in the preceding export procedure. Click **OK**.
6. Click **OK** to close the **Settings** dialog box. During the next full compilation, the software creates the **.qxp** file in the specified directory.

Exporting a Lower-Level Block within a Project

Step 3 in "Exporting a Lower-Level Partition to be Used in a Top-Level Project" on page 2–34 enables you to create a **.qxp** file for a lower-level block within a Quartus II project. When you do this, the command exports the entire hierarchy under the specified partition into the **.qxp** file.

You can use this feature to add test logic around a lower-level block to be exported as a design partition for a top-level design. You can also instantiate additional design components in a lower-level project so it matches the top-level design environment. For example, you can include a top-level PLL in your lower-level project so that you can optimize the design with information about the frequency multipliers, phase shifts, compensation delays, and any other PLL parameters. The software then captures timing and resource requirements more accurately while ensuring that the timing analysis in the lower-level project is complete and accurate. You can export the lower-level partition, without exporting any auxiliary components to the top-level design.

In addition, you can use this feature in a top-down design flow to create **.qxp** files for specific design partitions that are complete. You can then import the **.qxp** file back into the project and use the **Imported** netlist type, as described in the following section. In this usage, the **.qxp** file acts as an archive for the partition, including the netlist and placement and routing information in one file. If you change the source code for the partition, you must change the netlist type back to **Source File** to use the source instead of the imported information.

Using a .qxp File as a Source File in the Top-Level Project

To include the design netlist from a **.qxp** file, you can simply use the **.qxp** file as a source file in your design (just like a Verilog or VHDL source file).

The **.qxp** file contains the design block exported from the subdesign project and has the same name as the partition. When you instantiate the design block in a top-level design and include the **.qxp** file as a source file in the project, the software adds the exported netlist to the database for the top-level project. The **.qxp** port names are case sensitive if the original HDL of the lower-level partition were case sensitive.

The software also filters the assignments from the subdesign to bring the appropriate assignments into the top-level project. Refer to the sections in "Importing Assignments and Advanced Import Settings" on page 2–38 that describe which assignments are included in the top-level project. The assignments in the **.qxp** file are treated like assignments made in an HDL source file, and can be overridden by assignments in the top-level project.

When you use a **.qxp** file as a source file in this way, you cannot currently import any post-fit database information into your project. If you want to import post-fit information from the exported netlist, refer to the following section, "Importing a Lower-Level Partition Into the Top-Level Project".

When you use a **.qxp** file as a source file, you can choose whether you want the file to be a partition in the top-level project. If you do not designate the **.qxp** instance as a partition, the software removes unconnected ports and unused logic just like a regular source file. If you do assign the instance as a design partition, the partition boundary is always preserved, as discussed in "Impact of Design Partitions on Design Optimization" on page 2–11.

If you use the **Locate** command into the **.qxp** file or try to open the **.qxp** file in the Quartus II software, the software opens an exported partition information file that explains the contents of the file and provides the port list. Because the **.qxp** file is a binary file, you cannot view the design netlist itself.

Importing a Lower-Level Partition Into the Top-Level Project

The import process imports the design netlist from the **.qxp** file into a particular design partition and adds the netlist to the database for the top-level project. Importing allows you to re-use the post-fitting results from the exported partition. Importing also filters the assignments from the subdesign to create the appropriate assignments in the top-level project. Before you can import a partition, you must have performed an elaboration of the design hierarchy and assigned the design partitions. If you elaborate the design with Empty partitions and have not created a black box wrapper file to define the port connections, Analysis and Elaboration generates error messages about undefined ports but you can still proceed with the import process.

To import a subdesign partition into a top-level design, perform the following steps:

1. In the top-level project, use one of the following methods to open the **Import Design Partition** dialog box:
   - In the Design Partition Planner right-click within the colored box that represents a partition and click **Import Design Partition**.
   - In the Design Partitions window, right-click on the partition that you want to import and click **Import Design Partition**.
   - On the Project menu, click **Import Design Partition**.
2. In the Partition(s) box, browse to the desired partition if required. To choose a partition, highlight the partition name in the **Select Partition(s)** dialog box and use the appropriate buttons to select or deselect the desired partitions.

   ☞ You can select multiple partitions if your top-level design has multiple instances of the subdesign partition and you want to use the same imported netlist.

3. Under **Import file**, type the name of the **.qxp** file or browse for the file that you want to import into the selected partition. This file is required only during importation, and is not used during subsequent compilations unless you reimport the partition.

   ☞ If you have already imported the **.qxp** file for this partition at least once, you can use the same location as the previous import instead of specifying the file name again. To do so, turn on **Reimport using the latest import files at previous locations**. This option is especially useful when you import the new **.qxp** files for several partitions that you have already imported at least once. You can select all the partitions to be imported in the Partition(s) box and then use the **Reimport using latest import files at previous locations** option to import all partitions using their previous locations, without specifying individual file names.

4. Optional: To view the contents of the selected **.qxp** file, click **Load Properties**. The properties displayed include the Netlist Type, Entity name, Device, and statistics about the partition size and ports.
5. Optional: Click **Advanced Import Settings** and make selections, as appropriate, to control how assignments and regions are integrated from a subdesign into a top-level design partition. During importation, some regions may be resized or slightly moved. Click **OK** to apply the settings.

   For more information about the advanced settings, refer to "Importing Assignments and Advanced Import Settings" on page 2–38.

6. To start importation, in the **Import Design Partition** dialog box, click **OK**. The specified **.qxp** file is imported into the database for the current top-level project.

Importing Assignments and Advanced Import Settings

When you import a subdesign partition into a top-level design, the software sets certain assignments by default and also imports relevant assignments from the subdesign into the top-level design.

Design Partition Properties after Importing

When you import a subdesign partition, the import process sets the partition's Netlist Type to **Imported**.

If you compile the design and make changes to the place-and-route results, use the **Post-Fit (Import-based)** Netlist Type on the subsequent compilation. To discard an imported netlist and recompile from source code, compile the partition with netlist type set to **Source File** and be sure to include the relevant source code with the top-level project.

The import process sets the partition's Fitter Preservation Level to the setting with the highest degree of preservation supported by the imported netlist. For example, if a post-fit netlist is imported with placement information, the level is set to **Placement**, but you can change it to the **Netlist Only** value.

Refer to "Setting the Netlist Type for Design Partitions" on page 2–19 for details about the Netlist Type and Fitter Preservation Level setting.

Importing Design Partition Assignments Within the Subdesign

Design partition assignments defined within the subdesign project are not imported into the top-level project. All logic in the subdesign is imported as one partition in the **.qxp** file.

Synopsys Design Constraint Files for the Quartus II TimeQuest Timing Analyzer

Timing assignments made for the Quartus II TimeQuest Timing Analyzer in a Synopsys Design Constraint (**.sdc**) file are not imported into the top-level project. Ensure that the top-level project includes all of the timing requirements for the entire project.

Refer to "Importing SDC Constraints from Lower-Level Partitions" on page 2–45 for recommendations about managing the SDC constraints for the top-level and lower-level projects.

Importing LogicLock Assignments

LogicLock regions are set to a fixed size when imported. If you instantiate multiple instances of a subdesign in the top-level design, the imported LogicLock regions are set to a Floating location. Otherwise, they are set to a Fixed location. You can change the location of LogicLock regions after they are imported, or change them to a Floating location to allow the software to place each region but keep the relative locations of nodes within the region wherever possible. To preserve changes made to a partition after compilation, use the Netlist Type **Post-Fit (Import-Based)**.

The LogicLock Member State assignment is set to **Locked** to signify that it is a preserved region.

LogicLock back-annotation and node location data is not imported because the **.qxp** file contains all of the relevant placement information. Altera strongly recommends that you do not add to or delete members from an imported LogicLock region.

Importing Other Instance Assignments

All instance assignments are imported, with the exception of design partition assignments, SDC constraints, and LogicLock assignments, as described previously.

Importing Global Assignments

Global assignments are not imported. The project lead should make global assignments in the top-level design. Note that clock settings for the Quartus II Classic Timing Analyzer are global assignments, and are not imported. When it is possible for a particular constraint, the global assignment is converted to an instance-specific assignment for the target design partition.

Advanced Import Settings

The **Advanced Import Settings** dialog box allows you to specify the options that control how assignments and regions are integrated and how to resolve assignment conflicts when importing a subdesign partition into a top-level design. The following subsections describe each of these options.

Allow Creation of New Assignments

Allows the import command to add new assignments from the imported project to the top-level project.

When this option is turned off, it imports updates to existing assignments, but no new assignments are allowed.

Promote Assignments to all Instances of the Imported Entity

Converts and promotes entity-level assignments from the subdesign into instance-level assignments in the top-level design.

Assignment Conflict Resolution: LogicLock Regions

Choose one of the following options to determine how to handle conflicting LogicLock assignments (that is, subdesign assignments that do not match the top-level assignments):

- **Always replace regions in the current project** (default)—Deletes existing regions and replaces them with the new subdesign region. Any changes made to the LogicLock region after the assignments were imported are also deleted.
- **Always update regions in the current projects—**Overwrites existing region assignments to reflect any new subdesign assignments with the exception of the LogicLock Origin, in case the project lead has made floorplan location assignments in the top-level design.
- **Skip conflicting regions—**Ignores and does not import subdesign assignments that conflict with any assignments that exist in the top-level design.

Assignment Conflict Resolution: Other Assignments

Choose one of the following options to determine how to handle conflicts with other types of assignments (that is, the subdesign assignments do not match the top-level assignments):

- **Always replace assignments in the current project** (default)—Overwrites or updates existing instance assignments with the new subdesign assignments.
- **Skip conflicting assignments**—Ignores and does not import subdesign assignments that conflict with any assignments that exist in the top-level design.

Generating Bottom-Up Design Partition Scripts for Project Management

The bottom-up design partition scripts automate the process of transferring top-level project information to lower-level design blocks. The Quartus II software provides an interface for managing resources and timing budgets in the top-level design. This makes it easier for designers of lower-level modules to implement the instructions from the project lead, and avoid conflicts between projects when importing and incorporating the projects into the top-level design. This helps reduce the need to further optimize the designs after integration, and improves overall designer productivity and team collaboration.

Generating bottom-up design partition scripts is optional.

For example design scenarios using these scripts, refer to "Implementing a Team-Based Bottom-Up Design Flow" on page 2–53. In a typical bottom-up design flow, the project lead must perform some or all of the following tasks to ensure successful integration of the subprojects:

- Manually determine which assignments should be propagated from the top level to the bottom levels. This requires detailed knowledge of which Quartus II assignments are required to set up low-level projects.
- Manually communicate the top-level assignments to the low-level projects. This requires detailed knowledge of Tcl or other scripting languages to efficiently communicate project constraints.
- Manually determine appropriate timing and location assignments that help overcome the limitations of bottom-up design. This requires examination of the logic in the lower levels to determine appropriate timing constraints.
- Perform final timing closure and resource conflict avoidance at the top level. Because the low-level projects have no information about each other, meeting constraints at the lower levels does not guarantee they are met when integrated at the top-level. It then becomes the project lead's responsibility to resolve the issues, even though information about the low-level implementation may not be available.

Using the Quartus II software to generate bottom-up design partition scripts from the top level of the design makes these tasks much easier and eliminates the chance of error when communicating between the project lead and lower-level designers. Partition scripts pass on assignments made in the top-level design, and create some new assignments that guide the placement and help the lower-level designers see how their design connects to other partitions. If necessary, you can exclude specific design partitions.

 © March 2009 Altera Corporation

Generate design partition scripts after a successful compilation of the top-level design. On the Project menu, click **Generate Bottom-Up Design Partition Scripts**. The design can have empty partitions as placeholders for lower-level blocks, and you can perform an Early Timing Estimation instead of a full compilation to reduce compilation time.

The following subsections describe the information that can be included in the bottom-up design partition Tcl scripts. Use the options in the **Generate Bottom-Up Design Partition Scripts** dialog box to choose which types of assignments you want to pass down and create in the lower-level partition projects. Each time you rerun the script generation process, the Quartus II software recreates the files and replaces older versions.

For information about current limitations in the bottom-up partition scripts, refer to "Register Packing and Partition Boundaries" on page 2–66.

Project Creation

You can use the **Create lower-level project if one does not exist** option for the partition scripts to create lower-level projects if they are required. The Quartus II Project File for each lower-level project has the same name as the entity name of its corresponding design partition.

With this project creation feature, the scripts work by themselves to create a new project, or can be sourced to make assignments in an existing project.

Excluded Partitions

Use the **Excluded partition(s)** option at the bottom of the dialog box to exclude specific partitions from the Tcl script generation process. Use the browse button, then highlight the partition name in the **Select Partition(s)** dialog box and use the appropriate buttons to select or deselect the desired partitions.

Assignments from the Top-Level Design

By default, any assignments made at the top level (not including default assignments or project information assignments) are passed down to the appropriate lower-level projects in the scripts. The software uses the assignment variables and determines the logical partition(s) to which the assignment pertains. This includes global assignments, instance assignments, and entity-level assignments. The software then changes the assignments so that they are syntactically valid in a project with its target partition's logic as the top-level entity.

The names of the design files that apply to the specific partition are added to each lower-level project.

The script uses the file name(s) specified in the top-level project. If the top-level project used a placeholder wrapper file with a different name than the design file in the lower-level project, be sure to add the appropriate file to the lower-level project.

The scripts process wildcard assignments correctly, provided there is only one wildcard. Assignments with more than one wildcard are ignored and warning messages are issued.

Use the following options to specify which types of assignments to pass down to the lower-level projects:

- **Timing assignments**—When this option is turned on, all Classic Timing Analyzer global timing assignments for the lower-level projects are included in the script, including t_{CO}, t_{SU}, and f_{MAX} constraints. In addition, after you've compiled a design using TimeQuest constraints, a separate Tcl script is generated to create an **.sdc** file for each lower-level project that provides the clock constraints and any minimum or maximum delays. This option may also include timing constraints on internal partition connections.
- **Design partition assignments**—When this option is turned on, script assignments related to design partitions in the lower-level projects are included, as well as assignments associated with LogicLock regions.
- **Pin location assignments**—When this option is turned on, all pin location assignments for lower-level project ports that connect to pins in the top-level design are included in the script, controlling the overuse of I/Os at the top-level during the integration phase and preserving placement.

Virtual Pin Assignments

When **Create virtual pins at low-level ports connected to other design units** is turned on, the Quartus II software searches partition netlists and identifies all ports that have cross-partition dependencies. For each lower-level project pin associated with an internal port in another partition or in the top-level project, the script generates a virtual pin assignment, ensuring more accurate placement, because virtual pins are not directly connected to I/O ports in the top-level project. These pins are removed from a lower-level netlist when it is imported into the top-level design.

Virtual Pin Timing and Location Assignments

One of the main issues in bottom-up design methodologies is that each individual design block includes no information about how it is connected to other design blocks. If you turn on the option to write virtual pin assignments, you can also turn on options to constrain these virtual pins to achieve better timing performance after the lower-level partitions are integrated at the top level.

When **Place created virtual pins at location of top-level source/sink** is turned on, the script includes location constraints for each virtual pin created. Virtual output pins are assigned to the location of the connection's destination in the top-level project, and virtual input pins are assigned to the location of the connection's source in the top-level project. If the top-level design uses Empty partitions, the final location of the connection is not known, but the pin is still assigned to the LogicLock region that contains its source or destination.

As a result, these virtual pins are no longer placed inside the LogicLock region of the lower-level project, but at their location in the top-level design, eliminating resource consumption in the lower-level project and providing more information about lower-level projects and their port dependencies. These location constraints are not imported into the top-level project.

When **Add maximum delay to created virtual input pins**, **Add maximum delay from created virtual output pins**, or both are turned on, the script includes timing constraints for each virtual pin created. The value you enter in the dialog box is the maximum delay allowed to or from all paths between virtual pins to help meet the timing requirements for the complete design. The software uses the `INPUT_MAX_DELAY` assignment or `OUTPUT_MAX_DELAY` assignment to apply the constraint.

This option allows the project lead to specify a general timing budget for all lower-level internal pin connections. The lower-level designer can override these constraints by applying individual node-level assignments on any specific pin as needed.

LogicLock Region Assignments

When **Copy LogicLock region assignments from top-level** is turned on, the script includes assignments identifying the LogicLock assignment for the partition.

The script can also pass assignments to create the LogicLock regions for all other partitions. When **Include all LogicLock regions in lower-level projects** is turned on, the script for each partition includes all LogicLock region assignments for the top-level project and each lower-level partition, revealing the floorplan for the complete design in each partition. Regions that do not belong to other partitions contain virtual pins representing the source and destination ports for cross-partition connections. This allows each designer to view the connectivity between their partition and other partitions in the top-level design more easily, and helps ensure that resource conflicts at the top level are minimized.

When **Remove existing LogicLock regions from lower-level projects** is turned on, the script includes commands to remove LogicLock regions defined in the lower-level project prior to running the script. This ensures that LogicLock regions not part of the top-level project do not become part of the complete design, and avoids any location conflicts by ensuring lower-level designs use the LogicLock regions specified at the top level.

Global Signal Promotion Assignments

To help prevent conflicts in global signal usage when importing projects into the top-level design, you can choose to write assignments that control how signals are promoted to global routing resources in the lower-level partitions. These options can help resource balancing of global routing resources.

When **Promote top-level global signals in lower-level projects** is turned on, the Quartus II software searches partition netlists and identifies global resources, including clock signals. For the relevant partitions, the script then includes a global signal promotion assignment, providing information to the lower-level projects about global resource allocation.

When **Disable automatic global promotion in lower-level projects** is turned on, the script includes assignments that turn off all automatic global promotion settings in the lower-level projects. These settings include the **Auto Global Memory Control Signals** logic option, output enable logic options, and clock and register control promotions. If you select the **Disable automatic global promotion in lower-level projects** option in conjunction with the **Promote top-level global signals in lower-level projects** option, you can ensure that only signals promoted to global resources in the top-level are promoted in the lower-level projects.

Makefile Generation

Makefiles allow you to use **make** commands to ensure that a bottom-up project is up-to-date if you have a make utility installed on your computer. The **Generate makefiles to maintain lower-level and top-level projects option** creates a makefile for each design partition in the top-level design, as well as a master makefile that can run the lower-level project makefiles. The Quartus II software places the master makefiles in the top-level directory, and the partition makefiles in their corresponding lower-level project directories.

You must specify the dependencies in the makefiles to indicate which source file should be associated with which partition. The makefiles use the directory locations generated using the **Create lower-level project if one does not exist** option. If you created your lower-level projects without using this option, you must modify the variables at the top of the makefile to specify the directory location for each lower-level project.

To run the makefiles, use a command such as `make -f master_makefile.mak` from the script output directory. The master makefile first runs each lower-level makefile, which sources its Tcl script and then generates a **.qxp** file to export the project as a design partition. Next, run the top-level makefile that specifies these newly generated **.qxp** files as the import files for their respective partitions in the top-level project. The top-level makefile then imports the lower-level results and performs a full compilation, producing a final design.

To exclude a certain partition from being compiled, edit the `EXCLUDE_FLAGS` section of **master_makefile.mak** according to the instructions in the file, and specify the appropriate options. You can also exclude some partitions from being built, exported, or imported using **make** commands. To exclude a partition, run the makefile using a command such as the one for the GNU **make** utility shown in the following example:

```
gnumake -f master_makefile.mak exclude_<partition directory>=1 ↵
```

This command instructs that the partition whose output files are in <*partition directory*> are not built. Multiple directories can be excluded by adding multiple `exclude_`<*partition directory*> commands. Command-line options override any options in the makefile.

Another feature of makefiles is the ability to have the master makefile invoke the low-level makefiles in parallel on systems with multiple processors. This option can help designers working with multiple CPUs greatly improve their compilation time. For the GNU make utility, add the `-j`<*N*> flag to the **make** command. The value <*N*> is the number of processors that can be used to run the build.

The makefile does not include a make clean option, so the design may recompile when **make** is run again and a **.qxp** file already exists.

 © March 2009 Altera Corporation

Importing SDC Constraints from Lower-Level Partitions

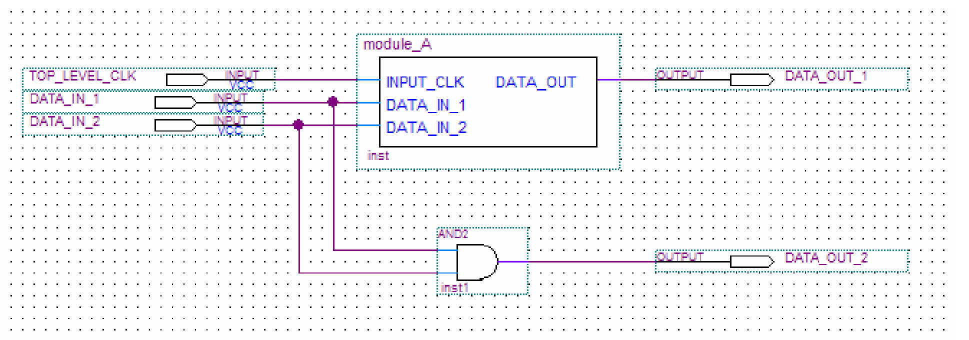
The bottom-up design partition scripts described in the previous section automate the process of transferring top-level project information and constraints to lower-level projects, so that lower-level designers each have a consistent view of the constraints that apply to the entire design. The lower-level partition designers might have to transfer new constraints back to the project lead, so that these constraints are included in final timing sign-off of the entire design. The **Import** command can be used to import assignments from lower-level partition projects into the top-level project; however, the automatic import does not include SDC format constraints for the TimeQuest Timing Analyzer.

Passing additional timing constraints to the top-level project must be managed carefully. This section provides recommendations for managing the timing constraints in a bottom-up incremental compilation flow.

To ensure that there are no conflicts between the project lead's top-level constraints and those added by the lower-level designer, use two Synopsys Design Constraint (**.sdc**) files for each lower-level project: an **.sdc** file created by the project lead that includes project-wide constraints and an **.sdc** file created by the lower-level partition designer that includes partition-specific constraints.

This section uses the example design shown in Figure 2–13 to illustrate these recommendations. The top-level design instantiates a lower-level design block called `module_A` that is set as a design partition and developed by another designer in a separate Quartus II project.

Figure 2–13. Example Design to Illustrate SDC Constraints



In this top-level project, there is a single clock setting called `clk` associated with the FPGA input called `top_level_clk`. The top-level **.sdc** file contains the following constraint for the clock:

```
create_clock -name {clk} -period 3.000 -waveform { 0.000 1.500 }
[get_ports {TOP_LEVEL_CLK}]
```

Creating an .sdc File with Project-Wide Constraints

The **.sdc** file with project-wide constraints for the lower-level project should contain all constraints that are not completely localized to the lower-level partition. This file should be maintained by the top-level project lead. The project lead must ensure that these timing constraints are delivered to the individual partition owners and that they are syntactically correct for each of the lower-level projects. This can be challenging when the design is in flux and hierarchies change. The project lead can use the Generate Bottom-Up Design Partition Scripts tool to automatically generate some of these constraints, as described in the previous section.

The **.sdc** file with project-wide constraints is used in the lower-level project, but is not exported back to the top-level project lead. The lower-level partition designer should not modify this file. If changes are necessary, they should be communicated to the top-level project lead, who can then update the SDC constraints and distribute new files to all lower-level partition designers as required.

The **.sdc** file should include clock creation and clock constraints for any clock used by more than one lower-level project. This is particularly important when dealing with complex clocking structures, such as the following:

- Cascaded clock multiplexers
- Cascaded PLLs
- Multiple independent clocks on the same clock pin
- Redundant clocking structures required for secure applications
- Virtual clocks and generated clocks which are consistently used for source synchronous interfaces
- Clock uncertainties

In addition, the **.sdc** file with project-wide constraints should contain all project-wide timing exception assignments, such as the following:

- Multicycle assignments, `set_multicycle_path`
- False path assignments, `set_false_path`
- Maximum delay assignments, `set_max_delay`
- Minimum delay assignments, `set_min_delay`

The project-wide **.sdc** file can also contain any `set_input_delay` or `set_output_delay` constraints on a lower-level project's ports, because these represent delays external to a given partition. If a lower-level designer wants to set these constraints within the lower-level project, the team must ensure that the I/O port names are identical in the two projects so the assignments can be imported successfully without any changes.

Similarly, a constraint on a path that crosses a partition boundary should be in the project-wide **.sdc** file, because it is not completely localized in a single lower-level project.

Example Step 1: Project Lead Produces .sdc File with Project-Wide Constraints for Lower-Level Project

The device input `top_level_clk` in Figure 2–13 drives the `input_clk` port of `module_A`. To make sure the clock constraint is passed correctly to the lower-level project, the project lead creates an **.sdc** file with project-wide constraints for `module_A` that contains the following command:

```
create_clock -name {clk} -period 3.000 -waveform { 0.000 1.500 }
[get_ports {INPUT_CLK}]
```

The designer of `module_A` includes this **.sdc** file as part of the lower-level project.

Creating an .sdc File with Partition-Specific Constraints

The **.sdc** file with partition-specific constraints should contain all constraints that affect only the lower-level partition. For instance, a `set_false_path` or `set_multicycle_path` constraint for a path entirely within the lower-level partition should be in the partition-specific **.sdc** file. These constraints are required for correct compilation of the partition, but need not be present in any other lower-level projects.

The partition-specific **.sdc** file should be maintained by the individual partition designer; it is their responsibility to add any constraints required to properly compile and analyze their partition.

The partition-specific **.sdc** file is used in the lower-level project and must be exported back to the project lead for the top-level design. The project lead must use the partition-specific constraints to properly constrain the placement, routing, or both if the partition logic is fit at the top level, and to ensure that final timing sign-off is accurate. Use the following guidelines in the partition-specific **.sdc** file to simplify this export/import step:

- Create a hierarchy variable for this partition (such as `module_A_hierarchy`) and set it to an empty string because the partition is the top-level instance in the separate project. The project lead modifies this variable for the top-level hierarchy, reducing the effort of translating constraints on lower-level design hierarchies into constraints that apply in the top-level hierarchy. Use the following Tcl command first to check if the variable is already defined in the project, so that the top-level project does not use this empty hierarchy path: `if {![info exists module_A_hierarchy]}`.
- Use the hierarchy variable in the partition-specific **.sdc** file as a prefix for assignments in the project. For example, instead of naming a particular instance of a register `reg:inst`, use `${module_A_hierarchy}reg:inst`. Also use the hierarchy variable as a prefix to any wildcard characters (such as '`*`').
- Be careful with assignments to I/O ports of the partition. Generally, these assignments should be specified in the **.sdc** file with project-wide constraints because the partition's interface depends on the top-level design. If you want to set I/O constraints within the lower-level project, the team must ensure that the I/O port names are identical in the two projects so the assignments can be imported successfully without any changes.

- Be careful with the `derive_clocks` and `derive_pll_clocks` commands. In most cases, the **.sdc** file with project-wide constraints should call these commands. Because these commands impact the entire design, importing them unexpectedly into the top-level design could cause problems.

If the team follows these recommendations, the project lead should be able to include the **.sdc** file with partition-specific constraints directly in the top-level project to add the **.sdc** constraints provided by the lower-level designer.

Example Step 2: Partition Designer Creates .sdc File with Partition-Specific Constraints

The lower-level designer compiles the design with the **.sdc** file with project-wide constraints and might want to add some additional constraints. In this example, the designer realizes that they must specify a false path between the register called `reg_in_1` and all destinations in this design block using the wildcard character `*`. This constraint applies entirely within the partition and must be exported to the top-level design, so it qualifies for inclusion in the **.sdc** file with partition-specific constraints. The designer first defines the `module_A_hierarchy` variable and uses it when writing the constraint as follows:

```
if {![info exists module_A_hierarchy]} {
   set module_A_hierarchy ""
}
set_false_path -from [get_registers ${module_A_hierarchy}reg_in_1] -to
[get_registers ${module_A_hierarchy}*]
```

Consolidating the SDC Files in the Top-Level Design

When the lower-level designers complete their designs, they export the results to the top-level project lead. The project lead receives the exported **.qxp** file and a copy of the **.sdc** file with partition-specific constraints.

To set up the top-level **.sdc** constraint file to accept the **.sdc** files from the lower-level projects, the top-level **.sdc** file should define the hierarchy variables specified in the lower-level **.sdc** files. List the variable for each lower-level partition and set it to the hierarchy path, up to and including the instantiation of the lower-level partition in the top-level project, including the final '`|`' hierarchy character.

To ensure that the **.sdc** files are used in the correct order, the project lead can use the Tcl Source command to load each **.sdc** file.

Example Step 3: Project Lead Performs Final Timing Analysis and Sign-off

With these commands, the project lead’s top-level **.sdc** file looks like the following example:

```
create_clock -name {clk} -period 3.000 -waveform { 0.000 1.500 }
[get_ports {TOP_LEVEL_CLK}]
# Include the lower-level SDC file
set module_A_hierarchy "module_A:inst|"  # Note the final '|' character
source <partition-specific constraint file such as
..\module_A\module_A_constraints>.sdc
```

When the project lead performs top-level timing analysis, the false path assignment from the lower-level `module_A` project expands to the following:

```
set_false_path -from module_A:inst|reg_in_1 -to module_A:inst|*
```

Adding the hierarchy path as a prefix to the SDC command makes the constraint legal in the top-level project, and ensures that the wildcard does not affect any nodes outside of the partition that it was intended to target.

By following the guidelines in this section, constraint propagation between the projects has been managed effectively.

Recommended Design Flows and Compilation Application Examples

This section provides design flows for solving common timing closure and team-based design issues using incremental compilation. Each flow describes the situation in which it should be used, and gives a step-by-step description of the commands required to implement the flow.

The following top-down incremental design flow examples reduce compilation time while making incremental changes to the design. The examples also allow you to achieve timing closure more quickly by optimizing or preserving the results for some of your design partitions:

- "Reducing Compilation Time When Changing a Source File for One Partition"
- "Preserving Results for Some Partitions Before Adding Other Partitions" on page 2–51
- "Optimizing a Timing-Critical Partition to Achieve Timing Closure" on page 2–50
- "Debugging Incrementally with the SignalTap II Logic Analyzer" on page 2–52

All examples assume you have set up the project to use the full incremental compilation flow, using the steps described in "Quick Start Guide—Summary of Incremental Compilation" on page 2–7.

The following bottom-up design flow examples illustrate team-based design methodologies and design reuse:

- "Implementing a Team-Based Bottom-Up Design Flow" on page 2–53
- "Performing Design Iteration in a Bottom-Up Design Flow" on page 2–56
- "Creating Hard-Wired Macros (or Precompiled Design Blocks) for IP Reuse" on page 2–57
- "Using an Exported Partition to Send a Design without Including Source Files" on page 2–59

Reducing Compilation Time When Changing a Source File for One Partition

Use this flow to update the source file in one partition without having to recompile the other parts of the design. To reduce the compilation time, keep the post-fit netlists for the unchanged partitions. This also preserves the performance for these blocks, which reduces additional timing closure efforts.

Example background: You have just performed a lengthy, complete compilation of a design that consists of multiple partitions. An error is found in the HDL source file for one partition and it is being fixed. Because the design is currently meeting timing requirements and the fix is not expected to affect timing performance, it makes sense to compile only the affected partition and preserve the rest of the design.

Perform the following steps to update the single source file:

1. Apply and save the fix to the HDL source file.
2. On the Assignments menu, click **Design Partitions Window**.
3. For the partitions that should be preserved, change the Netlist Type to **Post-Fit**. You can set the **Fitter Preservation Level** to either **Placement** or **Placement and Routing**. For the partition that contains the fix, you can change the netlist type to **Source File**. (Making the **Source File** setting is optional because the Quartus II software recompiles partitions by default if changes are detected in a source file.)
4. Click **Start Compilation** to incrementally compile the fixed HDL code. This compilation should take much less time than the initial full compilation.
5. Run simulation again to ensure that the error is fixed, and use the Timing Analyzer report to ensure that timing results have not degraded.

Optimizing a Timing-Critical Partition to Achieve Timing Closure

Use this flow to optimize the results of one partition when the other partitions in the design already meet their requirements. You can use this flow iteratively to lock down the performance of one partition and then move on to optimization of another partition.

Example background: You have just performed a lengthy full compilation of a design that consists of multiple partitions. The Timing Analyzer reports that the clock timing requirement is not met and you have to optimize one particular partition. You want to try optimization techniques such as raising the Placement Effort Multiplier, enabling Physical Synthesis, and running the Design Space Explorer. Because these techniques all involve significant compilation time, it makes sense to apply them to only the partition in question.

Perform the following steps to preserve the results for partitions that meet their timing requirements, and recompile a timing-critical partition with new optimization settings:

1. On the Assignments menu, click **Design Partitions Window**.
2. For the partition in question, set the Netlist Type to **Post-Synthesis** if you are changing a Fitter setting, such as raising the Placement Effort Multiplier. This causes the partition to be placed and routed with the new Fitter settings (but not resynthesized) during the next compilation. Set the Netlist Type to **Source File** if you are changing an optimization setting that affects synthesis, such as certain Physical Synthesis optimizations.
3. For the remaining partitions (including the top-level entity), set the Netlist Type to **Post-Fit**. Set the **Fitter Preservation Level** to **Placement** to allow for the most flexibility during routing. These partitions are preserved during the next compilation.
4. Apply the desired optimization settings.

5. Click **Start Compilation** to perform incremental compilation on the design with the new settings. During this compilation, the Partition Merge stage automatically merges the critical partition's netlist with the post-fit netlists of the remaining partitions. The Fitter then refits only the required partition. Because the effort is reduced as compared to the initial full compilation, the compilation time is also reduced.

To use the Design Space Explorer, perform the following steps:

1. Repeat steps 1–3 of the previous set of steps.
2. Save the project and run the Design Space Explorer.

Preserving Results for Some Partitions Before Adding Other Partitions

Use this flow to compile one set of partitions in isolation and lock the placement to preserve the results while you complete the rest of your design.

Example background: To reduce compilation time and help achieve timing closure, you decide to use one of the following compilation flows:

In the first variation, a timing-critical partition is placed and routed by itself, with extra optimizations turned on (manually or with the Design Space Explorer). After timing closure is achieved for this partition, its content and placement are preserved and the remaining partitions are fit with normal or reduced optimization levels so that the compilation time can be reduced. For example, you can compile an IP block that comes with instructions to perform optimization before you incorporate the rest of your custom logic.

In the second variation, only the quick-compiling partitions are placed and routed initially with normal or reduced optimization levels, using floorplan location assignments to reserve space in the floorplan for the partitions to be added in the future. These quick-compiling partitions are then preserved so they do not have to be compiled again when the last partitions are introduced, with extra optimizations turned on (manually or with the Design Space Explorer).

To implement this design flow, perform the following steps:

1. Partition the design and create floorplan location assignments.
2. For the partitions to be compiled first, on the Assignments menu, click **Design Partitions Window and** set Netlist Type to **Source File**.
3. For the remaining partitions (other than any direct or indirect parents of partitions in step 2), set the Netlist Type to **Empty**.
4. To compile with the desired optimizations turned on, click **Start Compilation**.
5. Check Timing Analyzer reports to ensure that timing requirements are met. If so, proceed to step 6. Otherwise, repeat steps 4 and 5 until the requirements are met.
6. In the **Design Partitions Window**, set the Netlist Type to **Post-Fit** for the first partitions. Set the **Fitter Preservation Level to Placement and Routing** only if necessary to preserve results of the timing-critical blocks; otherwise, use **Placement** to allow for the most flexibility during routing.
7. Change the Netlist Type from **Empty** to **Source File** for the remaining partitions.

8. Set the appropriate level of optimizations and compile the design. Changing the optimizations at this point does not affect any fitted partitions, because each partition has its Netlist Type set to **Post-Fit**.
9. Check Timing Analyzer reports to ensure that timing requirements are met. If not, make design or option changes and repeat step 8 and step 9 until the requirements are met.

This flow is similar to a bottom-up design flow in which a module is implemented separately and is merged into the rest of the design afterwards. Generally, optimization in this flow works only if each critical path is contained within a single partition. This is one reason why both the inputs and outputs of each partition should be registered. Ensure that if there are any partitions representing a design file that is missing from the project, you create a placeholder wrapper file that defines the port interface. Refer to "Empty Partitions" on page 2–22 for more information.

Debugging Incrementally with the SignalTap II Logic Analyzer

Incremental compilation enables you to preserve the synthesis and fitting results of your original design and add the SignalTap II Logic Analyzer to your design without recompiling your original source code.

Use this flow to reduce compilation times when adding the logic analyzer to debug your design, or when you want to modify the configuration of the SignalTap II file without modifying your logic design or its placement.

It is not necessary to create any design partitions to use the SignalTap II Incremental Compilation feature. When your project has the default **Full incremental compilation** option turned on, the SignalTap II Logic Analyzer acts as its own separate design partition.

Perform the following steps to use the SignalTap II Embedded Logic Analyzer in an incremental compilation flow:

1. On the Assignments menu, click **Design Partitions Window**.
2. Set the Netlist Type to **Post-fit** for all partitions to preserve their placement.

The netlist type for the top-level partition defaults to **Source File**, so be sure to change this Top partition in addition to any design partitions that you created.

3. If you have not already compiled the design with the current set of partitions, perform a full compilation. If the design has already been compiled with the current set of partitions, the design is ready to add the SignalTap II Logic Analyzer.
4. Set up your SignalTap II file using the **SignalTap II: post-fitting** filter in the **Node Finder** to add signals for logic analysis. This allows the Fitter to add the SignalTap II logic to the post-fit netlist without modifying the design results.

   To add signals from the pre-synthesis netlist, set the partition's Netlist Type to **Source File** and use the **SignalTap II: pre-synthesis** filter in the **Node Finder**. This allows the software to resynthesize the partition and tap directly to the pre-synthesis node names that you choose. In this case, the partition is refit, so the placement is typically different from previous fitting results.

Do not use the netlist type **Post-Synthesis** with the SignalTap II Logic Analyzer.

For more information about setting up the SignalTap II Logic Analyzer, refer to the *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Implementing a Team-Based Bottom-Up Design Flow

This example describes how to use incremental compilation in a bottom-up design flow.

Example background: A project consists of several lower-level subdesigns that are implemented separately by different designers. The top-level project instantiates each of these subdesigns exactly once. The subdesign designers want to optimize their designs independently and pass on the results to the project lead.

As the project lead in this scenario, perform the following steps to prepare the design for a successful bottom-up design methodology:

1. Create a new Quartus II project to ultimately contain the full implementation of the entire design.
2. To prepare for the bottom-up methodology, create a "skeleton" of the design that defines the hierarchy for the subdesigns implemented by separate designers. The top-level design implements the top-level entity in the design and instantiates wrapper files that represent each subdesign by defining only the port interfaces but not the implementation.
3. Make project-wide settings. Select the device, make global assignments for clocks and device I/O ports, and make any global signal constraints to specify which signals can use global routing resources.
4. Make design partition assignments for each subdesign and set the Netlist Type for each design partition to be imported to **Empty** in the Design Partitions window.
5. Create LogicLock regions for each of the lower-level partitions to create a design floorplan. This floorplan should consider the connectivity between partitions and estimates of the size of each partition based on any initial implementation numbers and knowledge of the design specifications.
6. On the Project menu, click **Generate Bottom-Up Design Partition Scripts**, or launch the script generator from Tcl or the command prompt.
7. Make any changes to the default script options as desired. Altera recommends that you pass all the default constraints, including LogicLock region, for all partitions and virtual pin location assignments. Altera further recommends that you add a maximum delay timing constraint for the virtual I/O connections in each partition to help timing closure during integration at the top level. If lower-level projects have not already been created by the other designers, use the partition script to set up the projects so that you can easily take advantage of makefiles.
8. Provide all lower-level designers with the Tcl file to create their project with the appropriate constraints. If you are using makefiles, provide the makefile for each partition.

As the designer of a lower-level subdesign in this example, perform the appropriate set of steps to successfully export your design, whether your design team is using makefiles or exporting and importing the design manually.

If you are using makefiles, perform the following steps:

1. Use the **make** command and the makefile provided by the project lead to create a Quartus II project with all design constraints, and compile the project.
2. The information about which source file should be associated with which partition is not available to the software automatically, so you must specify this information in the makefile. You must specify the dependencies before the software rebuilds the project after the initial call to the makefile.
3. When you have achieved the desired compilation results and the design is ready to be imported into the top-level design, the project lead can use the `master_makefile` command to export this lower-level partition and create a **.qxp** file, and then import it into the top-level design.

If you are not using makefiles, perform the following steps:

1. Create a new Quartus II project for the subdesign.
2. Make LogicLock region assignments and global assignments (including clock settings) as specified by the project lead.
3. Make Virtual Pin assignments for ports which represent connections to core logic instead of external device pins in the top-level module.
4. Make floorplan location assignments to the Virtual Pins so they are placed in their corresponding regions as determined by the top-level module. This provides the Fitter with more information about the timing constraints between modules. Alternatively, you can apply timing I/O constraints to the paths that connect to virtual pins.
5. Proceed to compile and optimize the design as needed.
6. When you have achieved the desired compilation results, on the Project menu, click **Export Design Partition**. The **Export Design Partition** dialog box appears.
7. Under **Netlist to export**, select the netlist type **Post-fit netlist** to preserve the placement and performance of the subdesign, and turn on **Export routing** to include the routing information if required. You can export **Post-synthesis netlist** instead if placement or performance preservation is not required.
8. Provide the **.qxp** file to the project lead.

Finally, as the project lead in this example, perform the appropriate set of steps to import the files sent in by the designers of each lower-level subdesign partition.

If you are using makefiles, perform the following steps:

1. Use the `master_makefile` command to export each lower-level partition and create **.qxp** files, and then import them into the top-level design.
2. The software does not have all the information about which source files should be associated with which partition, so you must specify this information in the makefile. The software cannot rebuild the project if source files change unless you specify the dependencies.

If you are not using makefiles, perform the following steps:

1. After you obtain the **.qxp** file for each subdesign from the other designers on the team, on the Project menu, click **Import Design Partition** and specify the partition in the top-level project that is represented by the subdesign **.qxp** file.
2. Repeat the import process described in step 1 for each partition in the design. After you have imported each partition once, select all the design partitions and use the **Reimport using latest import files at previous locations** option to import all of the files from their previous locations at one time.

Resolving Assignment Conflicts During Import

When importing the subdesigns, the project lead may notice some assignment conflicts. This can occur, for example, if the subdesign designers changed their LogicLock regions to account for additional logic or placement constraints, or if the designers applied I/O port timing constraints that differ from constraints added to the top-level project by the project lead. To address these conflicts, the project lead can take one or both of the following actions:

- Allow new assignments to be imported.
- Allow existing assignments to be replaced or updated.

When LogicLock region assignment conflicts occur, the project lead may take one of the following actions:

- Allow the imported region to replace the existing region.
- Allow the imported region to update the existing region.
- Skip assignment import for regions with conflicts.

The project lead can address all of these situations using **Advanced Import Settings** as described in "Importing Assignments and Advanced Import Settings" on page 2–38.

If the placement of different subdesigns conflict, the project lead can also set the set the partition's **Fitter Preservation Level** to **Netlist Only**, which allows the software to re-perform placement and routing with the imported netlist.

Importing a Partition to be Instantiated Multiple Times

In this variation of the scenario, one of the subdesigns is instantiated more than once in the top-level design. The designer of the subdesign may want to compile and optimize the entity once under a lower-level project, and then import the results as multiple partitions in the top-level project.

In this case, placement conflict resolution as described in "Resolving Assignment Conflicts During Import" on page 2–55 is mandatory because the top-level partitions share the same imported post-fit netlist. If you import multiple instances of a subdesign in the top-level design, the imported LogicLock regions are automatically set to Floating status.

If you resolve conflicts manually, you can use the import options and manual LogicLock assignments to specify the placement of each instance in the top-level design.

Performing Design Iteration in a Bottom-Up Design Flow

Use this flow if you re-optimize lower-level partitions in a bottom-up compilation by incorporating additional constraints from the integrated top-level design.

Example background: A project consists of several lower-level subdesigns that have been exported from separate Quartus II projects and imported into the top-level design in a bottom-up compilation flow. In this example, integration at the top level has failed because the timing requirements are not met. The timing requirements are met in each individual lower-level project, but critical inter-partition paths in the top level are causing timing requirements to fail.

After trying various optimizations at the top level, the project lead determines that the design cannot meet the timing requirements given the current lower-level partition placements that were imported. The project lead decides to pass additional constraints to the lower-level projects to improve the placement.

To implement this design flow, perform the following steps:

1. In the top-level design, on the Project menu, click **Generate Bottom-Up Design Partition Scripts**, or launch the script generator from Tcl or the command line.
2. Because lower-level projects have already been created for each partition, turn off **Create lower-level project if one does not exist**.
3. Make any additional changes to the default script options as desired. Altera recommends that you pass all the default constraints, including LogicLock regions, for all partitions and virtual pin location assignments. Altera also recommends that you add a maximum delay timing constraint for the virtual I/O connections in each partition.
4. The Quartus II software generates Tcl scripts for all partitions, but in this scenario, you would focus on the partitions that make up the cross-partition critical paths. The following assignments are important in the script:
   - Virtual pin assignments for module pins not connected to device I/O ports in the top-level design.
   - Location constraints for the virtual pins that reflect the initial top-level placement of the pin's source or destination. These help make the lower-level placement "aware" of its surroundings in the top level, leading to a greater chance of timing closure during integration at the top level.
   - `INPUT_MAX_DELAY` and `OUTPUT_MAX_DELAY` timing constraints on the paths to and from the I/O pins of the partition. These constrain the pins to optimize the timing paths to and from the pins.
5. The lower-level designers source the file provided by the project lead.
   - To source the Tcl script from the Quartus II GUI, on the Tools menu, click **Utility Windows** and open the Tcl console. Navigate to the script's directory, and type the following command:

     ```
     source <filename> ↵
     ```
   - To source the Tcl script at the system command prompt, type the following command:

     ```
     quartus_cdb -t <filename>.tcl ↵
     ```

6. The lower-level designers recompile their designs with the new assignments and ensure that the internal timing requirements are met.
7. The lower-level designers re-export their results.
8. The top-level designer re-imports the results.
9. You can now analyze the design to determine whether the timing requirements have been achieved. Because the lower-level partitions were compiled with more information about connectivity at the top level, it is more likely that the inter-partition paths have improved placement which helps to meet the timing requirements.

Creating Hard-Wired Macros (or Precompiled Design Blocks) for IP Reuse

Use this design flow to create a hard-wired macro or precompiled IP block that can be instantiated in a top-level design. This flow provides the ability to export a design block with post-synthesis or placement (and, optionally, routing) information and to import any number of copies of this pre-compiled macro into another design.

Example background: An IP provider wants to produce and sell an IP core for a component to be used in higher-level systems. The IP provider wants to optimize the placement of their block for maximum performance in a specific Altera device and then deliver the placement information to their end customer. To preserve their IP, they also prefer to send a compiled netlist instead of providing the HDL source code to their customer.

The customer first specifies which Altera device is being used for this project and provides the design specifications.

As the IP provider in this example, perform the following steps to export a preplaced IP core (or hard macro):

1. Create a black box wrapper file that defines the port interface for the IP core and provide the file to the customer to instantiate as an empty partition in the top-level design.
2. Create a Quartus II project for the IP core.
3. Create a LogicLock region for the design hierarchy to be exported.

Altera recommends creating a floorplan using LogicLock regions, although it is not required for the generation and use of **.qxp** files. Using a LogicLock region for the IP core allows the customer to create an empty placeholder region to reserve space for the IP in the design floorplan. This ensures there are no conflicts with the top-level design logic, and that the IP core does not affect the timing performance of other logic in the top-level design.

LogicLock regions can be effective to reduce resource utilization conflicts and to enable performance preservation. In addition, without LogicLock regions, placement can be preserved only in an absolute manner. With LogicLock regions, you can preserve placement absolutely or relative to the origin of the associated regions. This is important when a **.qxp** file is imported for multiple partition hierarchies in the same project, because in this case, the location of at least one instance in the top-level project does not match the location used by the IP provider.

4. If required, add any logic (such as PLLs or other logic defined in the customer's top-level design) around the design hierarchy to be exported. If you do so, create a design partition for the design hierarchy that is to be exported as an IP core.

   For more information, refer to "Exporting a Lower-Level Block within a Project" on page 2–35.

5. Optimize the design and close timing to meet the design specifications.

6. Export the appropriate level of hierarchy into a single **.qxp** file. Following a successful compilation of the project, you can generate a **.qxp** file from the GUI, the command-line, or with Tcl commands:

   - If you are using the Quartus II GUI, use the **Export Design Partition** command.
   - If you are using command-line executables, run **quartus_cdb** with the `--incremental_compilation_export` option.
   - If you are using Tcl commands, use the following command: `execute_flow -incremental_compilation_export`.

7. Provide the **.qxp** file to the customer. Note that you do not have to send any of your design source code to the customer; the design netlist and placement and routing information is contained within this single file.

As the customer in this example, incorporate the IP core in your design by performing the following steps:

1. Create a Quartus II project for the top-level design that targets the same device and instantiate a copy or multiple copies of the IP core. Use a black box wrapper file to define the port interface of the IP core.

2. On the Processing menu, point to **Start** and click **Perform Analysis & Elaboration** to identify the design hierarchy.

3. Create a design partition for each instance of the IP core (refer to "Creating Design Partitions" on page 2–69) with the Netlist Type set to **Empty** (refer to "Setting the Netlist Type for Design Partitions" on page 2–19).

4. You can now continue work on your part of the design and accept the IP core from the IP provider whenever it is ready.

5. Import the **.qxp** file from the IP provider for the appropriate partition hierarchy. You can import a **.qxp** file from the GUI, the command-line, or with Tcl commands.

   - If you are using the Quartus II GUI, use the **Import Design Partition** command.
   - If you are using command-line executables, run **quartus_cdb** with the `--incremental_compilation_import` option.
   - If you are using Tcl commands, use the following command: `execute_flow -incremental_compilation_import`.

6. You can set the imported LogicLock regions to floating or move them to a new location, with the relative locations of the region contents preserved. Routing information is preserved whenever possible.

The Fitter ignores relative placement assignments if the LogicLock region's location in the top-level design is not compatible with the locations exported in the **.qxp** file.

7. You can control whether to preserve the imported netlist only, placement, or placement and routing (if the placement or placement and routing information was exported in the **.qxp** file) with the Fitter Preservation Level.

By default, the software preserves the absolute placement and routing of all nodes in the imported netlist if you choose to preserve placement and routing. However, if you use the same **.qxp** files for multiple partitions in the same project, the software preserves the relative placement for each of the imported modules (relative to the origin of the LogicLock region).

If the IP provider did not define a LogicLock region in the exported partition, the software preserves absolute placement locations and this leads to placement conflicts if the partition is imported for more than one instance.

Using an Exported Partition to Send a Design without Including Source Files

Use this flow to package a full design as a single file to send to an end customer or another design location.

Example background: A designer wants to produce a design block and needs to send out their design, but to preserve their IP, they prefer to send a synthesized netlist instead of providing the HDL source code to the recipient.

As the sender in this example, perform the following steps to export a design block:

1. Provide the device family name to the sender. If you send placement information with the synthesized netlist, also provide the exact device selection so they can set up their project to match.
2. Create a black box wrapper file that defines the port interface for the design block and provide it to the recipient for instantiating the block as an empty partition in the top-level design.
3. Create a Quartus II project for the design block, and complete the design.
4. Export the appropriate level of hierarchy into a single **.qxp** file. If you use the Quartus II GUI, use the **Export Design Partition** command (refer to "Exporting a Lower-Level Partition to be Used in a Top-Level Project" on page 2–34").
5. Select the option to include just the **Post-synthesis netlist** if you do not have to send placement information. If the recipient wants to reproduce your exact Fitter results, you can select the **Post-fitting netlist** option, and optionally enable **Export routing**.
6. Provide the **.qxp** file to the recipient. Note that you do not have to send any of your design source code.

As the recipient in this example, first create a Quartus II project for your top-level design and ensure that your project targets the same device (or at least the same device family if the **.qxp** file does not include placement information), as specified by the IP provider sending the design block. Instantiate the design block using the port information provided. Then incorporate the design block into a top-level design by performing one of the following procedures.

To use the **.qxp** file as a design file in your design, simply add the **.qxp** file from the IP provider as a source file in your Quartus II project. When you use a **.qxp** file as a source file in this way, you cannot import any post-fit netlist information. You can choose whether you want the file to be a partition in the top-level project.

To import the design instance from the **.qxp** file as a design partition and optionally include the post-fit netlist information, perform the following steps:

1. On the Processing menu, point to **Start** and click **Perform Analysis & Elaboration** to identify the design hierarchy.
2. Create a design partition for the design instance from the **.qxp** file (refer to "Creating Design Partition Assignments" on page 2–16) with the Netlist Type set to **Empty** (refer to "Setting the Netlist Type for Design Partitions" on page 2–19).
3. Import the **.qxp** file from the IP provider for the appropriate partition hierarchy. If you are using the Quartus II GUI, use the **Import Design Partition** command and browse to the **.qxp** file provided (refer to "Using a .qxp File as a Source File in the Top-Level Project" on page 2–36).
4. If the sender provides Fitter information, you can control whether to preserve the imported netlist only, placement, or placement and routing, with the Fitter Preservation Level.

Incremental Compilation Restrictions

This section documents the restrictions and limitations that you may encounter when using incremental compilation, including interactions with other Quartus II features. Some restrictions apply to both top-down and bottom-up design flows, while some additional restrictions apply only to bottom-up design flows.

The following restrictions and limitations are covered:

- "Preserving Exact Timing Performance" on page 2–61
- "When Placement and Routing May Not Be Preserved Exactly" on page 2–61
- "Using Incremental Compilation with Quartus II Archive Files" on page 2–61
- "Formal Verification Support" on page 2–61
- "Importing Encrypted IP Cores in Bottom-Up Flows" on page 2–62
- "SignalProbe Pins and Engineering Change Management with the Chip Planner" on page 2–62
- "SignalTap II Embedded Logic Analyzer in Bottom-Up Compilation Flows" on page 2–64
- "Logic Analyzer Interface in Bottom-Up Compilation Flows" on page 2–64
- "Migrating Projects with Design Partitions to Different Devices" on page 2–64
- "HardCopy Compilation and Migration Flows" on page 2–64
- "Assignments Made in HDL Source Code in Bottom-Up Flows" on page 2–65
- "Restrictions on Megafunction Partitions" on page 2–65
- "Register Packing and Partition Boundaries" on page 2–66
- "I/O Register Packing" on page 2–66

Preserving Exact Timing Performance

Timing performance might change slightly in a partition with placement and routing preserved when other partitions are incorporated or re-placed and routed. Timing changes are due to changes in parasitic loading or crosstalk introduced by the other (changed) partitions. These timing changes are very small, typically less than 30 ps on a timing path. Additional fan-out on routing lines when partitions are added can also degrade timing performance.

To ensure that a partition continues to meet its timing requirements when other partitions change, a very small timing margin might be required. The Fitter automatically works to achieve such margin when compiling any design, so you do not need to take any action.

When Placement and Routing May Not Be Preserved Exactly

The Fitter may have to refit affected nodes if the two nodes are assigned to the same location, due to imported netlists or empty partitions set to re-use a previous post-fit netlist. There are two cases in which routing information cannot be preserved exactly. First, when multiple partitions are imported, there might be routing conflicts because two lower-level blocks could be using the same routing wire, even if the floorplan assignments of the lower-level blocks do not overlap. These routing conflicts are automatically resolved by the Quartus II Fitter re-routing on the affected nets. Second, if an imported LogicLock region is moved in the top-level design, the relative placement of the nodes is preserved but the routing cannot be preserved, because the routing connectivity is not perfectly uniform throughout a device.

Using Incremental Compilation with Quartus II Archive Files

The post-synthesis and post-fitting netlist information for each design partition is stored in the project database, the **incremental_db** directory. When you archive a project, the database information is not included in the archive unless you include the compilation database in the **.qar** file.

Altera recommends that you include the database files in the **Archive Project** dialog box so compilation results are preserved. Click **Advanced**, and choose a file set that includes the compilation database, or turn on **Compilation database files** to create a Custom file set.

When you include the database, the file size of the **.qar** archive file may be significantly larger than an archive without the database.

The netlist information for imported partitions is already saved in the corresponding **.qxp** file. Imported **.qxp** files are automatically saved in a subdirectory called **imported_partitions**, so you do not need to archive the project database to keep the results for imported partitions. When you restore a project archive, the partition is automatically reimported from the **.qxp** file in this directory if it is available.

Formal Verification Support

You cannot use design partitions for incremental compilation if you are creating a netlist for a formal verification tool.

Importing Encrypted IP Cores in Bottom-Up Flows

Proper license information is required to compile encrypted IP cores. If an IP core is imported as a **.qxp** file from another Quartus II project in a bottom-up compilation flow, the top-level designer must have the correct license. That is, you require a full license to generate an unrestricted programming file. If you do not have a license, but the IP in the **.qxp** file was compiled with OpenCore Plus hardware evaluation support, you can generate an evaluation programming file without a license. If the IP supports OpenCore simulation only, you can fully compile the design and generate a simulation netlist, but you cannot create programming files unless you have a full license.

SignalProbe Pins and Engineering Change Management with the Chip Planner

When you create SignalProbe pins or use the Resource Property Editor to make changes due to engineering change orders (ECOs) after performing a full compilation, recompiling the entire design is not necessary. These changes are made directly to the netlist without performing a new placement and routing. You can preserve these changes using a post-fit netlist with placement and routing. When a partition is recompiled, SignalProbe pins and ECO changes in unaffected partitions are preserved.

For more information about using the SignalProbe feature to debug your design, refer to the *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*. For more information about using the Chip Planner and the Resource Property Editor to make ECOs, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

To preserve SignalProbe pins or ECO changes, the partition netlist type should be set to **Post-Fit** with the Fitter Preservation Level set to **Placement and Routing**. If any partitions with SignalProbe pins or ECO changes are set to **Post-Fit** without Routing or to **Netlist Only**, the software issues a warning and internally uses the Post-fit netlist with **Placement and Routing**. If the partitions are set to use the source code or a Post-synthesis netlist, the software issues a warning and the Post-fit SignalProbe pins or ECO changes are not included in the new compilation. However, partitions can become linked due to the SignalProbe pins or ECO changes, as described below, in which case all linked partitions inherit the netlist type from the linked partition with the highest level of preservation.

Linked Partitions Due to SignalProbe Pins or ECO Changes

If ECO changes affect more than one partition or the connection between any partitions, the partitions become linked. All of the higher-level “parent” partitions up to their nearest common parent are also linked. In this case, the connection between the partitions is actually defined outside of the two partitions immediately affected, so all the partitions must be compiled together. All linked partitions use the same netlist type, and they inherit the netlist type from the linked partition with the highest level of preservation.

When a SignalProbe pin is created, it affects the partition that contains the node being probed. In addition, any pipeline registers are created in the same partition as the node being probed. The SignalProbe output pin is assigned to the top-level partition. Therefore, there is a new connection formed between the top-level partition and the lower-level partition that is being probed. Because of this connection, the lower-level partition being probed and all of the higher-level "parent" partitions up to the top level become linked. All linked partitions use the same netlist type, and they inherit the netlist type from the linked partition with the highest level of preservation.

When partitions are linked, they can change which netlists are preserved when you recompile the design, as follows:

- If all the linked partitions are set to use the source code or a post-synthesis netlist, the partitions are refit as normal. In this case, the SignalProbe pins or ECO changes are not included in the new netlists, so you must reapply the changes in the Change Manager.
- If any of the linked partitions are set to the **Post-Fit** netlist type, and there are no source code changes, the software issues a warning and internally uses the post-fit netlist with placement and routing for all linked partitions. By preserving the appropriate post-fit netlists, the software can preserve the SignalProbe pins or ECO changes.
- If any of the linked partitions are set to the **Post-Fit (Strict)** netlist type, the software issues a warning and internally uses the post-fit netlist with placement and routing for all linked partitions, regardless of any source code changes. By preserving the appropriate post-fit netlists, the software can preserve the SignalProbe pins or ECO changes. Note that in this case, source code changes in any of the linked partitions are not included in the new netlist.
- If any of the linked partitions are recompiled due to a change in source code, the software issues a warning and recompiles the other linked partitions as well. When this occurs, the SignalProbe pins or ECO changes are not included in the new netlist, so you must reapply the changes in the Change Manager.

Exported Partitions

In a bottom-up incremental compilation, the exported netlist includes all currently saved SignalProbe pins and ECO changes. This might require flattening and combining lower-level partitions in the child project to avoid partition boundary violations at the top level. After importing this netlist, changes made in the lower-level partition do not appear in the Change Manager at the top level.

If you make any ECO changes that affect the interface to the lower-level partition, the software issues a warning message during the export process that this netlist does not work in the top-level design without modifying the top-level HDL code to reflect the lower-level change.

SignalTap II Embedded Logic Analyzer in Bottom-Up Compilation Flows

You can use the SignalTap II Embedded Logic Analyzer in any project that you can compile and program into an Altera device.

You cannot export a lower-level project that uses a SignalTap II File (**.stp**) for the SignalTap II Logic Analyzer in a bottom-up incremental compilation flow. You must disable the SignalTap II feature and recompile the design before you export the design as a partition.

You can instantiate the SignalTap II megafunction directly in your lower-level design (instead of using an **.stp** file) and export the entire design to the top level in a bottom-up flow.

You can tap any nodes in a Quartus II project, including nodes imported from other projects. Use the appropriate filter in the Node Finder to find your node names. Use **SignalTap II: post-fitting** if the Netlist Type is **Post-Fit** to incrementally tap node names in the post-fit netlist database. Use **SignalTap II: pre-synthesis** if the Netlist Type is **Source File** to make connections to the source file (pre-synthesis) node names when you synthesize the partition from the source code.

For details about using the SignalTap II logic analyzer in an incremental design flow, refer to the *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Logic Analyzer Interface in Bottom-Up Compilation Flows

You can use the Logic Analyzer Interface in any project that you can compile and program into an Altera device. You cannot export a lower-level project that uses the Logic Analyzer Interface in a bottom-up incremental compilation flow. You must disable the Logic Analyzer Interface feature and recompile the design before you export the design as a partition.

For more information about the Logic Analyzer Interface, refer to the *In-System Debugging Using External Logic Analyzers* chapter in volume 3 of the *Quartus II Handbook*.

Migrating Projects with Design Partitions to Different Devices

Partition assignments are still valid if you migrate to a different device density or family. LogicLock region size is valid if you migrate to a device in the same family, but the origin location is not valid. Specific floorplan assignments are not valid for different devices or families because the location coordinates change between devices.

Post-synthesis netlists are valid if you migrate to a different-sized device in the same family. Post-fit netlists are not valid if you migrate to a different device density or family.

HardCopy Compilation and Migration Flows

HardCopy APEX and HardCopy Stratix Devices

Incremental compilation with the Quartus II software is not supported for HardCopy APEX or HardCopy Stratix design flows.

HardCopy ASIC Migration Flows

Top-down incremental compilation is supported for the base family in HardCopy migration flows for both the FPGA first and HardCopy first flows. Design partition assignments are migrated to the companion device. However, you can not make changes to the design after migration because the design would not match the compilation results for the base family. Therefore, you can perform top-down incremental compilation on one device family, but cannot perform any incremental compilations after migration.

The Netlist Only preservation level is not supported for Post-fit netlists for FPGA or HardCopy ASIC device compilations when a migration device is specified (that is, for HardCopy ASIC device compilations with a FPGA migration device, or FPGA device compilations with a HardCopy ASIC migration device).

Bottom-up incremental compilation is not supported in HardCopy ASIC or FPGA device compilations when there is a migration device setting. The Revision Compare feature requires that the HardCopy ASIC and FPGA netlists are the same. Therefore, all operations performed on one revision must also occur on the other revision. This is accomplished by logging all operations and replaying them on the other revision. Using the bottom-up flow and importing partitions does not support this requirement. You can often use a top-down flow with **Empty** partitions to implement behavior similar to a bottom-up flow, as long as you do not change any global assignments between compilations. All global assignments must be the same for all compiled partitions, so the assignments can be reproduced in the companion device after migration.

HardCopy ASIC Stand-Alone Compilations

You can use both top-down and bottom-up incremental compilation for stand-alone HardCopy ASIC compilations.

Routing preservation is not supported for HardCopy ASICs. Therefore, the **Placement and Routing** preservation level is not available, and routing cannot be exported in the bottom-up flow.

Assignments Made in HDL Source Code in Bottom-Up Flows

Assignments made with I/O primitives or the `altera_attribute` HDL synthesis attribute in lower-level partitions are passed to the top-level design, but do not appear in the top-level QSF file or Assignment Editor. These assignments are considered part of the source netlist files. You can override assignments made in these source files by changing the value with an assignment in the top-level design.

Restrictions on Megafunction Partitions

The Quartus II software does not support partitions for megafunction instantiations. If you use the MegaWizard™ Plug-In Manager to customize a megafunction variation, the MegaWizard-generated wrapper file instantiates the megafunction. You can create a partition for the MegaWizard-generated megafunction custom variation wrapper file.

The Quartus II software does not support creating a partition for inferred megafunctions (that is, where the software infers a megafunction to implement logic in your design). If you have a module or entity for the logic that is inferred, you can create a partition for that hierarchy level in the design.

The Quartus II software does not support creating a partition for any Quartus II internal hierarchy that is dynamically generated during compilation to implement the contents of a megafunction.

Register Packing and Partition Boundaries

The Quartus II software performs register packing during compilation automatically. However, when incremental compilation is enabled, logic in different partitions cannot be packed together because partition boundaries prevent cross-boundary optimization. This restriction applies to all types of register packing, including I/O cells, DSP blocks, sequential logic, and unrelated logic. Similarly, logic from two partitions cannot be packed into the same ALM.

I/O Register Packing

Cross-partition register packing of I/O registers is allowed in certain cases where your input and output pins exist in the top-level hierarchy (and the Top partition), but the corresponding I/O registers exist in other partitions.

The following specific circumstances are required for input pin cross-partition register packing:

- The input pin feeds exactly one register.
- The path between the input pin and register includes only input ports of partitions that have one fan-out each.

The following specific circumstances are required for output register cross-partition register packing:

- The register feeds exactly one output pin.
- The output pin is fed by only one signal.
- The path between the register and output pin includes only output ports of partitions that have one fan-out each.

Output pins with an output enable signal cannot be packed into the device I/O cell if the output enable logic is part of a different partition from the output register. To allow register packing for output pins with an output enable signal, structure your HDL code or design partition assignments so that the register and tri-state logic are defined in the same partition.

Bidirectional pins are handled in the same way as output pins with an output enable signal. If the registers that need to be packed are in the same partition as the tri-state logic, you can perform register packing.

The restrictions on tri-state logic exist because the I/O atom (device primitive) is created as part of the partition that contains tri-state logic. If an I/O register and its tri-state logic are contained in the same partition, the register can always be packed with tri-state logic into the I/O atom. The same cross-partition register packing restrictions also apply to I/O atoms for input and output pins. The I/O atom must

feed the I/O pin directly with exactly one signal. The path between the I/O atom and the I/O pin must include only ports of partitions that have one fan-out each. Refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook* for more information and examples of cross-partition boundary I/O packing.

Bottom-Up Design Partition Script Limitations

The Quartus II software has some limitations related to bottom-up design partition scripts.

Warnings About Extra Clocks Due to Bottom-Up Design Partition Scripts

The generated scripts include applicable clock information for all clock signals in the top-level project. Some of those clocks may not exist in the lower-level projects, so you may see warning messages related to clocks that do not exist in the project. You can ignore these warnings or edit your constraints so the messages are not generated.

Synopsys Design Constraint Files for the TimeQuest Timing Analyzer in Bottom-Up Design Partition Scripts

As described in "Generating Bottom-Up Design Partition Scripts for Project Management" on page 2–40, design partition scripts include only clock constraints and minimum and maximum delay settings for the TimeQuest Timing Analyzer.

PLL settings and timing exceptions are not passed to lower-level designs in the scripts. Refer to"Importing SDC Constraints from Lower-Level Partitions" on page 2–45 for suggestions on managing SDC constraints between top-level and lower-level projects.

Wildcard Support in Bottom-Up Design Partition Scripts

When applying constraints with wildcards, note that wildcards are not analyzed across hierarchical boundaries. For example, an assignment could be made to these nodes: `Top|A:inst|B:inst|*`, where `A` and `B` are lower-level partitions, and hierarchy `B` is a child of `A`, that is `B` is instantiated in hierarchy `A`. This assignment is applied to modules `A`, `B`, and all children instances of `B`. However, the assignment `Top|A:inst|B:inst*` is applied to hierarchy `A`, but is not applied to the `B` instances because the single level of hierarchy represented by `B:inst*` is not expanded into multiple levels of hierarchy. To avoid this issue, ensure that you apply the wildcard to the hierarchical boundary if it should represent multiple levels of hierarchy.

When using the wildcard to represent a level of hierarchy, only single wildcards are supported. This means assignments such as `Top|A:inst|*|B:inst|*` are not supported. The Quartus II software issues a warning in these cases.

Derived Clocks and PLLs in Bottom-Up Design Partition Scripts

If a clock in the top level is not directly connected to a pin of a lower-level partition, the lower-level partition does not receive assignments and constraints from the top-level pin in the design partition scripts.

This issue is of particular importance for clock pins that require timing constraints and clock group settings. Problems can occur if your design uses logic or inversion to derive a new clock from a clock input pin. Make appropriate timing assignments in your lower-level Quartus II project to ensure that clocks are not unconstrained.

In addition, if you use a PLL in your top-level design and connect it to lower-level partitions, the lower-level partitions do not have information about the multiplication or phase shift factors in the PLL. Make appropriate timing assignments in your lower-level Quartus II project to ensure that clocks are not unconstrained or constrained with the incorrect frequency. Alternately, manually duplicate the top-level derived clock logic or PLL in the lower-level design file to ensure that you have the correct multiplication or phase-shift factors, compensation delays and other PLL parameters for complete and accurate timing analysis. Create a design partition for the rest of the lower-level design logic for export to the top level. When the lower-level design is complete, export only the partition that contains the relevant logic with the feature described in "Exporting a Lower-Level Block within a Project" on page 2–35.

Pin Assignments for GXB and LVDS Blocks in Bottom-Up Design Partition Scripts

Pin assignments for high-speed GXB transceivers and hard LVDS blocks are not written in the scripts. You must add the pin assignments for these hard IP blocks in the lower-level projects manually.

Virtual Pin Timing Assignments in Bottom-Up Design Partition Scripts

Design partition scripts use `INPUT_MAX_DELAY` and `OUTPUT_MAX_DELAY` assignments to specify inter-partition delays associated with input and output pins, which would not otherwise be visible to the project. These assignments require that the software specify the clock domain for the assignment and set this clock domain to '*'.

This clock domain assignment means that there may be some paths constrained and reported by the timing analysis engine that are not required.

To restrict which clock domains are included in these assignments, edit the generated scripts or change the assignments in your lower-level Quartus II project. In addition, because there is no known clock associated with the delay assignments, the software assumes the worst-case skew, which makes the paths seem more timing critical than they are in the top-level design. To make the paths appear less timing-critical, lower the delay values from the scripts. If required, enter negative numbers for input and output delay values.

Top-Level Ports that Feed Multiple Lower-Level Pins in Bottom-Up Design Partition Scripts

When a single top-level I/O port drives multiple pins on a lower-level module, it unnecessarily restricts the quality of the synthesis and placement at the lower-level. This occurs because in the lower-level design, the software must maintain the hierarchical boundary and cannot use any information about pins being logically equivalent at the top level. In addition, because I/O constraints are passed from the top-level pin to each of the children, it is possible to have more pins in the lower level than at the top level. These pins use top-level I/O constraints and placement options

that might make them impossible to place at the lower level. The software avoids this situation whenever possible, but it is best to avoid this design practice to avoid these potential problems. Restructure your design so that the single I/O port feeds the design partition boundary and the single connection is split into multiple signals within the lower-level partition.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF form.

For more information about Tcl scripting, refer to the *Tcl Scripting chapter* in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Preparing a Design for Incremental Compilation

To set or modify the current mode of incremental compilation, use the following command:

```
set_global_assignment -name INCREMENTAL_COMPILATION <value> ↵
```

The incremental compilation *<value>* setting must be one of the following values:

- FULL_INCREMENTAL_COMPILATION—Full incremental compilation (this is the default)
- OFF—No incremental compilation is performed

Creating Design Partitions

To create a partition, use the following command:

```
set_instance_assignment -name PARTITION_HIERARCHY \
<file name> -to <destination> -section_id <partition name>
```

The *<destination>* should be the entity's short hierarchy path. A short hierarchy path is the full hierarchy path without the top-level name (including quotation marks), for example:

```
"ram:ram_unit|altsyncram:altsyncram_component"
```

For the top-level partition, you can use the pipe (|) symbol to represent the top-level entity.

For more information about hierarchical naming conventions, refer to the *Node-Naming Conventions in Quartus II Integrated Synthesis* section in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

The <*partition name*> is the user-designated partition name, which must be unique and less than 1024 characters. The name can consist only of alphanumeric characters, and the pipe (|), colon (:), and underscore (_) characters. Altera recommends enclosing the name in double quotation marks (" ").

The <*file name*> is the name used for internally generated netlists files during incremental compilation. Netlists are named automatically by the Quartus II software based on the instance name if you create the partition in the user interface. If you are using Tcl to create your partitions, you must assign a custom file name that is unique across all partitions. For the top-level partition, the specified file name is ignored; you can use any dummy value. To ensure the names are safe and platform independent, file names must be unique regardless of case. For example, if a partition uses the file name `my_file`, no other partition can use the file name `MY_FILE`. For simplicity, Altera recommends that you base each file name on the corresponding instance name for the partition.

The software stores all netlists in the **\incremental_db** compilation database directory.

Setting Properties of Design Partitions

After a partition is created, set its Netlist Type with the following command:

```
set_global_assignment -name PARTITION_NETLIST_TYPE <value> \
-section_id <partition name>
```

The netlist type <*value*> setting is one of the following values:

- `SOURCE`—Source File
- `POST_SYNTH`—Post-Synthesis
- `POST_FIT`—Post-Fit
- `STRICT_POST_FIT`—Post-Fit (Strict)
- `IMPORTED`—Imported
- `IMPORT_BASED_POST_FIT`—Post-Fit (Import-based)
- `EMPTY`—Empty

Set the Fitter Preservation Level for a post-fit or imported netlist using the following command:

```
set_global_assignment -name PARTITION_FITTER_PRESERVATION_LEVEL \
<value> -section_id <partition name>
```

The Fitter Preservation Level <*value*> setting is one of the following values:

- `NETLIST_ONLY`—Netlist only
- `PLACEMENT`—Placement
- `PLACEMENT_AND_ROUTING`—Placement and routing
- `PLACEMENT_AND_ROUTING_AND_TILE`—Placement and routing, as well as the high-speed power tile settings

For details about these partition properties, refer to "Setting the Netlist Type for Design Partitions" on page 2–19.

Creating Floorplan Location Assignments—Excluding or Filtering Certain Device Elements (Such as RAM or DSP Blocks)

Resource filtering uses the optional Tcl argument `-exclude_resources` in the `set_logiclock_contents` function of the `incremental_compilation` Tcl package. If left unspecified, no resource filter is created.

The argument takes a list of resources-to-be-excluded as input. The list is a colon-delimited string of the keywords in Table 2–6.

Table 2–6. Resources-to-be-Excluded Keywords

Keyword	Resource
`REGISTER`	Any registers in the logic cells
`COMBINATIONAL`	Any combinational elements in the logic cells
`SMALL_MEM`	The small TriMatrix memory blocks (M512 or MLAB)
`MEDIUM_MEM`	The medium TriMatrix memory blocks (M4K or M9K)
`LARGE_MEM`	The large TriMatrix memory blocks (M-RAM or M144K)
`DSP`	Any DSP blocks
`VIRTUAL_PIN`	Any virtual pins

For example, the following command assigns everything under **alu:alu_unit** to the ALU region, excluding all the DSP and M512 blocks:

```
set_logiclock_contents -region ALU -to alu:alu_unit -exceptions \
"DSP:SMALL_MEM"
```

In the **.qsf** file, resource filtering uses an extra LogicLock membership assignment called `LL_MEMBER_RESOURCE_EXCLUDE`. For example, the following line in the **.qsf** file is used to specify a resource filter for the `alu:alu_unit` entity assigned to the ALU region. The value of the assignment takes the same format as the resource listing string taken by the previous Tcl command.

```
set_instance_assignment -name LL_MEMBER_RESOURCE_EXCLUDE \
"DSP:SMALL_MEM" -to "alu:alu_unit" -section_id ALU
```

Generating Bottom-Up Design Partition Scripts

To generate scripts, type the following Tcl command at a Tcl prompt:

```
generate_bottom_up_scripts <options> ↵
```

The command is part of the `database_manager` package, which must be loaded using the following command before the command can be used:

```
load_package database_manager
```

You must open a project before you can generate scripts.

The Tcl options are the same as those available in the GUI. The exact format of each option is specified in Table 2–7.

Table 2–7. Options for Generating Bottom-Up Partition Scripts with Tcl Commands (Part 1 of 2)

Option	Default
`-include_makefiles` <*on*|*off*>	On
`-include_project_creation` <*on*|*off*>	On

Table 2–7. Options for Generating Bottom-Up Partition Scripts with Tcl Commands (Part 2 of 2)

Option	Default
-include_virtual_pins <*on*|*off*>	On
-include_virtual_pin_timing <*on*|*off*>	On
-include_virtual_pin_locations <*on*|*off*>	On
-include_logiclock_regions <*on*|*off*>	On
-include_all_logiclock_regions <*on*|*off*>	On
-include_global_signal_promotion <*on*|*off*>	Off
-include_pin_locations <*on*|*off*>	On
-include_timing_assignments <*on*|*off*>	On
-include_design_partitions <*on*|*off*>	On
-remove_existing_regions <*on*|*off*>	On
-disable_auto_global_promotion <*on*|*off*>	Off
-bottom_up_scripts_output_directory <*output directory*>	Current project directory
-virtual_pin_delay <*delay in ns*>	*(1)*

Note to Table 2–7:

(1) No default.

The following example shows how to use the Tcl command:

```
load_package database_manager
set project test_proj
project_open $project
generate_bottom_up_scripts -bottom_up_scripts_output_directory test \
    -include_virtual_pin_timing on -virtual_pin_delay 1.2
project_close
```

Command Line Support

To generate scripts at the command prompt, type the following command:

```
quartus_cdb <project name> --generate_bottom_up_scripts=on <options> ↵
```

Once again, the options map to the same as those in the GUI. To add an option, append "--<*option_name*>=<*val*>" to the command line call.

The command prompt options are the same as those available in the GUI. They are listed in Table 2–8.

Table 2–8. Options for Generating Bottom-Up Partition Scripts (Part 1 of 2)

Option	Default
--include_makefiles_with_bottom_up_scripts=<*on*|*off*>	On
--include_project_creation_in_bottom_up_scripts=<*on*|*off*>	On
--include_virtual_pins_in_bottom_up_scripts=<*on*|*off*>	On
--include_virtual_pin_timing_in_bottom_up_scripts=<*on*|*off*>	On
--bottom_up_scripts_virtual_pin_delay=<*delay in ns*>	*(1)*
--include_virtual_pin_locations_in_bottom_up_scripts=<*on*|*off*>	On
--include_logiclock_regions_in_bottom_up_scripts=<*on*|*off*>	On
--include_all_logiclock_regions_in_bottom_up_scripts=<*on*|*off*>	On

Table 2–8. Options for Generating Bottom-Up Partition Scripts (Part 2 of 2)

Option	Default
`--include_global_signal_promotion_in_bottom_up_scripts=`<*on*|*off*>	Off
`--include_pin_locations_in_bottom_up_scripts=`<*on*|*off*>	On
`--include_timing_assignments_in_bottom_up_scripts=`<*on*|*off*>	On
`--include_design_partitions_in_bottom_up_scripts=`<*on*|*off*>	On
`--remove_existing_regions_in_bottom_up_scripts=`<*on*|*off*>	On
`--disable_auto_global_promotion_in_bottom_up_scripts=`<*on*|*off*>	Off
`--bottom_up_scripts_output_directory=`<*output directory*>	Current project directory

Note to Table 2–8:

(1) No default. You must provide this option if you are including virtual pin timing.

Exporting a Partition to be Used in a Top-Level Project

Use the `quartus_cdb` executable to export a file for a bottom-up incremental compilation flow with the following command:

```
quartus_cdb --INCREMENTAL_COMPILATION_EXPORT=<file> \
[--incremental_compilation_export_netlist_type=<POST_SYNTH|POST_FIT>]
\
[--incremental_compilation_export_partition_name=<partition name>] \
[--incremental_compilation_export_routing=<on|off>]
```

The <*file*> argument is the file path to the exported file. The <*partition name*> is the name of the partition, not its hierarchical path. If you do not specify the options, the executable uses any settings in the **.qsf** file, or it uses default values. The default partition is the top-level partition in the project, the default netlist type is post-fit, and the default for routing is on (for all device families that support exported routing).

The command reads the assignment `INCREMENTAL_COMPILATION_EXPORT_NETLIST_TYPE` to determine which netlist type to export; the default is post-fit.

You can also use the flow `INCREMENTAL_COMPILATION_EXPORT` in the `execute_flow` Tcl command contained in the `flow` Tcl package.

Use the following commands to export a **.qxp** file for a given partition, choose the netlist type, and specify whether to export routing:

```
load_package flow
set_global_assignment -name INCREMENTAL_COMPILATION_EXPORT_FILE \
<filename>
set_global_assignment -name
INCREMENTAL_COMPILATION_EXPORT_NETLIST_TYPE \
<POST_FIT|POST_SYNTH>
set_global_assignment -name \
INCREMENTAL_COMPILATION_EXPORT_PARTITION_NAME <partition name>
set_global_assignment -name INCREMENTAL_COMPILATION_EXPORT_ROUTING \
<on|off>
execute_flow -INCREMENTAL_COMPILATION_EXPORT
```

The default partition is the top-level partition in the project, the default netlist type is post-fit, and the default for routing is on (for all device families that support exported routing).

To turn on the option to always perform exportation following compilation, use the following Tcl command:

```
set_global_assignment -name AUTO_EXPORT_INCREMENTAL_COMPILATION ON
```

Importing a Lower-Level Partition into the Top-Level Project

Use the `quartus_cdb` executable to import a lower-level partition with the following command:

```
quartus_cdb -- INCREMENTAL_COMPILATION_IMPORT ↵
```

You can also use the flow called `INCREMENTAL_COMPILATION_IMPORT` in the `execute_flow` Tcl command contained in the `flow` Tcl package.

The following example script shows how to import a partition using a Tcl script:

```
load_package flow
# commands to set the import-related assignments for each partition
execute_flow --INCREMENTAL_COMPILATION_IMPORT
```

Specify the location for the imported file with the `PARTITION_IMPORT_FILE` assignment. Note that the file specified by this assignment is read only during importation. For example, the project is completely independent from any files from the lower-level projects after importing. In the command-line and Tcl flow, any partition that has this assignment set to a non-empty value is imported.

The following assignments specify how the partition should be imported:

```
PARTITION_IMPORT_PROMOTE_ASSIGNMENTS = <on|off>
PARTITION_IMPORT_NEW_ASSIGNMENTS = <on|off>
PARTITION_IMPORT_EXISTING_ASSIGNMENTS = \
replace_conflicting | skip_conflicting
PARTITION_IMPORT_EXISTING_LOGICLOCK_REGIONS = \
replace_conflicting | update_conflicting | skip_conflicting
```

Makefiles

For an example of how to use incremental compilation with a `makefile` as part of the bottom-up design flow, refer to the **read_me.txt** file that accompanies the `incr_comp` example located in the **/qdesigns/incr_comp_makefile** subdirectory. When using a bottom-up incremental compilation flow, the Generate Bottom-Up Design Partition Scripts feature can write makefiles that automatically export lower-level design partitions and import them into the top-level project whenever design files change.

Recommended Design Flows and Compilation Application Examples—Scripting and Command-Line Operation

This section provides scripting examples that cover some of the topics discussed in the main section of the chapter.

The script shown in Example 2–1 opens a project called `AB_project`, sets up two partitions, entities A and B, for the first time, and performs an initial complete compilation.

Example 2–1. AB_project

```
set project AB_project

package require ::quartus::flow
project_open $project

# Ensure that incremental compilation is turned on
set_global_assignment -name INCREMENTAL_COMPILATION \
FULL_INCREMENTAL_COMPILATION

# Set up the partitions
set_instance_assignment -name PARTITION_HIERARCHY \
 incremental_db/A_inst -to A -section_id "Partition_A"
set_instance_assignment -name PARTITION_HIERARCHY \
 incremental_db/B_inst -to B -section_id "Partition_B"

# Set the netlist types to post-fit for subsequent
# compilations (all partitions are compiled during the
# initial compilation since there are no post-fit
# netlists)
set_global_assignment -name PARTITION_NETLIST_TYPE \
 POST_FIT -section_id "Partition_A"
set_global_assignment -name PARTITION_NETLIST_TYPE \
 POST_FIT -section_id "Partition_B"

# Run initial compilation:
export_assignments
execute_flow -full_compile

project_close
```

Reducing Compilation Time When Changing a Source File for One Partition—Command-Line Example

Example background: You have run the initial compilation shown in the example script in the previous section. You have modified the HDL source file for partition **A** and want to recompile it.

Run the standard flow compilation command in your Tcl script:

```
execute_flow -full_compile
```

Or, type the following command at a system command prompt:

```
quartus_sh --flow compile AB_project↵
```

Assuming the source files for partition **B** do not depend on **A**, only **A** is recompiled. The placement of **B** and its timing performance is preserved, which also saves significant compilation time.

Optimizing the Placement for a Timing-Critical Partition

Example background: You have run the initial compilation shown in the example script under "Recommended Design Flows and Compilation Application Examples—Scripting and Command-Line Operation" on page 2–74. You would like to apply Fitter optimizations, such as physical synthesis, only to partition **A**. No changes have been made to the HDL files.

To ensure the previous compilation result for partition **B** is preserved, and to ensure that Fitter optimizations are applied to the post-synthesis netlist of partition **A**, set the netlist type of **B** to **Post-Fit** (which was already done in the initial compilation, but is repeated here for safety), and the netlist type of **A** to **Post-Synthesis**, as shown in Example 2–2:

Example 2–2. AB_project (2)

```
set project AB_project

package require ::quartus::flow
project_open $project

# Turn on Physical Synthesis Optimization
set_global_assignment -name \
PHYSICAL_SYNTHESIS_REGISTER_RETIMING ON

# For A, set the netlist type to post-synthesis
set_global_assignment -name PARTITION_NETLIST_TYPE POST_SYNTH \
-section_id "Partition_A"

# For B, set the netlist type to post-fit
set_global_assignment -name PARTITION_NETLIST_TYPE POST_FIT \
-section_id "Partition_B"

# Run incremental compilation:
export_assignments
execute_flow -full_compile

project_close
```

Conclusion

With the Quartus II incremental compilation feature described in this chapter, you can preserve the results and performance of unchanged logic in your design as you make changes elsewhere. The various applications of incremental compilation enable you to improve your productivity while designing for high-density FPGAs, using either top-down or bottom-up design methodologies.

Referenced Documents

This chapter references the following documents:

- *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*
- *In-System Debugging Using External Logic Analyzers* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*

- *Quartus II Settings File Reference Manual*
- *Quick Design Debugging Using the SignalProbe* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 2–9 shows the revision history for this chapter.

Table 2–9. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Split up Netlist Types table ■ Moved "Quartus II Exported Partition Files (.qxp)" and "Bottom-Up Incremental Compilation Summary" into the "Exporting and Importing Partitions" section. ■ Added new section "Preparing a Design for Bottom-Up Incremental Compilation" on page 2–31 ■ Removed "Exporting a Lower-Level Partition that Uses a JTAG Feature" restriction ■ Other edits throughout chapter	Updated for Quartus II software version 9.0.
November 2008 v8.1.0	■ Added new section "Importing SDC Constraints from Lower-Level Partitions" on page 2–44 ■ Removed the Incremental Synthesis Only option ■ Removed section "OpenCore Plus Feature for MegaCore Functions in Bottom-Up Flows" ■ Removed section "Compilation Time with Physical Synthesis Optimizations" ■ Added information about using a **.qxp** file as a source design file without importing ■ Reorganized several sections ■ Updated Figure 2–10	Updated for Quartus II software version 8.1.

Table 2–9. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
May 2008 v8.0.0	■ Added several references to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter ■ Simplified “Choosing a Quartus II Compilation Flow” section ■ Clarified material in “Quartus II software versions before version 8.1 included an “incremental synthesis only” option that did not preserve placement results. This option has been removed beginning with version 8.1. You can use a post-synthesis netlist to preserve synthesis results with full incremental compilation.” section, added information about “mixed” design flows, and added a note about HardCopy ASIC flows ■ Removed “When Design is Resynthesized” and “When Design is Refit” from Table 2–1. ■ Reorganized “Choosing Design Partitions” section ■ Added instructions for using the Design Partition Planner ■ Added information about design changes to Table 2–2 in “Setting the Netlist Type for Design Partitions” ■ Removed requirement for HDL wrapper file for Empty partitions that are Imported ■ Added details to “What Changes Trigger a Partition’s Automatic Resynthesis?” section ■ Added “What LogicLock Changes Trigger Refitting?” section ■ Removed existing section “Guidelines for Creating Good Design Partitions and LogicLock Regions” because it is covered in the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter and moved some of the material to other sections of the document ■ Renamed and reorganized Application Examples ■ Removed example Placing All but One Critical Partition in a Multiple-Partition Design in a Top-Down Compilation Flow and combined it with previous example ■ Added recommendation to use a version-compatible database when archiving ■ Clarified HardCopy ASIC restrictions for bottom-up flows ■ Clarified export and import of SDC constraints in bottom-up flows ■ Added “Optimizing the Placement for a Timing-Critical Partition” section ■ Added “Using an Exported Partition to Send a Design without Including Source Files” section	Updated for Quartus II software version 8.0.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

3. Quartus II Design Flow for MAX+PLUS II Users

QII51002-9.0.0

Introduction

The feature-rich Quartus® II software helps you shorten your design cycles and reduce time-to-market. With support for FLEX®, ACEX®, and MAX® device families, as well as all of Altera's newest devices, the Quartus II software is the most widely accepted Altera® design software tool today.

This chapter describes how to convert MAX+PLUS® II designs to Quartus II projects, as well as the similarities and differences between the MAX+PLUS II and Quartus II design flows. This discussion includes supported device families, GUI comparisons, and the advantages of the Quartus II software.

There are many features in the Quartus II software to help MAX+PLUS II users easily transition to the Quartus II software design environment. These include a customizable **Look & Feel** feature, which changes the GUI to display menus, toolbars, and utility windows as they appear in the MAX+PLUS II software without sacrificing Quartus II software functionality.

Chapter Overview

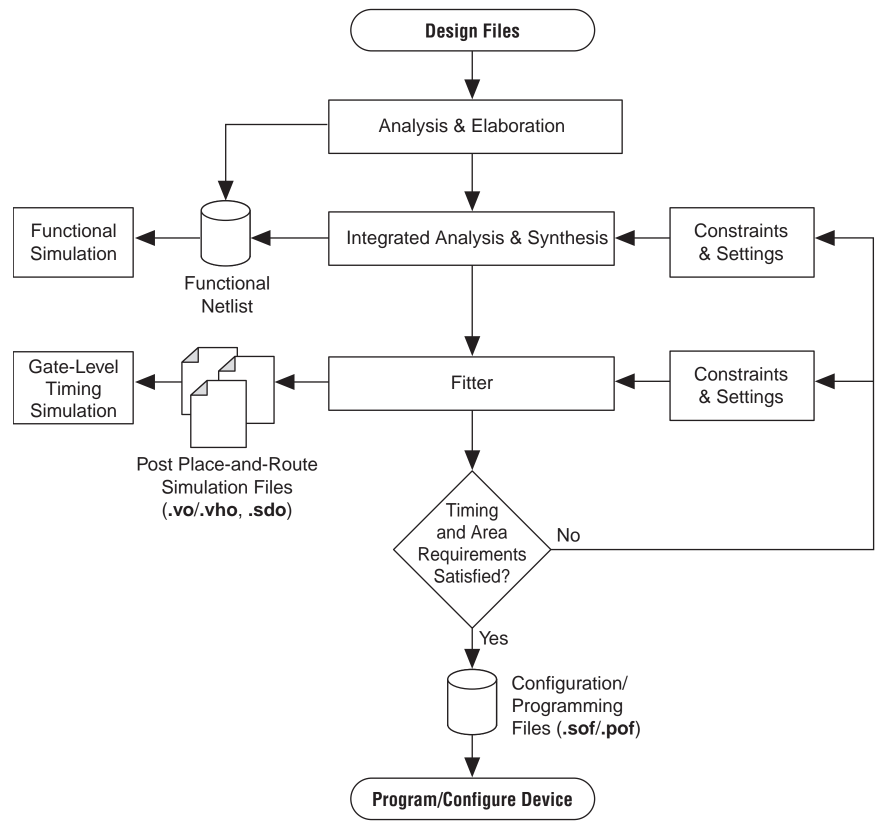
This chapter covers the following topics:

- "Typical Design Flow" on page 3–2
- "Device Support" on page 3–2
- "Quartus II GUI Overview" on page 3–3
- "Setting Up MAX+PLUS II Look and Feel in the Quartus II Software" on page 3–5
- "Compiler Tool" on page 3–7
- "MAX+PLUS II Design Conversion" on page 3–9
- "Quartus II Design Flow" on page 3–12

Typical Design Flow

Figure 3–1 shows a typical design flow with the Quartus II software.

Figure 3–1. Quartus II Software Design Flow



Device Support

The Quartus II software supports most of the devices supported in the MAX+PLUS II software, but it does not support any obsolete devices or packages. The devices supported by these two software packages are shown in Table 3–1.

Table 3–1. Device Support Comparison (Part 1 of 2)

Device Supported	Quartus II	MAX+PLUS II
Arria® GX	✓	—
Stratix® Series	✓	—
Cyclone® Series	✓	—
HardCopy® Series	✓	—
MAX® II	✓	—
Classic™	—	✓
MAX 3000A	✓	✓
MAX 7000S/AE/B	✓	✓
MAX 7000E	—	✓

Table 3–1. Device Support Comparison (Part 2 of 2)

Device Supported	Quartus II	MAX+PLUS II
MAX 9000	—	✓
ACEX® 1K	✓	✓
FLEX® 6000	✓	✓
FLEX 8000	—	✓
FLEX 10K	✓ *(1)*	✓
FLEX 10KA	✓	✓
FLEX 10KE	✓ *(2)*	✓
APEX™ II	✓	—
APEX 20K	✓	—

Notes to Table 3–1:

(1) PGA packages (represented as package type G in the ordering code) are not supported in the Quartus II software.

(2) Some packages are not supported.

Quartus II GUI Overview

The Quartus II software provides the following utility windows to assist in the development of your designs:

- Project Navigator
- Node Finder
- Tcl Console
- Messages
- Status
- Change Manager

Project Navigator

The **Hierarchy** tab of the Project Navigator window is similar to the MAX+PLUS II Hierarchy Display and provides additional information such as logic cell, register, and memory bit resource utilization. The **Files** and **Design Units** tabs of the Project Navigator window provide a list of project files and design units.

Node Finder

The Node Finder window provides the equivalent functionality of the MAX+PLUS II **Search Node Database** dialog box and allows you to find and use any node name stored in the project database.

Tcl Console

The Tcl Console window allows access to the Quartus II Tcl shell from within the GUI. You can use the Tcl Console window to enter Tcl commands and source Tcl scripts to make assignments, perform customized timing analysis, view information about devices, or fully automate and customize the way you run all components of the Quartus II software. There is no equivalent functionality in the MAX+PLUS II software.

For more information about using Tcl with the Quartus II software, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Messages

The Messages window is similar to the Message Processor window in the MAX+PLUS II software, providing detailed information, warnings, and error messages.You also can use it to locate a node from a message to various windows in the Quartus II software.

Status

The Status window displays information similar to the MAX+PLUS II Compiler window. Progress and elapsed time are shown for each stage of the compilation.

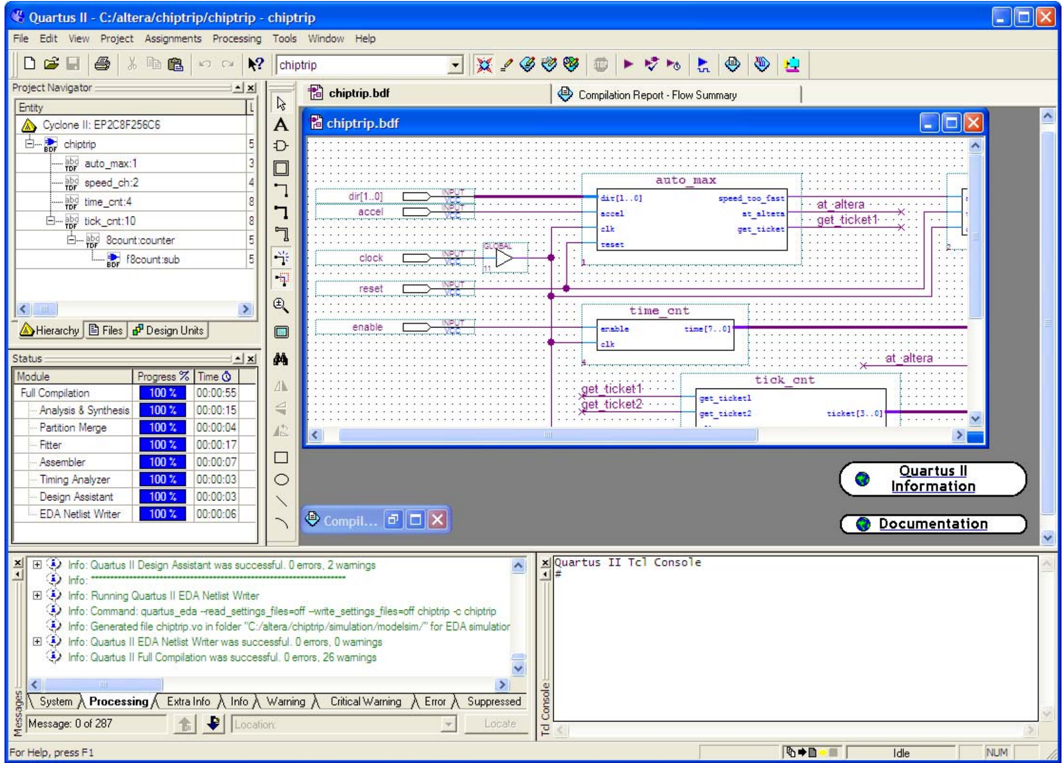
Change Manager

The Change Manager provides detailed tracking information about all design changes made with the Chip Planner.

For more information about the Engineering Change Manager and the Chip Editor, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Figure 3–2 shows a typical Quartus II software display.

Figure 3–2. Quartus II Look and Feel



Setting Up MAX+PLUS II Look and Feel in the Quartus II Software

You can choose the MAX+PLUS II look and feel by selecting MAX+PLUS II in the **Look & Feel** box of the **General** tab of the **Customize** dialog box on the Tools menu.

Any changes to the look and feel do not become effective until you restart the Quartus II software.

By default, when you select the MAX+PLUS II look and feel, the **MAX+PLUS II** quick menu appears on the left side of the menu bar. You can turn the Quartus II and MAX+PLUS II quick menus on or off. You also can change the preferred positions of the two quick menus. To change these options, perform the following steps:

1. On the Tools menu, click **Customize**. The **Customize** dialog box appears.
2. Click the **General** tab.
3. Under **Quick menus**, select your preferred options.

MAX+PLUS II Look and Feel

The MAX+PLUS II look and feel in the Quartus II software closely resembles the MAX+PLUS II software. Figure 3–3 and 3–6 compare the MAX+PLUS II software appearance with the Quartus II MAX+PLUS II look and feel.

Figure 3–3. MAX+PLUS II Software GUI

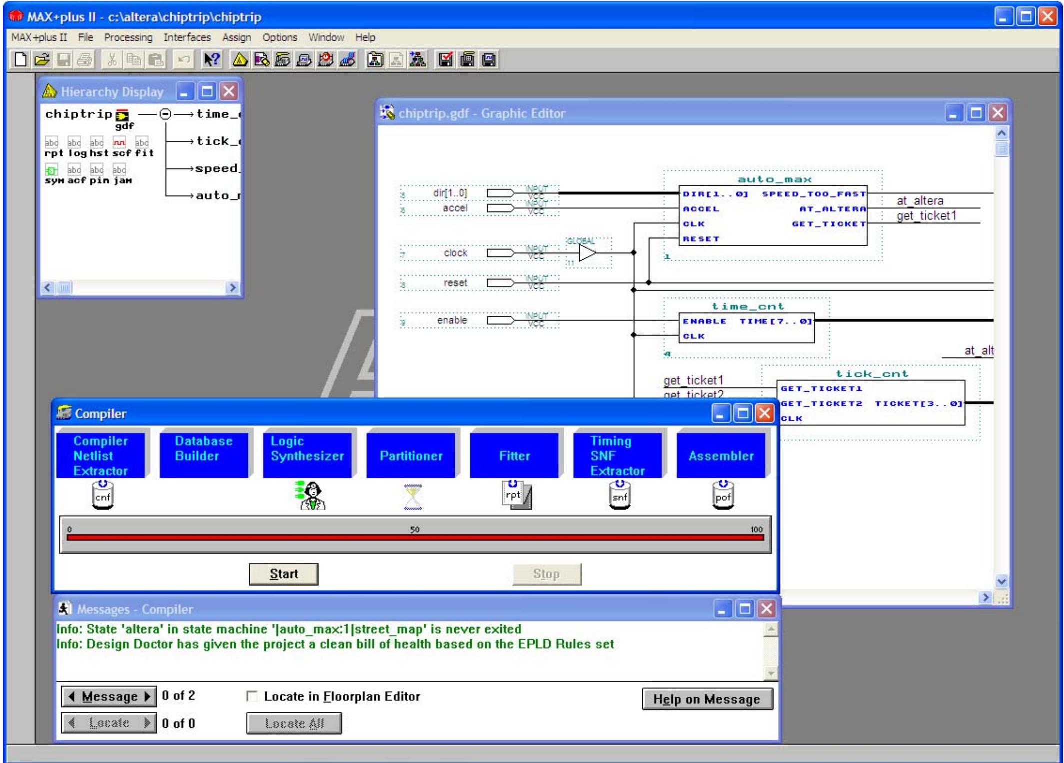
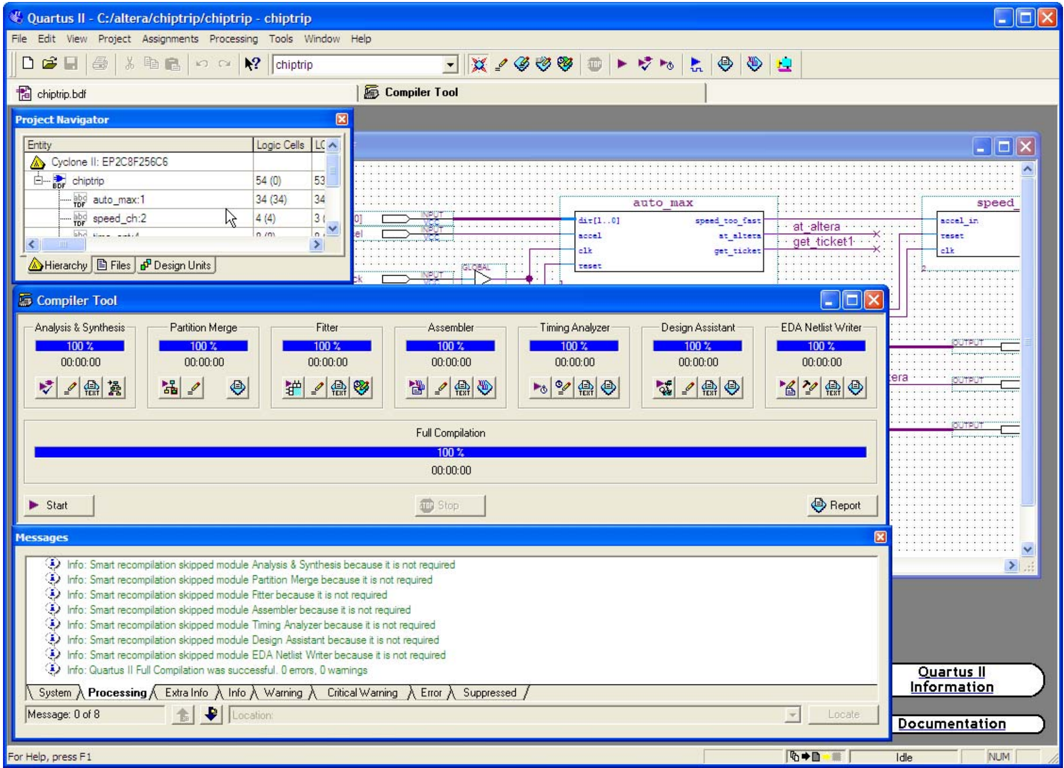


Figure 3–4. Quartus II Software with MAX+PLUS II Look and Feel



The standard MAX+PLUS II toolbar is also available in the Quartus II software with the MAX+PLUS II look and feel in the Quartus II software (Figure 3–5).

Figure 3–5. Standard MAX+PLUS II Toolbar

Compiler Tool

The Quartus II Compiler Tool provides an intuitive MAX+PLUS II style interface. You can edit the settings and view result files for the following modules:

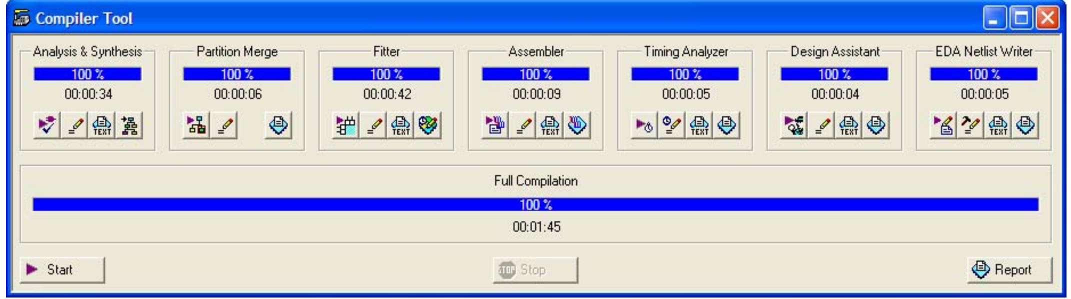
- Analysis and Synthesis
- Partition Merge
- Fitter
- Assembler
- Timing Analyzer
- EDA Netlist Writer
- Design Assistant

Each of these modules is described later in this section.

To start a compilation using the Compiler Tool, click **Compiler Tool** from either the MAX+PLUS II menu or the Tools menu and click **Start** in the Compiler Tool. The Compiler Tool, shown in Figure 3–6, displays all modules, including optional modules such as Partition Merge, Assembler, EDA Netlist Writer, and the Design Assistant.

For information about using the Quartus II software modules at the command line, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Figure 3–6. Running a Full Compilation with the Compiler Tool



Analysis and Synthesis

The Quartus II Analysis and Synthesis module analyzes your design, builds the design database, optimizes the design for the targeted architecture, and maps the technology to the design logic.

In MAX+PLUS II software, these functions are performed by the Compiler Netlist Extractor, Database Builder, and Logic Synthesizer. There is no module in the Quartus II software similar to the MAX+PLUS II Partitioner module.

Partition Merge

The optional Quartus II Partition Merge module merges the partitions to create a flattened netlist for further stages of the Quartus II compilation flow. The Partition Merge module is not similar to the MAX+PLUS II Partitioner. This tool is available only if you turn on incremental compilation. You can turn on incremental compilation by performing the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the **+** icon to expand **Compilation Process Settings**, and select **Incremental Compilation**. The **Full Incremental Compilation** page appears.
3. Under **Incremental compilation**, turn on **Incremental Compilation**.

Fitter

The Quartus II Fitter module uses the PowerFit Fitter to fit your design into the available resources of the targeted device. The Fitter places and routes the design. The Fitter module is similar to the Fitter stage of the MAX+PLUS II software.

Assembler

The optional Quartus II Assembler module creates a device programming image of your design so that you can configure your device. You can select from the following types of programming images:

- Programmer Object File (.**pof**)
- SRAM Output File (.**sof)**
- Hexadecimal (Intel-Format) Output File (.**hexout**)
- Tabular Text File (.**ttf)**
- Raw Binary File (.**rbf**)
- Jam™ STAPL Byte Code 2.0 File (.**jbc)**
- JEDEC STAPL Format File (.**jam**)

You can turn off the Assembler module during compilation by turning off **Run assembler** in the **Compilation Process Settings** page in the **Settings** dialog box. You also can turn off the Assembler by right-clicking in the Compiler Tool window. The Assembler module is similar to the Assembler stage of the MAX+PLUS II software.

Timing Analyzer

The Quartus II Timing Analyzer allows you to analyze more complex clocking schemes than is possible with the MAX+PLUS II Timing Analyzer. The Quartus II Timing Analyzer analyzes all clock domains in your design, including paths that cross clock domains, and also reports both f_{MAX} and slack. Slack is the margin by which the timing requirement is met or is not met. For more information on the Timing Analyzer, refer to "Timing Analysis" on page 3–20.

EDA Netlist Writer

The optional Quartus II EDA Netlist Writer module generates a netlist for simulation with an EDA simulation tool. The EDA Netlist Writer module is comparable to the VHDL and Verilog Netlist Writer in the MAX+PLUS II software.

Design Assistant

The optional Quartus II Design Assistant module checks the reliability of your design based on a set of design rules. The Design Assistant analyzes and generates messages for a design targeting any Altera device and is especially useful for checking the reliability of a design to be converted to HardCopy series devices. The Design Assistant is similar to the Design Doctor in the MAX+PLUS II software.

In the Quartus II software, you can reduce subsequent compilation time significantly by turning **Use smart compilation** on before compiling your design. The Smart Compilation feature skips any compilation stages which are not required and which may use more disk space. This Quartus II smart compilation option is similar to the MAX+PLUS II **Smart Recompile** command. To turn the **Use smart compilation** option on, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Compilation Process Settings**. The **Compilation Process Settings** page appears.
3. Turn on **Use smart compilation**.

MAX+PLUS II Design Conversion

With the Quartus II software, you can open MAX+PLUS II designs and convert MAX+PLUS II assignments and files.

The Quartus II software is project based. All the files for your design (HDL input, simulation vectors, assignments, and other relevant files) are associated with a project file. For more information about creating a new project, refer to "Creating a New Project" on page 3–12.

Converting an Existing MAX+PLUS II Design

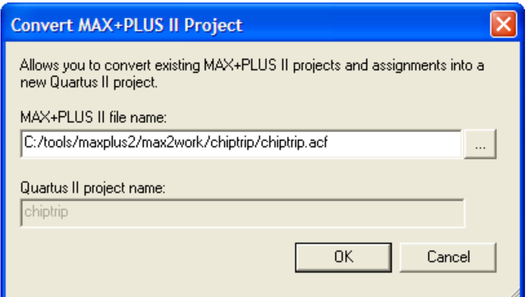
You can easily convert an existing MAX+PLUS II design for use with the Quartus II software with the **Convert MAX+PLUS II Project** command in the Quartus II software or the **Open Project** command. You can find these commands on the File menu.

If you use the **Convert MAX+PLUS II Project** command, browse to the MAX+PLUS II Assignments and Configuration File (**.acf**) or top-level design file (Figure 3–7) and click **Open**. The **Convert MAX+PLUS II Project** command generates a Quartus II Project File (**.qpf**) and a Quartus II Settings File (**.qsf**). The Quartus II software stores project and design assignments in the **.qsf** file, which is equivalent to the Assignments and Configuration File in the MAX+PLUS II software.

You also can open and convert a MAX+PLUS II design with the **Open Project** command. In the **Open Project** dialog box, browse to the Assignments and Configuration File or the top-level design file. Click **Open** to display the **Convert MAX+PLUS II Project** dialog box.

The Quartus II software can import all MAX+PLUS II-generated files, but it cannot save files in the MAX+PLUS II format. You cannot open a Quartus II project in the MAX+PLUS II software, nor can you convert a Quartus II project to a MAX+PLUS II project.

Figure 3–7. Convert MAX+PLUS II Project Dialog Box



 © March 2009 Altera Corporation

The conversion process performs the following actions:

- Converts the MAX+PLUS II Assignments and Configuration File into a **.qsf** file (equivalent to importing all MAX+PLUS II assignments)
- Creates a **.qpf** file
- Displays all errors and warnings in the Quartus II message window

The Quartus II software can read MAX+PLUS II generated Graphic Design Files (**.gdf**) and Simulation Channel Files (**.scf**) without converting them. These files are not modified during a MAX+PLUS II design conversion.

Converting MAX+PLUS II Graphic Design Files

The Quartus II Block Editor (similar to the MAX+PLUS II Graphic Editor) saves files as Block Design Files (**.bdf**). You can convert your MAX+PLUS II Graphic Design File into a Quartus II Block Design File using one of the following methods:

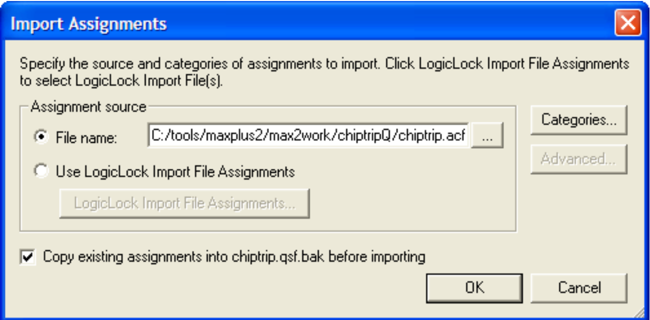
1. Open the Graphic Design File and on the File menu, click **Save As**. The **Save As** dialog box appears.
2. In the **Save as type** list, select **Block Diagram/Schematic File** (*.**bdf**).
3. Run the **quartus_g2b.exe** command line executable located in the \<*Quartus II installation*>**bin** directory. For example, to convert the **chiptrip.gdf** file to a Block Design File, type the following command at a command prompt:

```
quartus_g2b.exe chip_trip.gdf ↵
```

Importing MAX+PLUS II Assignments

You can import MAX+PLUS II assignments into an existing Quartus II project. Open the project, and on the Assignments menu, click **Import Assignments**. Browse to the Assignments and Configuration File (Figure 3–8). You can also import **.qsf** files and Entity Setting Files (**.esf**).

Figure 3–8. Import Assignments Dialog Box



The Quartus II software accepts most MAX+PLUS II assignments. However, some assignments can be imported incorrectly from the MAX+ PLUS II software into the Quartus II software due to differences in node naming conventions and the advanced Quartus II integrated synthesis algorithms.

The differing node naming conventions in the Quartus II and MAX+PLUS II software can cause improper mapping when importing your design from MAX+PLUS II software into the Quartus II software. Improper node names can interfere with the design logic if you are unaware of these node name differences and do not take appropriate steps to prevent improper node name mapping. Table 3–2 compares the differences between the naming conventions used by the Quartus II and MAX+PLUS II software.

Table 3–2. Quartus II and MAX+PLUS II Node and Pin Naming Conventions

Feature	Quartus II Format	MAX+PLUS II Format
Node name	auto_max:auto|q0	|auto_max:auto|q0
Pin name	d[0], d[1], d[2]	d0, d1, d2

When you import MAX+PLUS II assignments containing node names that use numbers, such as `signal0` or `signal1`, the Quartus II software imports the original assignment and also creates an additional copy of the assignment. The additional assignment has square brackets inserted around the number, resulting in `signal[0]` or `signal[1]`. The square bracket format is legal for signals that are part of a bus, but creates illegal signal names for signals that are not part of a bus in the Quartus II software. If your MAX+PLUS II design contains node names that end in a number and are not part of a bus, you can edit the **.qsf** file to remove the square brackets from the node names after importing them.

You can remove obsolete assignments in the **Remove Assignments** dialog box. Open this dialog box on the Assignments menu by clicking **Remove Assignments**.

The Quartus II software may not recognize valid MAX+PLUS II node names, or may split MAX+PLUS II nodes into two different nodes. As a result, any assignments made to synthesized nodes are not recognized during compilation.

For more information about Quartus II node naming conventions, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Quartus II Design Flow

The following sections include information to help you get started using the Quartus II software. They describe the similarities and differences between the Quartus II software and the MAX+PLUS II software. The following sections highlight improvements and benefits in the Quartus II software.

For an overview of the Quartus II software features and design flow, refer to the *Introduction to the Quartus II Software* manual.

Creating a New Project

The Quartus II software provides a wizard to help you create new projects. On the File menu, click **New Project Wizard** to start the New Project Wizard. The New Project Wizard generates the **.qpf** file and **.qsf** file for your project.

Design Entry

The Quartus II software supports the following design entry methods:

- Altera HDL (AHDL) Text Design File (.**tdf**)
- Block Diagram File
- EDIF Netlist File (.**edf**)
- Verilog Quartus Mapping Netlist File (.**vqm**)
- VHDL (.**vhd**)
- Verilog HDL (.**v**)

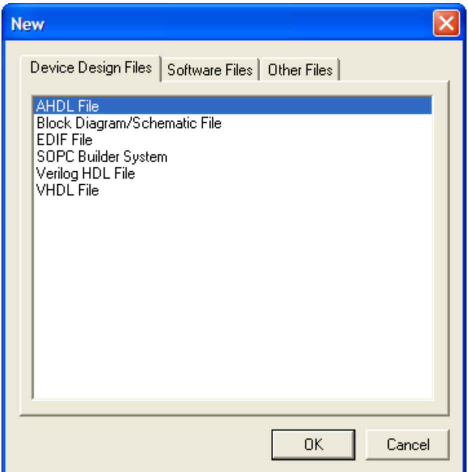
The Quartus II software has an advanced integrated synthesis engine that fully supports the Verilog HDL and VHDL languages and provides options to control the synthesis process.

For more information, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

To create a new design file, perform the following steps:

1. On the File menu, click **New**. The **New** dialog box appears.
2. Click the **Device Design Files** tab.
3. Select a design entry type.
4. Click **OK** (see Figure 3–9).

Figure 3–9. New Dialog Box



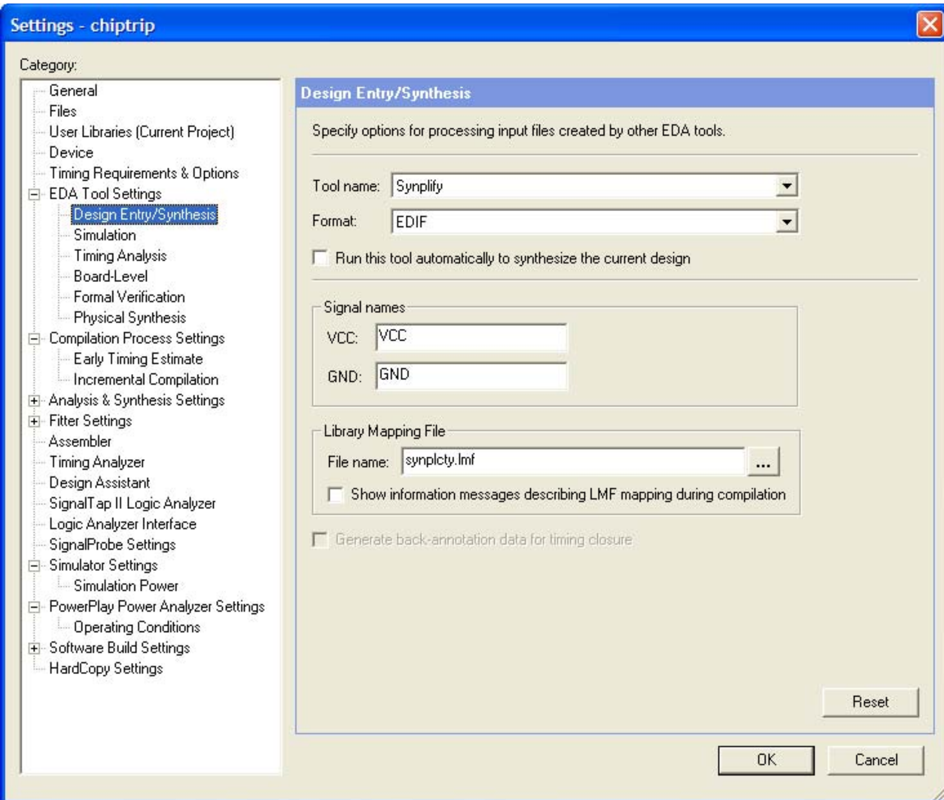
You can create other files from the **Software Files** tab and **Other Files** tab of the New dialog box on the File menu. For example, the Vector Waveform File (**.vwf**) is located in the **Other Files** tab.

To analyze a netlist file created by an EDA tool, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.

2. In the **Category** list, select **Design Entry & Synthesis**. The **Design Entry & Synthesis** page appears.
3. In the **Tool name** list, select the synthesis tool used to generate the netlist (Figure 3–10).

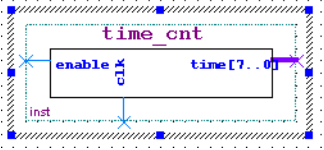
Figure 3–10. Settings Dialog Box Specifying Design Entry Tool



The Quartus II Block Editor has many advantages over the MAX+PLUS II Graphic Editor. The Block Editor offers an unlimited sheet size, multiple region selections, an enhanced Symbol Editor, and conduits.

The Symbol Editor allows you to change the positions of the ports in a symbol (refer to Figure 3–11). You can reduce wire congestion around a symbol by changing the positions of the ports.

Figure 3–11. Various Port Positions for a Symbol



To make changes to a symbol in a Block Design File, right-click a symbol in the Block Editor and select **Properties** to display the **Symbol Properties** dialog box. This dialog box allows you to change the instance name, add parameters, and specify the line and text color.

You can use conduits to connect blocks (including pins) in the Block Editor. Conduits contain signals for the connected objects (see Figure 3–12). You can determine the connections between various blocks in the **Conduit Properties** dialog box by right-clicking a conduit and clicking **Properties**.

Figure 3–12. Blocks and Pins Connected with Conduits

Making Assignments

The Quartus II software stores all project and design assignments in a **.qsf** file, which is a collection of assignments stored as Tcl commands and organized by the compilation stage and assignment type. The **.qsf** file stores all assignments, regardless of how they are made, from the Floorplan Editor, the Pin Planner, the Assignment Editor, with Tcl, or any other method.

Assignment Editor

The Assignment Editor is an intuitive spreadsheet interface designed to allow you to make, change, and manage a large number of assignments easily. With the Assignment Editor, you can list all available pin numbers and design pin names for efficiently creating pin assignments. You also can filter all assignments based on assignment categories and node names for viewing and creating assignments.

The Assignment Editor is composed of the **Category** bar, **Node Filter** bar, **Information** bar, **Edit** bar, and spreadsheet.

To make an assignment, follow these steps:

1. On the Assignments menu, click **Assignment Editor**. The Assignment Editor window appears.
2. Select an assignment category in the **Category** bar.
3. Select a node name using the Node Finder or type a node name filter into the **Node Filter** bar. (This step is optional; it excludes all assignments unrelated to the node name.)
4. Type the required values into the spreadsheet.
5. On the File menu, click **Save**.

If you are unsure about the purpose of a cell in the spreadsheet, select the cell and read the description displayed in the **Information** bar.

You can use the **Edit** bar to change the contents of multiple selected cells simultaneously. Select cells in the spreadsheet and type the value in the **Edit** box.

Other advantages of the Assignment Editor include clipboard support in the spreadsheet and automatic font coloring to identify the status of assignments.

For more information, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook.*

Timing Assignments

You can use the timing wizard to help you set your timing requirements. On the Assignments menu, click **Timing Wizard** to create global clock and timing settings. The settings include f_{MAX}, setup times, hold times, clock to output delay times, and individual absolute or derived clocks.

You also can set timing settings manually by performing the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Timing Requirements & Options**. The **Timing Requirements & Options** page appears.
3. Set your timing settings.

You can make more complex timing assignments with the Quartus II software than allowed by the MAX+PLUS II software, including multicycle and point-to-point assignments using wildcards and time groups.

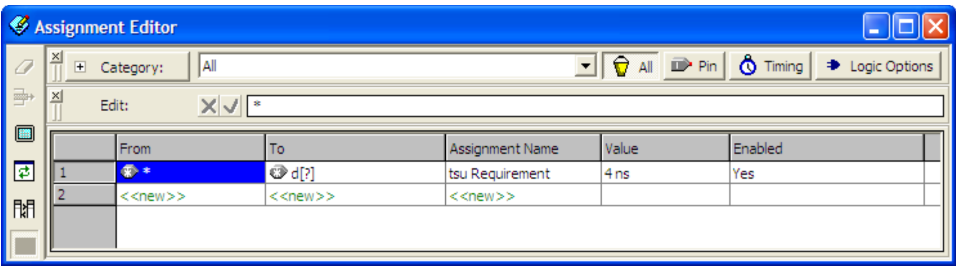
A time group is a collection of design nodes grouped together and represented as a single unit for the purpose of making timing assignments to the collection.

Multicycle timing assignments allow you to identify register-to-register paths in the design where you expect a delayed latch edge. This assignment enables accurate timing analysis of your design.

Point-to-point timing assignments allow you to specify the required delay between two pins, two registers, or a pin and a register. This assignment helps you optimize and verify your design timing requirements.

Wildcard characters "?" and " * " allow you to apply an assignment to a large number of nodes with just a few assignments. For example, Figure 3–13 shows a 4 ns t_{SU} requirement assignment to all paths from any node to the "d" bus in the Assignment Editor.

Figure 3–13. Single t_{SU} Timing Assignment Applied to All Nodes of a Bus



For more information, refer to the *Quartus II Classic Timing Analyzer* or the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook.*

Synthesis

The Quartus II advanced integrated synthesis software fully supports the hardware description languages, Verilog HDL, VHDL, and AHDL, schematic entry, and also provides options to control the synthesis process. With this synthesis support, the Quartus II software provides a complete, easy-to-use, stand-alone solution for today's designs.

You can specify synthesis options in the **Analysis & Synthesis Settings** page of the **Settings** dialog box. Similar to MAX+PLUS II synthesis options, you select one of these optimization techniques: **Speed**, **Area**, or **Balanced**.

To achieve higher design performance, you can turn on synthesis netlist optimizations that are available when targeting certain devices. You can unmap a netlist created by an EDA tool and remap the components in the netlist back to Altera primitives by turning on **Perform WYSIWYG primitive resynthesis**. Additionally, you can move registers across combinational logic to balance timing without changing design functionality by turning on **Perform gate-level register retiming**. Both of these options are accessible from the **Synthesis Netlist Optimizations** page under **Analysis & Synthesis Settings** in the **Settings** dialog box on the Assignments menu.

For more information, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook.*

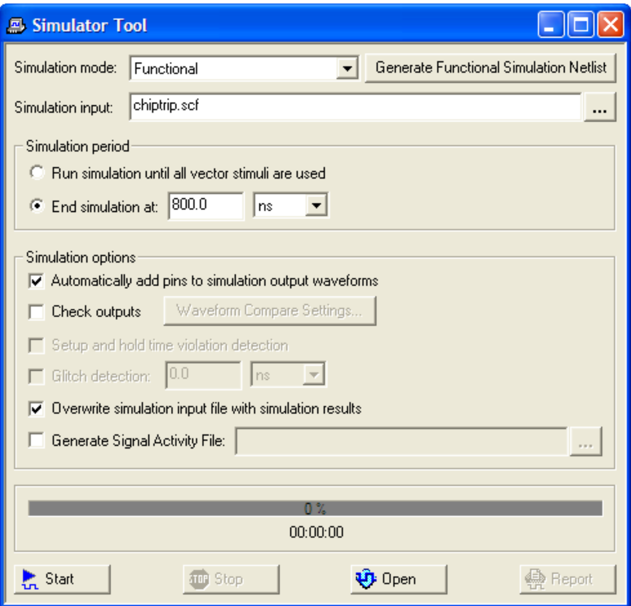
Functional Simulation

Similar to the MAX+PLUS II Simulator, the Quartus II Simulator Tool performs both functional and timing simulations.

To open the Simulator Tool, on MAX+PLUS II menu, click **Simulator**, or on the Tools menu, click **Simulator Tool**. Before you perform a functional simulation, an internal functional simulation netlist is required. Click **Generate Functional Simulation Netlist** in the **Simulator Tool** dialog box (Figure 3–14), or on the Processing menu, click **Generate Functional Simulation Netlist**.

Generating a functional simulation netlist creates a separate database that improves the performance of the simulation significantly.

Figure 3–14. Simulator Tool Dialog Box



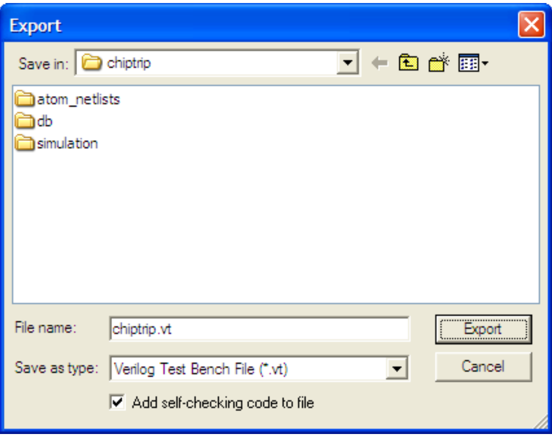
You can view and modify the simulator options on the **Simulator Settings** page of the **Settings** dialog box or in the **Simulator Tool** dialog box. You can set the simulation period and turn **Check outputs** on or off. You can choose to display the simulation outputs in the simulation report or in the Vector Waveform File. To display the simulation results in the simulation input vector waveform file, which is the MAX+PLUS II behavior, turn on **Overwrite simulation input file with simulation results**.

When using either the MAX+PLUS II or Quartus II software, you may have to compile additional behavioral models to perform a simulation with an EDA simulation tool. In the Quartus II software, behavioral models for library of parameterized modules (LPM) functions and Altera-specific megafunctions are available in the **altera_mf** and **220model** library files, respectively. The **220model** and **altera_mf** files can be found in the \<*Quartus II Installation*>**eda\sim_lib directory**.

The Quartus II schematic design files (Block Design File, or **.bdf**) are not compatible with EDA simulation tools. To perform a register transfer level (RTL) functional simulation of a Block Design File using an EDA tool, convert your schematic designs to a VHDL or Verilog HDL design file. Open the schematic design file and on the File menu, point to **Create/Update** and then click **Create HDL Design File for Current File** to create an HDL design file that corresponds to your Block Design File.

You can export a **.vwf** file or Simulator Channel File (**.scf**) as a Verilog HDL or VHDL testbench file for simulation with an EDA tool. Open your Vector Waveform File or **.scf** file and on the File menu, click **Export**. See Figure 3–15. Select **Verilog** or **VHDL Test Bench File (*.vt)** from the **Save as type** list. Turn on **Add self-checking code to file** to add additional self-checking code to the testbench.

Figure 3–15. Export Dialog Box



Place and Route

The Quartus II PowerFit is an incremental fitter that performs place-and-route to fit your design into the targeted device. You can control the Fitter behavior with options in the **Fitter Settings** page of the **Settings** dialog box on the Assignments menu.

High-density device families supported in the Quartus II software, such as the Stratix series, sometimes require significant fitter effort to achieve an optimal fit. The Quartus II software offers several options to reduce the time required to fit a design. You can control the effort the Quartus II Fitter expends to achieve your timing requirements with these options:

- **Optimize timing** performs timing-based placement using the timing requirements you specify for the design. You can use this option by itself or with one or more of the options below.
- **Optimize hold timing** optimizes the hold times within a device to meet timing requirements and assignments you specify. You can select this option only if the Optimize timing option is also chosen.
- **Optimize fast-corner timing** instructs the Fitter, when optimizing your design, to consider fast-corner delays, in addition to slow-corner delays, from the fast-corner timing model (fastest manufactured device, operating in low-temperature and high-voltage conditions). You can select this option only if the **Optimize timing** option is also chosen.

If minimizing compilation time is more important than achieving specific timing results, you can turn these options off.

Another way to decrease the processing time and effort the Fitter expends to fit your design is to select either **Standard Fit** or **Fast Fit** in the **Fitter Effort** box of the **Fitter Settings** page in the **Settings** dialog box on the Assignments menu. The option you select affects the Fitter behavior and your design as described below.

- Select **Standard Fit** for the Fitter to use the highest effort and preserve the performance from previous compilations.
- Select **Fast Fit** for up to 50% faster compilation times, although this may reduce design performance.

You can also select **Auto Fit** to decrease compilation time by directing the Fitter to reduce Fitter effort after meeting your timing requirements. The **Auto Fit** option is available for select devices.

For more information, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

To further reduce compilation times, turn on **Limit to one fitting attempt** in the **Fitter Settings** page in the **Settings** dialog box on the Assignments menu.

If your design is very close to meeting your timing requirements, you can control the seed number used in the fitting algorithm by changing the value in the **Seed** box of the **Fitter Settings** page of the **Settings** dialog box on the Assignments menu. The default seed value is 1. You can specify any non-negative integer value. Changing the value of the seed only repositions the starting location of the Fitter, but does not affect compilation time or the Fitter effort level. However, if your design is difficult to fit optimally or takes a long time to fit, sometimes you can improve results or processing time by changing the seed value.

Timing Analysis

Version 6.1 and later of the Quartus II software supports two native timing analysis tools: the TimeQuest Timing Analyzer and the Classic Timing Analyzer. Both timing analysis tools provide more complex clocking schemes than is possible with the MAX+PLUS II Timing Analyzer. The TimeQuest Timing Analyzer uses the industry-standard Synopsys Design Constraint (SDC) methodology for constraining designs and reporting results. In general, the TimeQuest Timing Analyzer provides more control in constraining a design as compared to the Classic Timing Analyzer. However, the Classic Timing Analyzer incorporates a basic graphical user interface and the timing analysis flow is similar to the flow in the MAX+PLUS II software. As such, the section that follows provides a more detailed look at timing analysis using the Classic Timing Analyzer.

For more information about choosing between the TimeQuest Timing Analyzer or the Classic Timing Analyzer, refer to the Timing Analysis Section in the *Introduction to the Quartus II Software* manual.

Launch the Classic Timing Analyzer tool on the MAX+PLUS II menu by clicking **Classic Timing Analyzer** or by selecting **Classic Timing Analyzer Tool** on the **Processing** menu. See Figure 3–16. To start the analysis, click **Start** in the Timing Analyzer Tool, or on the Processing menu, point to **Start** and clicking **Start Timing Analyzer**.

Figure 3–16. Registered Performance Tab of the Timing Analyzer Tool

The Quartus II Classic Timing Analyzer analyzes all clock domains in your design, including paths that cross clock domains. You can ignore paths that cross clock domains by using the following options in the **Timing Requirements & Options** page in the **Settings** dialog box on the Assignments menu:

- Create a **Cut Timing Path** assignment
- Turn on **Cut paths between unrelated clock domains**

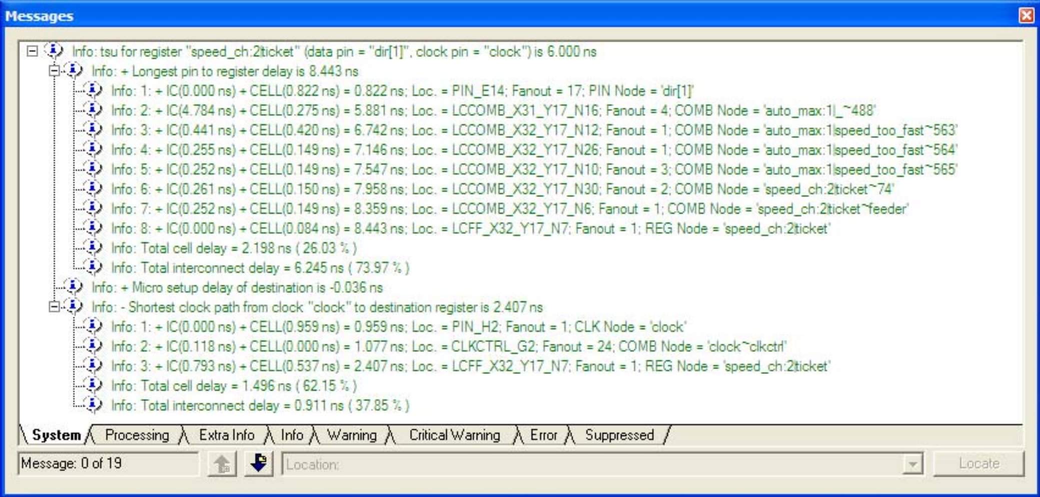
To view the results from the Classic Timing Analyzer Tool, click the **Report** button located at the bottom of the **Classic Timing Analyzer** dialog box, or to get specific information, click on any of the following tabs at the top of the Classic Timing Analyzer window:

- **Registered Performance**
- **tpd**
- **tsu**
- **tco**
- **th**
- **Custom Delays**

The Quartus II Classic Timing Analyzer reports both f_{MAX} and slack. Slack is the margin by which the timing requirement was met or not met. A positive slack value, displayed in black, indicates the margin by which a requirement was met. A negative slack value, displayed in red, indicates the margin by which a requirement was not met.

To analyze a particular path in more detail, select a path in the Classic Timing Analyzer Tool and click **List Paths**. This displays a detailed description of the path in the **System** tab of the Messages window (Figure 3–17).

Figure 3–17. Messages Window Displaying Detailed Timing Information



For more information, refer to the *Quartus II Classic Timing Analyzer* or the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook.*

Timing Closure Floorplan

The Quartus II Timing Closure Floorplan is similar to the MAX+PLUS II Floorplan Editor but has many improvements to help you more effectively view and debug your design. With its ability to display logic cell usage, routing congestion, critical paths, and LogicLock™ regions, the Timing Closure Floorplan also makes the task of improving your design performance much easier.

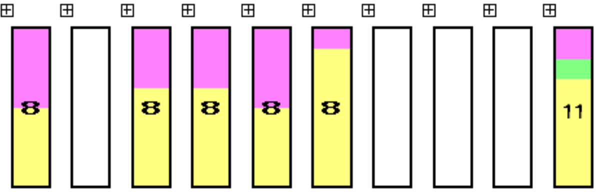
To view the Timing Closure Floorplan, on the MAX+PLUS II menu, click **Floorplan Editor** or **Timing Closure Floorplan**.

The Timing Closure Floorplan Editor provides Interior Cell views equivalent to the MAX+PLUS II logic array block (LAB) views. In addition to these views, available from the View menu, you also can select from the Interior MegaLABs (where applicable), Interior LABs, and Field views.

The Pin Planner is equivalent to the MAX+PLUS II Device view. The Pin Planner can be launched from the View menu or on the Assignments menu by clicking **Pin Planner**.

The Interior LABs view hides cell details for logic cells, Adaptive Logic Modules (ALM), and macrocells, and shows LAB information (see Figure 3–18). You can display the number of cells used in each LAB on the View menu by clicking **Show Usage Numbers**.

Figure 3–18. Interior LAB View of the Timing Closure Floorplan



The Field view is a color-coded, high-level view of your device resources that hides both cell and LAB details. In the Field view, you can see critical paths and routing congestion in your design.

The View Critical Paths feature shows a percentage of all critical paths in your floorplan. You can enable this feature on the View menu by clicking **Show Critical Paths**. You can control the number of critical paths shown by modifying the settings in the **Critical Paths Settings** dialog box on the View menu.

The View Congestion feature displays routing congestion by coloring and shading logic resources. Darker shading shows greater resource utilization. This feature assists in identifying locations where there is a lack of routing resources.

To show lower level details in any view, right-click on a resource and click **Show Details**.

For more information, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook.*

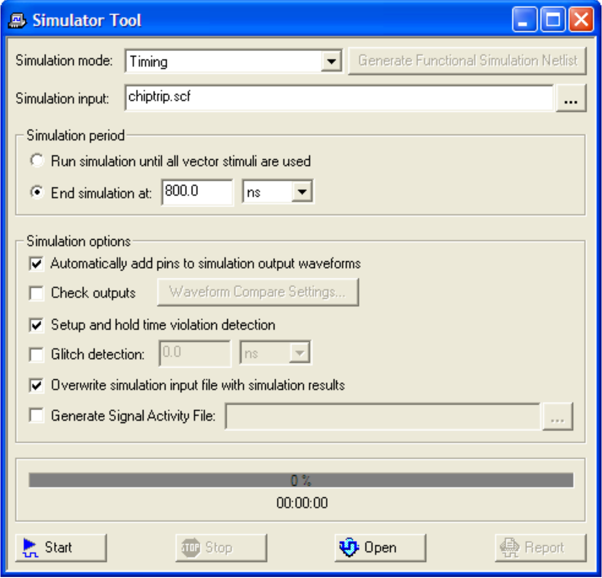
Timing Simulation

Timing simulation is an important part of the verification process. The Quartus II software supports native timing simulation and exports simulation netlists to third-party software for design verification.

Quartus II Simulator Tool

The Quartus II Simulator tool is an easy-to-use integrated solution that uses the compiler database to simulate the logical and timing performance of your design (Figure 3–19). When performing timing simulation, the simulator uses place-and-route timing information.

Figure 3–19. Quartus II Simulator Tool



You can use Vector Table Output Files (**.tbl**), Vector Waveform Files, Vector Files (**.vec**), or an existing **.scf** file as the vector stimuli for your simulation.

The simulation options available are similar to the options available in the MAX+PLUS II Simulator. You can control the length of the simulation and the type of checks performed by the Simulator. When the MAX+PLUS II look and feel is selected, the **Overwrite simulation input file with simulation results** option is on by default. If you turn it off, the simulation results are written to the report file. To view the report file, click **Report** in the Simulator Tool window.

EDA Timing Simulation

The Quartus II software also supports timing simulation with other EDA simulation software. Performing timing simulation with other EDA simulation software requires a Quartus II generated timing netlist file in the form of a Verilog Output File (**.vo**) or VHDL Output File (**.vho**), a Standard Delay Format Output File (**.sdo**), and a device-specific atom file (or files), shown in Table 3–3.

Table 3–3. Altera Timing Simulation Library Files

Verilog	VHDL
*<device_family>*_**atoms.v**	*<device_family>*_**atoms_87.vhd**
	*<device_family>*_**atoms.vhd**
	*<device_family>*_**components.vhd**

Specify your EDA simulation tool by performing the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears.
3. In the **Tool name list**, select your EDA Tool.

You can generate a timing netlist for the selected EDA simulator tool by running a full compile or on the Processing menu, by pointing to **Start** and clicking **Start EDA Netlist Writer**. The generated netlist and SDF file are placed into the ****<*project directory*>**\simulation**<*EDA simulator tool*> directory. The device-specific atom files are located in the ****<*Quartus II Install*>**\eda\sim_lib** directory.

Power Estimation

To develop an appropriate power budget and to design the power supplies, voltage regulators, heat sink, and cooling system, you need an accurate estimate of the power that your design consumes. You can estimate power by using the PowerPlay Early Power Estimation spreadsheet available on the Altera website at www.altera.com, or with the PowerPlay Power Analyzer in the Quartus II software.

You can perform early power estimation with the PowerPlay Early Power Estimation spreadsheet by entering device resource and performance information. The Quartus II PowerPlay Analyzer tool performs vector-based power analysis by reading either a Signal Activity File (**.saf**), generated from a Quartus II simulation, or a Verilog Value Change Dump File (**.vcd**) generated from a third-party simulation.

For more information about how to use the PowerPlay Power Analyzer tool, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook.*

Programming

The Quartus II Programmer has the same functionality as the MAX+PLUS II Programmer, including programming, verifying, examining, and blank checking operations. Additionally, the Quartus II Programmer now supports the erase capability for CPLDs. To improve usability, the Quartus II Programmer displays all programming-related information in one window (Figure 3–20).

Click **Add File** or **Add Device** in the Programmer window to add a file or device, respectively.

Figure 3–20. Programmer Window

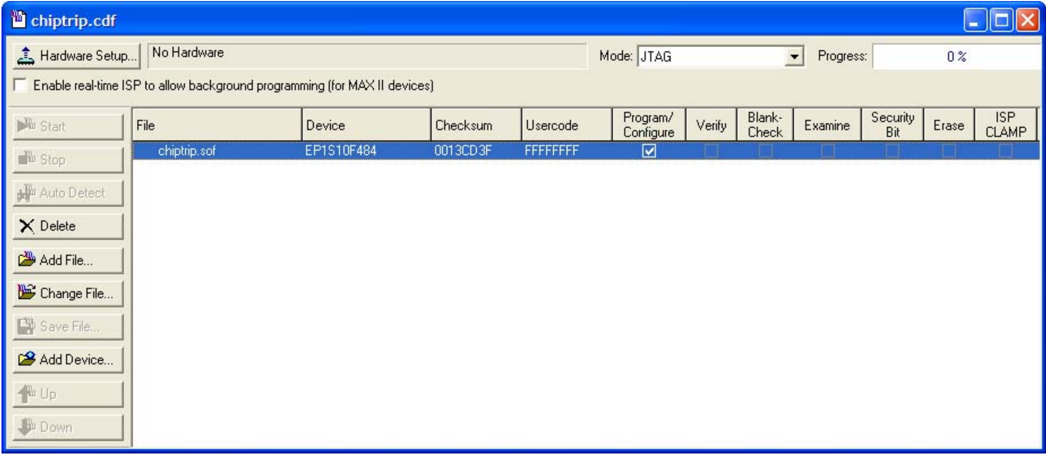


Figure 3–20 shows that the Programmer window now supports Erase capability.

You can save the programmer settings as a Chain Description File (**.cdf**). The **.cdf** file is an ASCII text file that stores device name, device order, and programming file name information.

Conclusion

The Quartus II software is the most comprehensive design environment available for programmable logic designs. Features such as the MAX+PLUS II look and feel help you make the transition from Altera's MAX+PLUS II design software and become more productive with the Quartus II software. The Quartus II software has all the capabilities and features of the MAX+PLUS II software and many more to speed up your design cycle.

Quartus II Command Reference for MAX+PLUS II Users

Table 3–4 lists the commands in the MAX+PLUS II software and gives their equivalent commands in the Quartus II software.

NA means either Not Applicable or Not Available. If a command is not listed, the command is the same in both tools.

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 1 of 8)

MAX+PLUS II Software	Quartus II Software
MAX+PLUS II Menu	
Hierarchy Display	View menu, Utility Windows, **Project Navigator**
Graphic Editor	Block Editor
Symbol Editor	Block Symbol Editor
Text Editor	Text Editor
Waveform Editor	Waveform Editor
Floorplan Editor	Assignments menu, **Timing Closure Floorplan**
Compiler	Tools menu, **Compiler Tool**
Simulator	Tools menu, **Simulator Tool**
Timing Analyzer	Tools menu, **Timing Analyzer Tool**
Programmer	Tools menu, **Programmer**
Message Processor	View menu, Utility Windows, **Messages**
File Menu	
File menu, Project, **Name** (Ctrl+J)	File menu, **Open Project** (Ctrl+J)
File menu, Project, **Set Project to Current File** (Ctrl+Shift+J)	Project menu, **Set as Top-Level Entity** (Ctrl+Shift+J) or File menu, **New Project Wizard**

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 2 of 8)

MAX+PLUS II Software	Quartus II Software
File menu, Project, **Save & Check** (Ctrl+K)	Processing menu, Start, **Start Analysis & Synthesis** (Ctrl+K) or Processing menu, Start, **Start Analysis & Elaboration**
File menu, Project, **Save & Compile** (Ctrl+L)	Processing menu, **Start Compilation** (Ctrl+L)
File menu, Project, **Save & Simulate** (Ctrl+Shift+L)	Processing menu, **Start Simulation** (Ctrl+I)
File menu, Project, **Compile & Simulate** (Ctrl+Shift+K)	Processing menu, **Start Compilation & Simulation** (Ctrl+Shift+K)
File menu, Project, **Archive**	Project menu, **Archive Project**
File menu, Project, <Recent Projects>	File menu, <*Recent Projects*>
File menu, **Delete File**	NA
File menu, **Retrieve**	NA
File menu, **Info** (Ctrl+I)	File menu, **File Properties**
File menu, **Create Default Symbol**	File menu, Create/Update, **Create Symbol Files for Current File**
File menu, **Edit Symbol**	(Block Editor) Edit menu, **Edit Selected Symbol**
File menu, **Create Default Include File**	File menu, Create/Update, **Create AHDL Include Files for Current File**
File menu, **Hierarchy Project Top** (Ctrl+T)	Project menu, Hierarchy, **Project Top** (Ctrl+T)
File menu, Hierarchy, **Up** (Ctrl+U)	Project menu, Hierarchy, **Up** (Ctrl+U)
File menu, Hierarchy, **Down** (Ctrl+D)	Project menu, Hierarchy, **Down** (Ctrl+D)
File menu, Hierarchy, **Top**	NA
File menu, Hierarchy, **Project Top** (Ctrl+T)	Project menu, Hierarchy, **Project Top** (Ctrl+T)
File menu, **MegaWizard Plug-In Manager**	Tools menu, **MegaWizard Plug-In Manager**
(Graphic Editor) File menu, **Size**	NA
(Waveform Editor) File menu, **End Time**	(Waveform Editor) Edit menu, **End Time**
(Waveform Editor) File menu, **Compare**	(Waveform Editor) View menu, **Compare to Waveforms in File**
(Waveform Editor) File menu**, Import Vector File**	File menu, **Open** (Ctrl+O)
(Waveform Editor) File menu**, Create Table File**	File menu, **Save As**
(Hierarchy Display) File menu, **Select Hierarchy**	NA
(Hierarchy Display) File menu, **Open Editor**	(Project Navigator) Double-click
(Hierarchy Display) File menu, **Close Editor**	NA
(Hierarchy Display) File menu, **Change File Type**	(Project Navigator) Select file in Files tab and select Properties on right click menu
(Hierarchy Display) File menu, **Print Selected Files**	NA
(Programmer) File menu, **Select Programming File**	File menu, **Open**
(Programmer) File menu, **Save Programming Data As**	File menu, **Save**

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 3 of 8)

MAX+PLUS II Software	Quartus II Software
(Programmer) File menu, **Inputs/Outputs**	NA
(Programmer) File menu, **Convert SRAM Object Files**	File menu, **Convert Programming Files**
(Programmer) File menu, **Archive JTAG Programming Files**	NA
(Programmer) File menu, **Create Jam or SVF File**	File menu, Create/Update, **Create JAM, SVF, or ISC File**
(Message Processor) Select Messages	NA
(Message Processor) Save Messages As	(Messages) Save Messages on right click menu
(Timing Analyzer) Save Analysis As	Processing menu, **Compilation Report** - Save Current Report on right click menu in Timing Analyzer sections
(Simulator) Create Table File	(Waveform Editor) File menu, **Save As**
(Simulator) Execute Command File	NA
(Simulator) Inputs/Outputs	NA
Edit Menu	
(Waveform Editor) Edit menu, **Overwrite**	(Waveform Editor) Edit menu, **Value**
(Waveform Editor) Edit menu, **Insert**	(Waveform Editor) Edit menu, **Insert Waveform Interval**
(Waveform Editor) Edit menu, **Align to Grid** (Ctrl+Y)	NA
(Waveform Editor) Edit menu, **Repeat**	(Waveform Editor) Edit menu, **Repeat Paste**
(Waveform Editor) Edit menu, **Grow or Shrink**	Edit menu, **Grow or Shrink** (Ctrl+Alt+G)
(Text Editor) Edit menu, **Insert Page Break**	(Text Editor) Edit menu, **Insert Page Break**
(Text Editor) Edit menu, **Increase Indent** (F2)	(Text Editor) Edit menu, **Increase Indent**
(Text Editor) Edit menu, **Decrease Indent** (F3)	(Text Editor) Edit menu, **Decrease Indent**
(Graphic Editor) Edit menu, **Toggle Connection Dot** (Double-Click)	(Block Editor) Edit menu, **Toggle Connection Dot**
(Graphic Editor) Edit menu, **Flip Horizontal**	(Block Editor) Edit menu, **Flip Horizontal**
(Graphic Editor) Edit menu, **Flip Vertical**	(Block Editor) Edit menu, **Flip Vertical**
(Graphic Editor) Edit menu, **Rotate**	(Block Editor) Edit menu, **Rotate by Degrees**
View Menu	
View menu, **Fit in Window** (Ctrl+W)	View menu, **Fit in Window** (Ctrl+W)
View menu, **Zoom In** (Ctrl+Space)	View menu, **Zoom In** (Ctrl+Space)
View menu, **Zoom Out** (Ctrl+Shift+Space)	View menu, **Zoom Out** (Ctrl+Shift+Space)
View menu, **Normal Size** (Ctrl+1)	NA
View menu, **Maximum Size** (Ctrl+2)	NA
(Hierarchy Display) View menu, **Auto Fit in Window**	NA
(Waveform Editor) View menu, **Time Range**	View menu, **Zoom**
Assign menu, **Device**	Assignments menu, **Device** or Assignments menu, **Settings** (Ctrl+Shift+E)

© March 2009 Altera Corporation

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 4 of 8)

MAX+PLUS II Software	Quartus II Software
Assign menu, **Pin/Location/Chip**	Assignments menu, **Assignment Editor -** Locations category
Assign menu, **Timing Requirements**	Assignments menu, **Assignment Editor -** Timing category
Assign menu, **Clique**	Assignments menu, **Assignment Editor -** Cliques category
Assign menu, **Logic Options**	Assignments menu, **Assignment Editor -** Logic Options category
Assign menu, **Probe**	NA
Assign menu, **Connected Pins**	Assignments menu, **Assignment Editor -** Simulation category
Assign menu, **Local Routing**	Assignments menu, **Assignment Editor -** Local Routing category
Assign menu, **Global Project Device Options**	Assignments menu, **Device -** Device and Pin Options
Assign menu, **Global Project Parameters**	Assignments menu, **Settings -** Analysis and Synthesis - Default Parameters
Assign menu, **Global Project Timing Requirements**	Assignments menu, **Timing Settings**
Assign menu, **Global Project Logic Synthesis**	Assignments menu, **Settings -** Analysis and Synthesis
Assign menu, **Ignore Project Assignments**	Assignments menu, **Assignment Editor -** disable
Assign menu, **Clear Project Assignments**	Assignments menu, **Remove Assignments**
Assign menu, **Back-Annotate Project**	Assignments menu, **Back-Annotate Assignments**
Assign menu, **Convert Obsolete Assignment Format**	NA
Utilities Menu	
Utilities menu, **Find Text** (Ctrl+F)	Edit menu, **Find** (Ctrl+F)
Utilities menu, **Find Node in Design File** (Ctrl+B)	Project menu, Locate, **Locate in Design File**
Utilities menu, **Find Node in Floorplan**	Project menu, Locate, **Locate in Timing Closure Floorplan**
Utilities menu, **Find Clique in Floorplan**	NA
Utilities menu, **Find Node Source** (Ctrl+Shift+S)	NA
Utilities menu**, Find Node Destination** (Ctrl+Shift+D)	NA
Utilities menu, **Find Next** (Ctrl+N)	Edit menu, **Find Next** (F3)
Utilities menu, **Find Previous** (Ctrl+Shift+N)	NA
Utilities menu, **Find Last Edit**	NA
Utilities menu, **Search and Replace** (Ctrl+R)	Edit menu, **Replace** (Ctrl+H)
Utilities menu, **Timing Analysis Source** (Ctrl+Alt+S)	NA
Utilities menu, **Timing Analysis Destination** (Ctrl+Alt+D)	NA
Utilities menu, **Timing Analysis Cutoff** (Ctrl+Alt+C)	NA
Utilities menu, **Analyze Timing**	NA
Utilities menu, **Clear All Timing Analysis Tags**	NA

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 5 of 8)

MAX+PLUS II Software	Quartus II Software
(Text Editor) Utilities menu, **Go To** (Ctrl+G)	Edit menu, **Go To** (Ctrl+G)
(Text Editor) Utilities menu, **Find Matching Delimiter** (Ctrl+M)	(Text Editor) Edit, **Find Matching Delimiter** (Ctrl+M)
(Waveform Editor) Utilities menu, **Find Next Transition** (Right Arrow)	(Waveform Editor) View menu, **Next Transition** (Right Arrow)
(Waveform Editor) Utilities menu, **Find Previous Transition** (Left Arrow)	(Waveform Editor) View menu, **Next Transition** (Left Arrow)
Options Menu	
Options menu, **User Libraries**	Assignments menu, **Settings** (Ctrl+Shift+E) Tools, Options, **Global User LIbraries**
Options menu, **Color Palette**	Tools menu, **Options**
Options menu, **License Setup**	Tools menu, **License Setup**
Options menu, **Preferences**	Tools menu, **Options**
(Hierarchy Display) Options menu, **Orientation**	NA
(Hierarchy Display) Options menu, **Compact Display**	NA
(Hierarchy Display) Options menu, **Show All Hierarchy Branches**	(Project Navigator) Expand All on right click menu
(Hierarchy Display) Options menu, **Hide All Hierarchy Branches**	NA
(Editors) Options menu, **Font**	Tools menu, **Options**
(Editors) Options menu, **Text Size**	Tools menu, **Options**
(Graphic Editor) Options menu, **Line Style**	Edit menu, **Line**
(Graphic Editor) Options menu, **Rubberbanding**	Tools menu, **Options**
(Graphic Editor) Options menu, **Show Parameters**	View menu, **Show Parameter Assignments**
(Graphic Editor) Options menu, **Show Probes**	NA
(Graphic Editor) Options menu**, Show Pins/Locations/Chips**	View menu, **Show Pin and Location Assignments**
(Graphic Editor) Options menu, **Show Clique, Timing & Local Routing Assignments**	NA
(Graphic Editor) Options menu, **Show Logic Options**	NA
(Graphic Editor) Options menu, **Show All** (Ctrl+Shift+M)	NA
(Graphic Editor) Options menu, **Show Guidelines** (Ctrl+Shift+G)	Tools menu, **Options** - Block/Symbol Editor page
(Graphic Editor) Options menu, **Guideline Spacing**	Tools menu, **Options** - Block/Symbol Editor page
(Symbol Editors) Options menu, **Snap to Grid**	Tools menu, **Options** - Block/Symbol Editor page
(Text Editor) Options menu, **Tab Stops**	Tools menu, **Options** - Text Editor page
(Text Editor) Options menu, **Auto-Indent**	Tools menu, **Options** - Text Editor page
(Text Editor) Options menu, **Syntax Coloring**	NA
(Waveform Editor) Options menu, **Snap to Grid**	View menu, **Snap to Grid**

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 6 of 8)

MAX+PLUS II Software	Quartus II Software
(Waveform Editor) Options menu, **Show Grid** (Ctrl+Shift+G)	Tools menu, **Options** - Waveform Editor page
(Waveform Editor) Options menu, **Grid Size**	Edit menu, **Grid Size** - Waveform Editor page
(Floorplan Editor) Options menu, **Routing Statistics**	NA
(Floorplan Editor) Options menu, **Show Node Fan-In**	View menu, Routing, **Show Fan-In**
(Floorplan Editor) Options menu, **Show Node Fan-Out**	View menu, Routing, **Show Fan-Out**
(Floorplan Editor) Options menu, **Show Path**	View menu, Routing, **Show Paths between Nodes**
(Floorplan Editor) Options menu, **Show Moved Nodes in Gray**	NA
(Simulator) Options menu, **Breakpoint**	Processing menu, Simulation Debug, **Breakpoints**
(Simulator) Options menu, **Hardware Setup**	NA
(Timing Analyzer) Options menu, **Time Restrictions**	Assignments menu, **Timing Settings**
(Timing Analyzer) Options menu, **Auto-Recalculate**	NA
(Timing Analyzer) Options menu, **Cell Width**	NA
(Timing Analyzer) Options menu, **Cut Off I/O Pin Feedback**	Assignments menu, **Timing Settings**
(Timing Analyzer) Options menu, **Cut Off Clear & Reset Paths**	Assignments menu, **Timing Settings**
(Timing Analyzer) Options menu, **Cut Off Read During Write Paths**	Assignments menu, **Timing Settings**
(Timing Analyzer) Options menu, **List Only Longest Path**	NA
(Programmer) Options menu, **Sound**	NA
(Programmer) Options menu, **Programming Options**	Tools menu, **Options** - Programmer page
(Programmer) Options menu, **Select Device**	(Programmer) Edit menu, **Change Device**
(Programmer) Options menu, **Hardware Setup**	(Programmer) Edit menu, **Hardware Setup**
Symbol (Graphic Editor)	
Symbol menu, **Enter Symbol** (Double-Click)	(Block Editor) Edit menu, **Insert Symbol** (Double-Click)
Symbol menu, **Update Symbol**	Edit menu, **Update Symbol or Block**
Symbol menu, **Edit Ports/Parameters**	Edit menu, **Properties**
Element (Symbol Editor)	
Element menu, **Enter Pinstub**	Double-click on edge of symbol
Element menu, **Enter Parameters**	NA
Templates (Text Editor)	
Templates	(Text Editor) Edit menu, **Insert Template**

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 7 of 8)

MAX+PLUS II Software	Quartus II Software
Node (Waveform Editor)	
Node menu, **Insert Node** (Double-Click)	Edit menu, **Insert Node or Bus** (Double-Click)
Node menu, **Enter Nodes from SNF**	Edit menu, **Insert Node -** click on Node Finder…
Node menu, **Edit Node**	Double-click on the Node
Node menu, **Enter Group**	Edit menu, **Group**
Node menu, **Ungroup**	Edit menu, **Ungroup**
Node menu, **Sort Names**	Edit menu, **Sort**
Node menu, **Enter Separator**	NA
Layout (Floorplan Editor)	
Layout menu, **Full Screen**	View menu, **Full Screen** (Ctrl+Alt+Space)
Layout menu, **Report File Equation Viewer**	View menu, **Equations**
Layout menu, **Device View** (Double-Click)	View menu, **Package Top** or View menu, **Package Bottom**
Layout menu, **LAB View** (Double-Click)	View menu, **Interior Labs**
Layout menu, **Current Assignments Floorplan**	View menu, Assignments, **Show User Assignments**
Layout menu, **Last Compilation Floorplan**	View menu, Assignments, **Show Fitter Assignments**
Processing (Compiler)	
Processing menu, **Design Doctor**	Processing menu, Start, **Start Design Assistant**
Processing menu, **Design Doctor Settings**	Assignments menu, **Settings** - Design Assistant
Processing menu, **Functional SNF Extractor**	Processing menu, **Generate Functional Simulation Netlist**
Processing menu, **Timing SNF Extractor**	Processing menu, **Start Analysis & Synthesis**
Processing menu, **Optimize Timing SNF**	NA
Processing menu, **Linked SNF Extractor**	NA
Processing menu, **Fitter Settings**	Assignments menu, **Settings** - Fitter Settings
Processing menu, **Report File Settings**	Assignments menu, **Settings**
Processing menu, **Generate AHDL TDO File**	NA
Processing menu, **Smart Recompile**	Assignments menu, **Settings** - Compilation Process
Processing menu, **Total Recompile**	Assignments menu, **Settings** - Compilation Process
Processing menu, **Preserve All Node Name Synonyms**	Assignments menu, **Settings** - Compilation Process
Interfaces (Compiler)	Assignments menu, **EDA Tool Settings**
Initialize (Simulator)	
Initialize menu, **Initialize Nodes/Groups**	NA
Initialize menu, **Initialize Memory**	NA

Table 3–4. Quartus II Command Reference for MAX+PLUS II Users (Part 8 of 8)

MAX+PLUS II Software	Quartus II Software
Initialize menu, **Save Initialization As**	NA
Initialize menu, **Restore Initialization**	NA
Initialize menu, **Reset to Initial SNF Values**	NA
Node (Timing Analyzer)	
Node menu, **Timing Analysis Source** (Ctrl+Alt+S)	NA
Node menu, **Timing Analysis Destination** (Ctrl+Alt+D)	NA
Node menu, **Timing Analysis Cutoff** (Ctrl+Alt+C)	NA
Analysis (Timing Analyzer)	
Analysis menu, **Delay Matrix**	(Timing Analyzer Tool) Delay tab
Analysis menu, **Setup/Hold Matrix**	NA
Analysis menu, **Registered Performance**	(Timing Analyzer Tool) Registered Performance tab
JTAG (Programmer)	
JTAG menu, **Multi-Device JTAG Chain**	(Programmer) Mode: JTAG
JTAG menu, **Multi-Device JTAG Chain Setup**	(Programmer) Window
JTAG menu, **Save JCF**	File menu, **Save**
JTAG menu, **Restore JCF**	File menu, **Open**
JTAG menu, **Initiate Configuration from Configuration Device**	Tools menu, **Options** - Programmer page
FLEX (Programmer)	
FLEX menu, **Multi-Device FLEX Chain**	(Programmer) Mode: Passive Serial
FLEX menu, **Multi-Device FLEX Chain Setup**	(Programmer) Window
FLEX menu, **Save FCF**	File menu, **Save**
FLEX menu, **Restore FCF**	File menu, **Open**

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Area and Timing Optimization chapter* in volume 2 of the *Quartus II Handbook*
- *Command Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Engineering Change Management with the Chip Planner* chapter in volume 3 of the *Quartus II Handbook*
- *Introduction to the Quartus II Software* manual
- *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*

- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 3–5 show the revision history of this chapter.

Table 3–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Removed "Quick Menu Reference"	Updated for the Quartus II 9.0 software release
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	Updated for the Quartus II 8.1 software release
May 2008 v8.0.0	Updated date and part number, added hypertext links.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

 © March 2009 Altera Corporation

4. Quartus II Support for HardCopy Series Devices

QII51004-9.0.0

Introduction

This chapter describes Quartus® II support for HardCopy® Series devices, including legacy HardCopy Stratix® series devices. This chapter is divided into the following sections:

- "HardCopy Series Device Support"
- "Legacy HardCopy Device Support" on page 4–34

HardCopy Series Device Support

Altera® HardCopy ASICs are the lowest risk, lowest total cost ASICs. The HardCopy system development methodology offers fast time-to-market, low risk, and using the Quartus II software, you can design with one set of RTL code and one IP set for both FPGA and ASIC implementations. This flow enables you to conduct true hardware/software co-design and completely prepare your system for production prior to ASIC design hand-off. Altera provides a turn-key process to convert your design to a HardCopy ASIC for production.

In this chapter, the term FPGA refers to a Stratix® II, Stratix III, or Stratix IV device which is the prototype device for a HardCopy II, HardCopy III, or HardCopy IV device, respectively.

For legacy HardCopy Stratix devices, refer to "Legacy HardCopy Device Support" on page 4–34.

This chapter discusses the following topics:

- "HardCopy Development Flow" on page 4–3
- "HardCopy Device Resource Guide" on page 4–7
- "HardCopy Companion Device Selection" on page 4–10
- "HardCopy Recommended Settings in the Quartus II Software" on page 4–11
- "HardCopy Utilities Menu" on page 4–17
- "HardCopy Design Readiness Check" on page 4–23
- "Performing ECOs with Quartus II Engineering Change Management with the Chip Planner" on page 4–29
- "Formal Verification of FPGA and HardCopy Revisions" on page 4–32
- "Legacy HardCopy Device Support" on page 4–34

For more information about HardCopy series devices, refer to the respective HardCopy device handbook on the Altera website (www.altera.com).

HardCopy Series Design Benefits

Designing with HardCopy ASICs offers substantial benefits over other ASIC offerings:

- Seamless prototyping using an FPGA for at-speed system verification and system development reduces total project development time and cost
- Dependable conversion from an FPGA prototype to a HardCopy ASIC expands product planning options
- Unified design methodology for FPGA design and HardCopy design reduces the need for ASIC development software, two sets of intellectual property, and project risk
- System development methodology delivers lowest total cost

Quartus II Features for HardCopy Planning

With the Quartus II software, you can design a HardCopy ASIC using seamless FPGA prototyping. The Quartus II software provides the following expanded features for HardCopy series device planning:

- **HardCopy Companion Device Assignment**—Identifies compatible HardCopy series devices for use with the FPGA prototyping device currently selected.

  This feature constrains the pins of your FPGA prototype, making it compatible with your HardCopy device. It also constrains the correct resources available for the HardCopy device, ensuring the compatibility of your FPGA design. In addition, you are required to compile the design targeting the HardCopy device to ensure that the design fits, routes, and meets timing requirements.

  Beginning with Quartus II software version 8.0, you can select HardCopy III as the companion device, but you cannot compile the HardCopy III device. This ensures that the FPGA is compatible with the HardCopy III device in the areas of pins, I/O standards, logic, and other resources. Compilation for the HardCopy III family will be supported in a later release of the Quartus II software.

  Beginning with Quartus II software version 8.1, you can select HardCopy IV E as the companion device, but you cannot compile the HardCopy IV E device. This ensures that the FPGA is compatible with the HardCopy IV E device in the areas of pins, I/O standards, logic, and other resources. Compilation for the HardCopy IV E will be supported in a later release of the Quartus II software. Companion and Compilation for the HardCopy IV GX family will be supported in a later release of the Quartus II software.

- **HardCopy Utilities**—The HardCopy Utilities menu provides a variety of functions to create or overwrite HardCopy companion revisions, change revisions to use, and compare revisions for equivalency.
- **HardCopy Advisor**—The HardCopy Advisor helps you follow the necessary steps to successfully submit a HardCopy design to Altera's HardCopy Design Center.

  The HardCopy Advisor is similar to other advisors in the Quartus II software. The HardCopy Advisor provides guidelines you can follow during development, reporting completed and uncompleted tasks.

- **HardCopy Floorplan**—The Quartus II Chip Planner software can show a preliminary floorplan view of your HardCopy design's Fitter placement results.
- **HardCopy Device Preliminary Timing**—The Quartus II TimeQuest software performs a timing analysis of HardCopy devices based on preliminary timing models and Fitter placements. Final timing results for HardCopy devices are provided by Altera's HardCopy Design Center.
- **HardCopy Design Readiness Check**—The Quartus II software tool checks the project settings to ensure compliance with the HardCopy device settings, I/O, PLL, and RAM usage checks.
- **HardCopy Handoff Report**—The Quartus II software generates a handoff report containing information about the HardCopy design used by Altera's HardCopy Design Center in the design review process.
- **HardCopy Design Archiving**—The Quartus II software archives the HardCopy design project's files required to hand off the design to Altera's HardCopy Design Center.
- **Formal Verification**—Cadence Encounter Conformal software performs formal verification between the source RTL design files and post-compile gate level netlist from a HardCopy design.

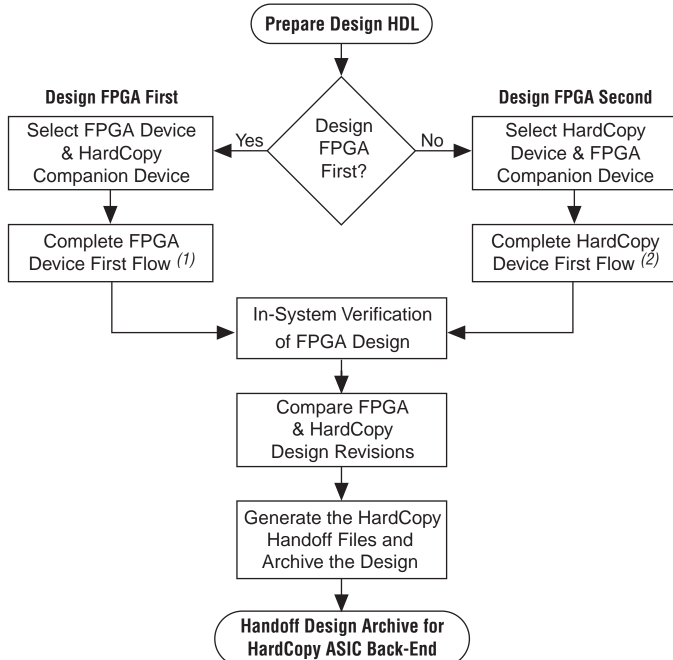
HardCopy Development Flow

In the Quartus II software, you design your FPGA and HardCopy companion device together in one Quartus II project using one of the following methods:

- Design the FPGA first for in-system verification and then create a HardCopy companion device second
- Design the HardCopy device first and then create the FPGA companion device second for in-system verification

Both of these flows are illustrated at a high level in Figure 4–1. The added features in the HardCopy Utilities menu help you complete your HardCopy design for submission to Altera's HardCopy Design Center for back-end implementation.

Figure 4–1. HardCopy Flow in Quartus II Software



Notes to Figure 4–1:

(1) Refer to Figure 4–2 on page 4–5 for an expanded description of this process.

(2) Refer to Figure 4–3 on page 4–7 for an expanded description of this process.

The FPGA first flow is the default flow and the rest of this chapter is based on this flow.

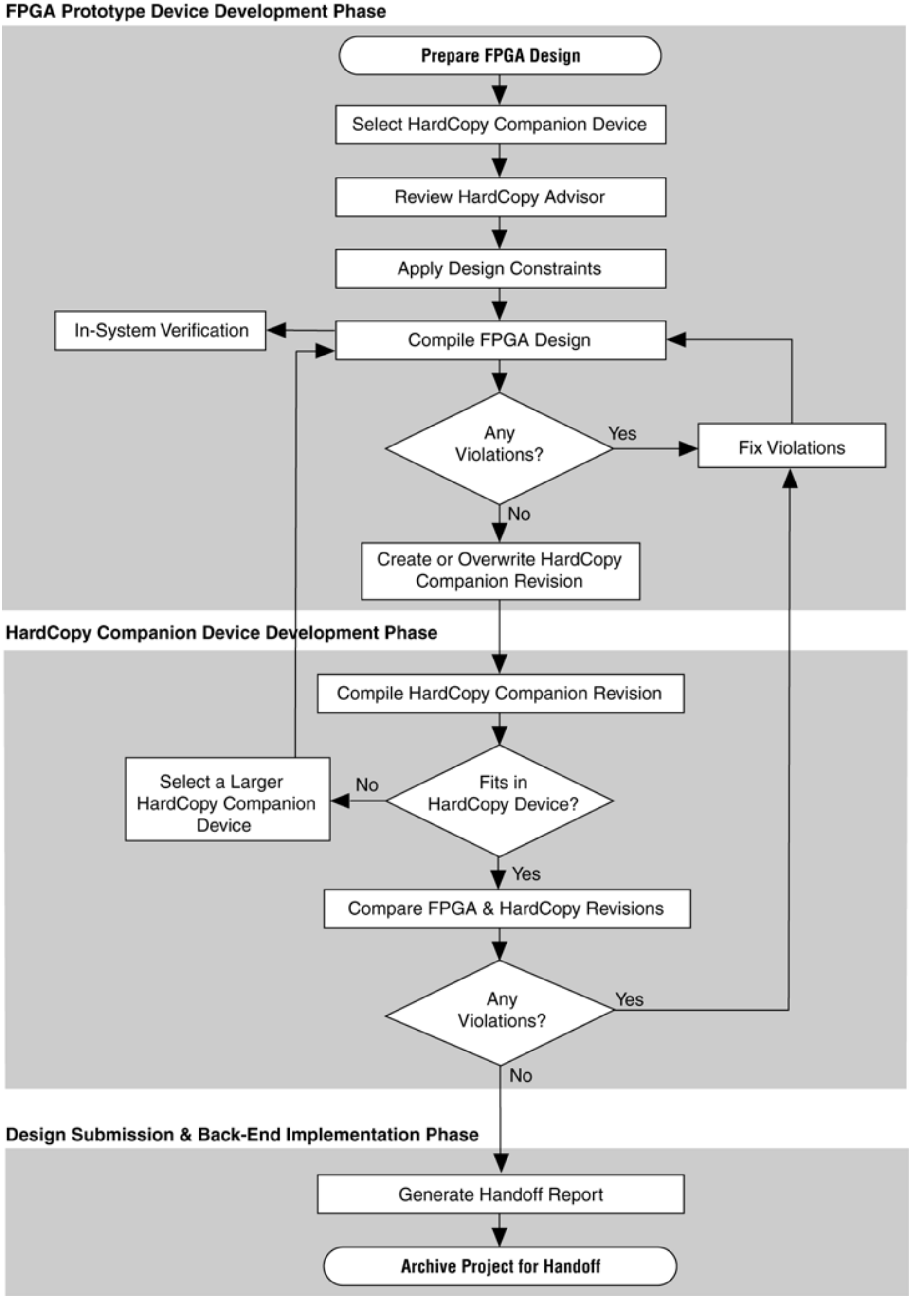
Designing the FPGA First

The HardCopy development flow, with the FPGA first flow begins with seamless FPGA prototyping, is identical to the traditional FPGA design flow, with a few additional tasks to be performed to convert the design to the HardCopy companion device within the same project. To design your HardCopy device when selecting the FPGA companion device first, complete the following tasks:

- Specify an FPGA device and a HardCopy companion device
- Compile the FPGA design
- Create and compile the HardCopy companion revision
- Compare the HardCopy companion revision compilation to the FPGA device compilation

Figure 4–2 provides an overview highlighting the development process for designing with an FPGA first and creating a HardCopy companion device second.

Figure 4–2. Designing FPGA Device First Flow



You must select a target FPGA device and a companion HardCopy device when compiling an FPGA design that you will migrate over to a HardCopy device.

During the early stages of the design, picking the right HardCopy device can be a problem. In such cases, the HardCopy Device Resource Guide should help. After you have selected an FPGA and a HardCopy Device, compile the FPGA and review the HardCopy Device Resource Guide to see if all resources are available in the targeted HardCopy device. If there are not enough resources available in the target HardCopy device, you must select a larger HardCopy device and restart the FPGA compilation.

Once the FPGA and the HardCopy devices have been finalized perform the following tasks:

- Review the HardCopy Advisor for required and recommended tasks to perform
- Enable Design Assistant to run during compilation
- Add timing and location assignments
- Compile your FPGA design
- Create your HardCopy companion revision
- Compile your design for the HardCopy companion device
- Compare the HardCopy companion device compilation with the FPGA revision
- Generate a HardCopy Handoff Report
- Generate a HardCopy Handoff Archive
- Arrange for submission of your HardCopy Handoff Archive to Altera's HardCopy Design Center for back-end implementation

For more information about the overall design flow using the Quartus II software, refer to the *Introduction to the Quartus II Software* manual.

Designing the HardCopy Device First

After you have selected an initial HardCopy ASIC device, you can design your HardCopy device first and create your FPGA prototype second in the Quartus II software. This approach is best when using the HardCopy ASIC to achieve higher performance than the FPGA prototype, because you can see your potential maximum performance in the HardCopy device immediately during development, and you can create a slower performing FPGA prototype of the design for in-system verification. This design process is similar to the HardCopy design flow where you build the FPGA first, but instead, you change the starting device family. The remaining tasks to complete your design for both the FPGA and HardCopy devices roughly follow the same process (Figure 4–3). The HardCopy Advisor adjusts its list of tasks based on which device family you start with, FPGA or HardCopy, to help you complete the process seamlessly.

Figure 4–3. Designing HardCopy Device First Flow

HardCopy Device Development Phase

Prepare HardCopy Design
Select FPGA Companion Device
Review HardCopy Advisor
Apply Design Constraints
Compile HardCopy Design
Any Violations?
Yes
Fix Violations
No
Create or Overwrite FPGA Companion Revision

FPGA Companion Device Development Phase

Compile FPGA Companion Revision
In-System Verification
Compare HardCopy & FPGA Revisions
Any Violations?
Yes
No

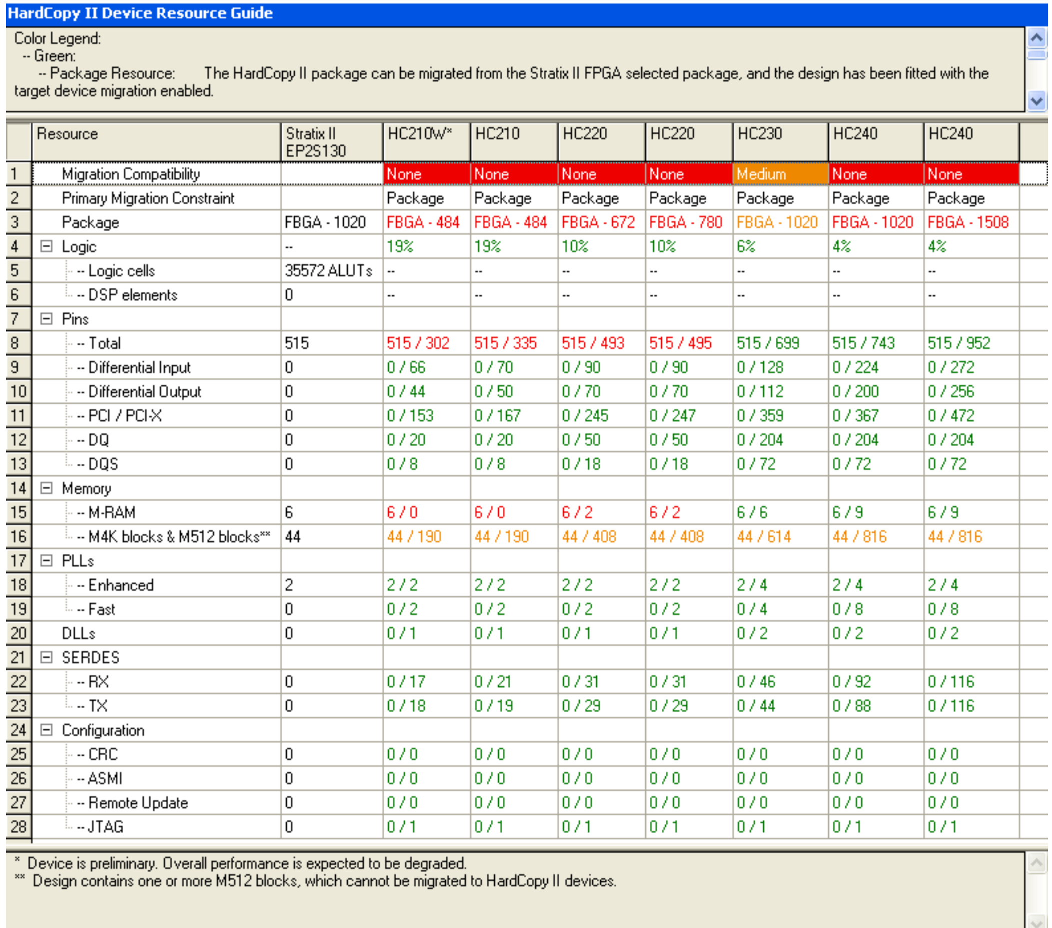
Design Submission & Back-End Implementation Phase

Generate Handoff Report
Archive HardCopy Archive for Handoff

HardCopy Device Resource Guide

The HardCopy Device Resource Guide compares the resources required to successfully compile a design with the resources available in the various HardCopy devices. The report rates each HardCopy device and each device resource on how well it fits the design. The Quartus II software generates the HardCopy Device Resource Guide for all designs successfully compiled for FPGA devices. This guide is found in the Fitter folder of the Compilation Report. Figure 4–4 shows an example of the HardCopy Device Resource Guide. Refer to Table 4–1 for an explanation of the color codes in Figure 4–4.

Figure 4–4. HardCopy Device Resource Guide



HardCopy II Device Resource Guide

Color Legend:
-- Green:
-- Package Resource: The HardCopy II package can be migrated from the Stratix II FPGA selected package, and the design has been fitted with the target device migration enabled.

	Resource	Stratix II EP2S130	HC210W*	HC210	HC220	HC220	HC230	HC240	HC240
1	Migration Compatibility		None	None	None	None	Medium	None	None
2	Primary Migration Constraint		Package	Package	Package	Package	Package	Package	Package
3	Package	FBGA - 1020	FBGA - 484	FBGA - 484	FBGA - 672	FBGA - 780	FBGA - 1020	FBGA - 1020	FBGA - 1508
4	⊟ Logic	--	19%	19%	10%	10%	6%	4%	4%
5	-- Logic cells	35572 ALUTs	--	--	--	--	--	--	--
6	-- DSP elements	0	--	--	--	--	--	--	--
7	⊟ Pins								
8	-- Total	515	515 / 302	515 / 335	515 / 493	515 / 495	515 / 699	515 / 743	515 / 952
9	-- Differential Input	0	0 / 66	0 / 70	0 / 90	0 / 90	0 / 128	0 / 224	0 / 272
10	-- Differential Output	0	0 / 44	0 / 50	0 / 70	0 / 70	0 / 112	0 / 200	0 / 256
11	-- PCI / PCI-X	0	0 / 153	0 / 167	0 / 245	0 / 247	0 / 359	0 / 367	0 / 472
12	-- DQ	0	0 / 20	0 / 20	0 / 50	0 / 50	0 / 204	0 / 204	0 / 204
13	-- DQS	0	0 / 8	0 / 8	0 / 18	0 / 18	0 / 72	0 / 72	0 / 72
14	⊟ Memory								
15	-- M-RAM	6	6 / 0	6 / 0	6 / 2	6 / 2	6 / 6	6 / 9	6 / 9
16	-- M4K blocks & M512 blocks**	44	44 / 190	44 / 190	44 / 408	44 / 408	44 / 614	44 / 816	44 / 816
17	⊟ PLLs								
18	-- Enhanced	2	2 / 2	2 / 2	2 / 2	2 / 2	2 / 4	2 / 4	2 / 4
19	-- Fast	0	0 / 2	0 / 2	0 / 2	0 / 2	0 / 4	0 / 8	0 / 8
20	DLLs	0	0 / 1	0 / 1	0 / 1	0 / 1	0 / 2	0 / 2	0 / 2
21	⊟ SERDES								
22	-- RX	0	0 / 17	0 / 21	0 / 31	0 / 31	0 / 46	0 / 92	0 / 116
23	-- TX	0	0 / 18	0 / 19	0 / 29	0 / 29	0 / 44	0 / 88	0 / 116
24	⊟ Configuration								
25	-- CRC	0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
26	-- ASMI	0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
27	-- Remote Update	0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
28	-- JTAG	0	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1

* Device is preliminary. Overall performance is expected to be degraded.
** Design contains one or more M512 blocks, which cannot be migrated to HardCopy II devices.

Use this report to determine which HardCopy device is a potential candidate for your design. The HardCopy device package must be compatible with the FPGA device package. A logic resource usage greater than 100% or a ratio greater than 1/1 in any category indicates that the design probably will not fit in that particular HardCopy device.

© March 2009 Altera Corporation

Table 4–1. HardCopy Device Resource Guide Color Legend

Color	Package Resource *(1)*	Device Resources
Green (High)	The design can map to the HardCopy package and the design has been fitted with target device migration enabled in the **HardCopy Companion Device** dialog box.	The resource quantity is within the range of the HardCopy device and the design can likely map if all other resources also fit. You are still required to compile the HardCopy revision to make sure the design is able to route and close timing.
Orange (Medium)	The design can map to the HardCopy package. However, the design has not been fitted with the target device migration enabled in the **HardCopy Companion Device** dialog box.	The resource quantity is within the range of the HardCopy device. However, the resource is at risk of exceeding the range for the HardCopy package. If your target HardCopy device falls in this category, compile your design targeting the HardCopy device as soon as possible to check if the design fits and is able to route and migrate all other resources. You might have to select a larger device.
Red (None)	The design cannot map to the HardCopy package.	The resource quantity exceeds the range of the HardCopy device. The design cannot migrate to this HardCopy device.

Note to Table 4–1:

(1) The package resource is constrained by the FPGA for which the design was compiled. Only vertical migration devices within the same package are able to migrate to HardCopy devices.

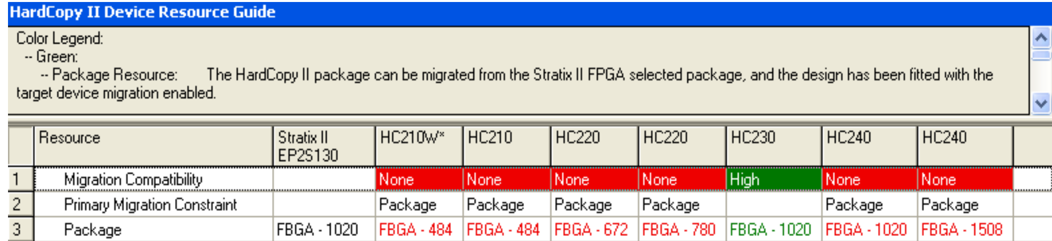
The HardCopy architecture consists of an array of fine-grained HCells, which are used to build logic equivalent to FPGA adaptive logic modules (ALMs) and digital signal processing (DSP) blocks. The DSP blocks in HardCopy devices match the functionality of the FPGA DSP blocks, though timing of these blocks is different than the FPGA DSP blocks because they are constructed of HCell Macros. The memory blocks in HardCopy devices are equivalent to the FPGA memory blocks. Preliminary timing reports of the HardCopy device are available in the Quartus II software. Final timing results of the HardCopy device are provided by Altera's HardCopy Design Center after the HardCopy back-end is complete.

For more information about the HardCopy device resources, refer to the respective HardCopy Series device handbook on the Altera website.

The report example in Figure 4–4 shows the resource comparisons for a design compiled for an EP2S130F1020 device. Based on the report, the HC230F1020 device in the 1,020-pin FineLine BGA package is an appropriate HardCopy device. If the HC230F1020 device is not specified as a migration target during the compilation, its package and migration compatibility is rated orange, or Medium. The migration compatibilities of the other HardCopy devices are rated red, or None, because the package types are incompatible with the FPGA device. The 1,020-pin FBGA HC240 device is rated red because it is only compatible with the EP2S180F1020 device.

Figure 4–5 shows the report after the (unchanged) design was recompiled with the HardCopy HC230F1020 device specified as a migration target. Now the HC230F1020 device package and migration compatibility is rated green, or High.

Figure 4–5. HardCopy Device Resource Guide with Target Migration Enabled



HardCopy II Device Resource Guide

Color Legend:
-- Green:
-- Package Resource: The HardCopy II package can be migrated from the Stratix II FPGA selected package, and the design has been fitted with the target device migration enabled.

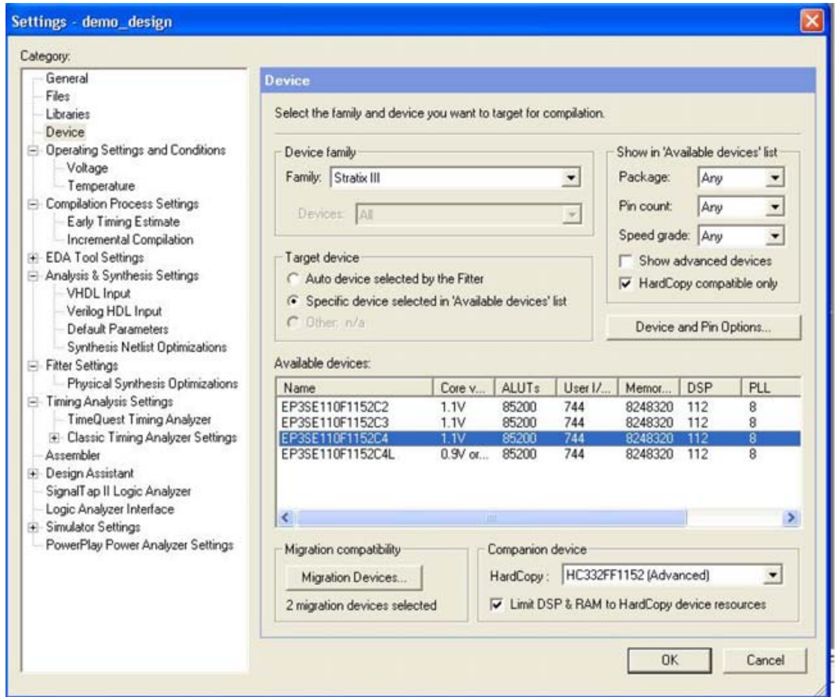
	Resource	Stratix II EP2S130	HC210W*	HC210	HC220	HC220	HC230	HC240	HC240
1	Migration Compatibility		None	None	None	None	High	None	None
2	Primary Migration Constraint		Package	Package	Package	Package		Package	Package
3	Package	FBGA - 1020	FBGA - 484	FBGA - 484	FBGA - 672	FBGA - 780	FBGA - 1020	FBGA - 1020	FBGA - 1508

HardCopy Companion Device Selection

In the Quartus II software, you can select a HardCopy companion device to ensure compatibility between the FPGA design and the HardCopy device's resources. To make your HardCopy companion device selection, on the Assignments menu, click **Device** (Figure 4–6) and select your companion device from the **Companion device** list.

Selecting a HardCopy companion device for your FPGA prototype constrains the memory blocks, DSP blocks, and pin assignments, so that your design fits into the HardCopy device resources. Pin assignments are constrained in the FPGA design revision, so that the HardCopy device selected is pin-compatible. The Quartus II software also constrains the FPGA design revision so that identical device resources are targeted in both the FPGA and the HardCopy ASIC.

Figure 4–6. Quartus II Settings Dialog Box



You can also specify your HardCopy companion device using the following tool command language (Tcl) command:

```
set_global_assignment -name\
DEVICE_TECHNOLOGY_MIGRATION_LIST <HardCopy Device Part Number>
```

For example, to select the HC230F1020 device as your HardCopy companion device for the EP2S130F1020C4 FPGA, the Tcl command is:

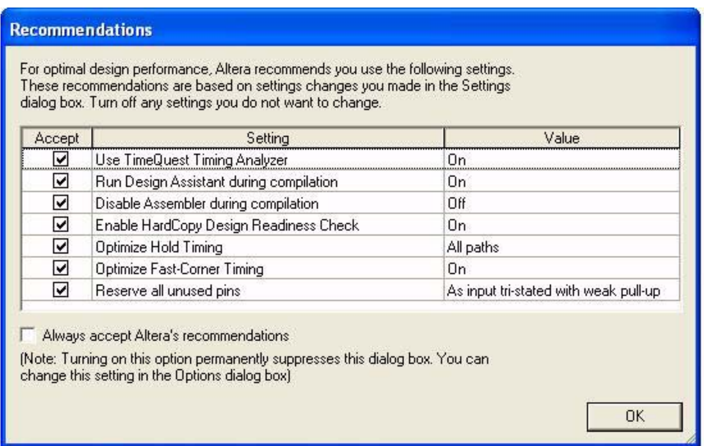
```
set_global_assignment -name\
DEVICE_TECHNOLOGY_MIGRATION_LIST HC230F1020C
```

HardCopy Recommended Settings in the Quartus II Software

The HardCopy development flow involves additional planning and preparation in the Quartus II software compared to a standard FPGA design. This is because you are developing your design to be implemented in two devices: a prototype of your design/system in an FPGA and a companion revision in a HardCopy device for production. Additional settings and constraints are required to make the FPGA design compatible with the HardCopy device, and in some cases, you must remove certain settings in the design. This section explains the additional settings and constraints necessary for your design to be successful in both FPGA and HardCopy ASIC devices.

Figure 4–7 shows the **Recommendations** dialog box with the recommended settings.

Figure 4–7. Quartus II Recommended Settings

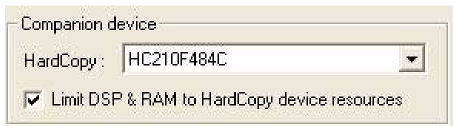


Limit DSP and RAM to HardCopy Device Resources

On the Assignments menu, click **Device**. For example, if the prototype device is a Stratix II FPGA, in the **Family** list, select **Stratix II**. Under **Companion device**, **Limit DSP & RAM to HardCopy device resources** is turned on by default (Figure 4–8). This setting maintains compatibility between the FPGA and HardCopy devices by ensuring your design does not use resources in the FPGA device that are not available in the selected HardCopy device or vice versa.

If you require additional memory blocks or DSP blocks for debugging purposes using the SignalTap® II Embedded Logic Analyzer, you can temporarily turn this setting off to compile and verify your design in your test environment. However, your final FPGA and HardCopy designs submitted to Altera for the HardCopy back-end must be compiled with this setting turned on.

Figure 4–8. Limit DSP & RAM to HardCopy Device Resources Check Box



Enable Design Assistant to Run During Compile

You must use the Quartus II Design Assistant to check all HardCopy designs for design rule violations before submitting the designs to Altera's HardCopy Design Center. Additionally, you must fix all critical and high-level errors.

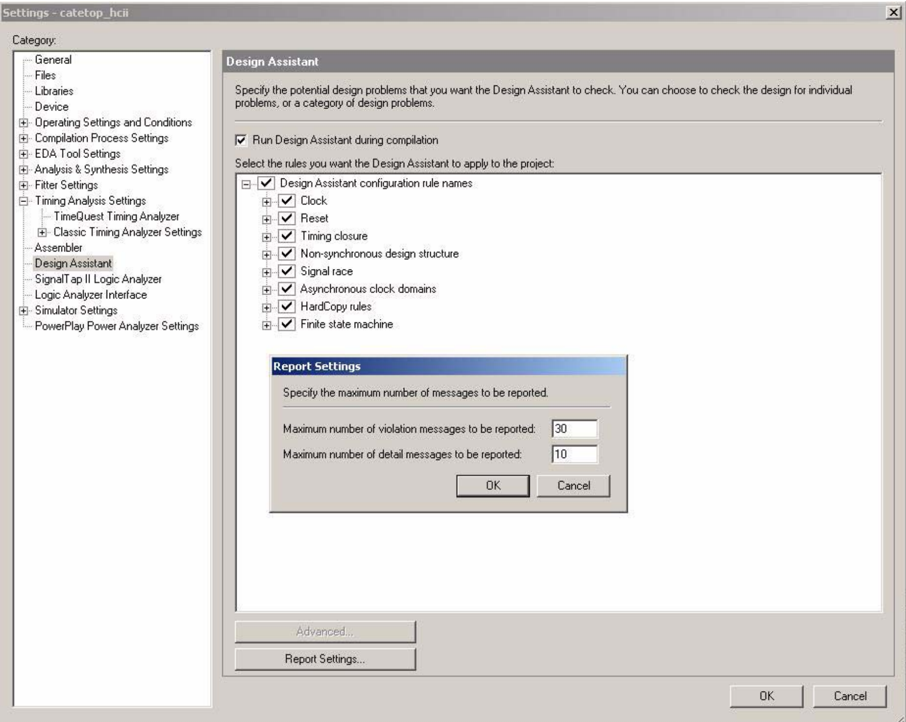
Altera recommends turning on the Design Assistant to run automatically during each compilation so that you can see the violations you must fix or waive after reviewing each violation.

For more information about the Design Assistant and its rules, refer to the respective HardCopy Series device handbook on the Altera website.

To enable the Design Assistant to run during compilation, on the Assignments menu, click **Settings**. In the **Category** list, select **Design Assistant** and turn on **Run Design Assistant during compilation** (Figure 4–9) or enter the following Tcl command in the Tcl Console:

```
set_global_assignment -name ENABLE_DRC_SETTINGS ON ↵
```

Figure 4–9. Enabling Design Assistant



Timing Settings

Beginning with Quartus II software version 7.1, the TimeQuest Timing Analyzer is the required timing analysis tool for all designs. The Classic Timing Analyzer is no longer supported and Altera's HardCopy Design Center does not accept any designs that use the Classic Timing Analyzer for timing closure.

If you are using the Classic Timing Analyzer, Altera strongly recommends that you switch to the TimeQuest Timing Analyzer.

For more information about switching to the TimeQuest Timing Analyzer, refer to the *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

When you specify the TimeQuest Timing Analyzer as the timing analysis tool, the TimeQuest Timing Analyzer guides the Fitter and analyzes timing results after compilation.

TimeQuest Timing Analyzer

The TimeQuest Timing Analyzer is a powerful ASIC-style timing analysis tool that validates timing in your design by using an industry-standard constraint, analysis, and reporting methodology. You can use the TimeQuest Timing Analyzer's GUI or command-line interface to constrain, analyze, and report results for all timing paths in your design.

Before running the TimeQuest Timing Analyzer, you must specify initial timing constraints that describe the clock characteristics, timing exceptions, and signal transition arrival and required times. You can specify timing constraints in the Synopsys Design Constraints (**.sdc**) file format using the GUI or command-line interface. The Quartus II Fitter optimizes the placement of logic to meet your constraints.

During timing analysis, the TimeQuest Timing Analyzer analyzes the timing paths in the design, calculates the propagation delay along each path, checks for timing constraint violations, and reports timing results as slack in the **Report** pane and in the **Console** pane. If the TimeQuest Timing Analyzer reports any timing violations, you can customize the reporting to view precise timing information about specific paths, and then constrain those paths to correct the violations. When your design is free of timing violations, you can be confident that the logic will operate as intended in the target device.

The TimeQuest Timing Analyzer is a complete static timing analysis tool that you use as a sign-off tool for Altera FPGAs and HardCopy ASICs.

Setting Up the TimeQuest Timing Analyzer

To use the TimeQuest Timing Analyzer for timing analysis, on the Assignments menu, click **Timing Analysis Settings**, and on the **Timing Analysis Settings** page, select **Use TimeQuest Timing Analyzer during compilation**.

Use the following Tcl command to use the TimeQuest Timing Analyzer as your timing analysis engine:

```
set_global_assignment -name USE_TIMEQUEST_TIMING_ANALYZER ON
```

You can launch the TimeQuest Timing Analyzer in one of the following modes:

- Directly from the Quartus II software
- Stand-alone mode
- Command-line mode

To perform a thorough Static Timing Analysis, you must specify all the timing requirements. The most important timing requirements are clocks and generated clocks, input and output delays, false paths and multi-cycle paths, and minimum and maximum delays.

In the TimeQuest Timing Analyzer, clock latency, and recovery and removal analysis are enabled by default.

For more information about the TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Constraints for Clock Effect Characteristics

The `create_clock`, `create_generated_clock` commands create ideal clocks and do not account for board effects. In order to account for clock effect characteristics, you can use the following commands:

- `set_clock_latency`
- `set_clock_uncertainty`

For more information about how to use these commands, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Beginning in Quartus II software version 7.1, you can use the command `derive_clock_uncertainty` to automatically derive the clock uncertainties in your **.sdc** file. This command is useful when you are unsure what the clock uncertainties might be. The calculated clock uncertainty values are based on I/O buffer, static phase errors (SPE) and jitter in the PLLs, clock networks, and core noise.

The `derive_clock_uncertainty` command applies inter-clock, intra-clock, and I/O interface uncertainties. This command automatically calculates and applies setup and hold clock uncertainties for each clock-to-clock transfer found in your design.

To determine I/O interface uncertainty, you must create a virtual clock, then assign delays to the input/output ports by using the `set_input_delay` and `set_output_delay` commands for that virtual clock.

These uncertainties are applied in addition to those you specified using the `set_clock_uncertainty` command. However, if a clock uncertainty assignment for a source and destination pair was already defined, the new one is ignored. In this case, you can use either the `-overwrite` command to overwrite the previous clock uncertainty command, or manually remove them by using the `remove_clock_uncertainty` command.

The syntax for the `derive_clock_uncertainty` command is as follows:

```
derive_clock_uncertainty [-h | -help] [-long_help] \
[-overwrite]
```

where the arguments are listed in Table 4–2:

Table 4–2. Arguments for derive_clock_uncertainty

Option	Description
`-h | -help`	Short help
`-long_help`	Long help with examples and possible return values
`-overwrite`	Overwrites previously performed clock uncertainty assignments

When the `derive_clock_uncertainty` constraint is used, a **PLLJ_PLLSPE_INFO.txt** file is automatically generated in the project directory. This file lists the names of the PLLs, as well as their jitter and SPE values in the design. This text file can be used by the HCII_DTW_CU_Calculator.

For more information about the `derive_clock_uncertainty` command, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Altera strongly recommends that you use the `derive_clock_uncertainty` command in the HardCopy revision. Altera's HardCopy Design Center does not accept designs that do not have clock uncertainty constraint by either using the `derive_clock_uncertainty` command or the HardCopy II Clock Uncertainty Calculator, and then using the `set_clock_uncertainty` command.

For more information about how to use the HardCopy II Clock Uncertainty Calculator, refer to the *HardCopy II Clock Uncertainty Calculator User Guide*.

Quartus II Software Features Supported for HardCopy Designs

The Quartus II software supports optimization features for HardCopy prototype development, including:

- Physical Synthesis Optimization
- LogicLock™ Regions
- PowerPlay Power Analyzer
- Incremental Compilation (Synthesis and Fitter)

Physical Synthesis Optimization

To enable **Physical Synthesis Optimizations** for the FPGA revision of the design, on the Assignments menu, click **Settings**. In the **Settings** dialog box, in the **Category** list, expand **Fitter Settings**. These optimizations are passed into the HardCopy companion revision for placement and timing closure. When designing with a HardCopy device first, physical synthesis optimizations can be enabled for the HardCopy device, and these post-fit optimizations are passed to the FPGA revision.

LogicLock Regions

The use of LogicLock regions in the FPGA is supported for designs targeted to HardCopy devices. However, LogicLock regions are not passed into the HardCopy companion revision. You can use LogicLock regions in the HardCopy design, but you must create new LogicLock regions in the HardCopy companion revision. In addition, LogicLock regions in HardCopy devices cannot have their properties set to **Auto Size**. However, floating LogicLock regions are supported. HardCopy LogicLock regions must be manually sized and placed in the floorplan. When LogicLock regions are created in a HardCopy device, they start with width and height dimensions set to (**1**,**1**), and the origin coordinates for placement are at **X1_Y1** in the lower left corner of the floorplan. You must adjust the size and location of the LogicLock regions you created in the HardCopy device before compiling the design.

For information about using LogicLock regions, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

PowerPlay Power Analyzer

You can perform power estimation and analysis of your HardCopy and FPGA devices using the PowerPlay Early Power Estimator. Use the PowerPlay Power Analyzer for more accurate estimation of your device’s power consumption.

For more information about using the PowerPlay Power Analyzer, refer to the *Quartus II PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

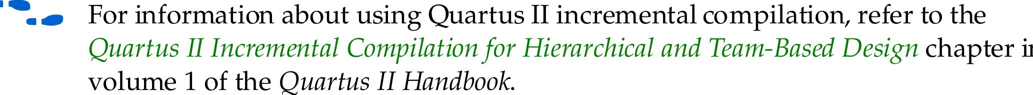
Incremental Compilation

Quartus II incremental compilation using the top-down flow in the FPGA is supported in both the FPGA First design flow and the HardCopy First design flow.

To take advantage of Quartus II Incremental Compilation, organize your design into logical and physical partitions for synthesis and fitting (or place-and-route). Incremental compilation preserves the compilation results and performance of unchanged partitions in your design. This feature dramatically reduces your design iteration time by focusing new compilations only on changed design partitions. New compilation results are then merged with the previous compilation results from unchanged design partitions. You can also target optimization techniques, such as physical synthesis, to specific partitions while leaving other partitions untouched.

In addition, be aware of the following guidelines:

- User partitions and synthesis results are passed to a companion device.
- LogicLock regions are suggested for user partitions, but are not migrated automatically.
- The compilation after migration to a companion device requires a full compilation (all partitions are compiled).
- The entire design must be migrated between the FPGA and HardCopy companion devices. The Quartus II software does not support migration of partitions between companion devices.
- Bottom-up Quartus II incremental compilation is not supported for designs targeting HardCopy devices.
- Physical Synthesis can be run on individual partitions within the originating device only. The resulting optimizations are preserved in the migration to the companion device.



For information about using Quartus II incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

HardCopy Utilities Menu

The HardCopy Utilities menu contains the main functions you use to develop your HardCopy design and FPGA prototype companion revision. To access this menu, on the Project menu, click **HardCopy Utilities**. From the HardCopy Utilities menu, you can perform the following tasks:

- Create or update HardCopy companion revisions
- Specify the current HardCopy companion revision
- Compare the companion revisions for functional equivalence
- Generate a HardCopy Handoff Report for design reviews
- Archive HardCopy Handoff Files for submission to Altera's HardCopy Design Center
- Enable the HardCopy Design Readiness Check tool if you disabled it (the tool is enabled by default)
- Track your design progress using the HardCopy Advisor

Each of the features within **HardCopy Utilities** is summarized in Table 4–3. The process for using each of these features is explained in the following sections.

Table 4–3. HardCopy Utilities Menu Options

Menu	Description	Applicable Design Revision	Restrictions
Create/Overwrite HardCopy Companion Revision	Create a new companion revision or update an existing companion revision for your FPGA and HardCopy design.	FPGA prototype design and HardCopy Companion Revision	■ Must disable Auto Device selection ■ Must set an FPGA device and a HardCopy companion device
Set Current HardCopy Companion Revision	Specify which companion revision to associate with current design revision.	FPGA prototype design and HardCopy Companion Revision	Companion Revision must already exist
Compare HardCopy Companion Revisions	Compares the FPGA design revision with the HardCopy companion design revision and generates a report.	FPGA prototype design and HardCopy Companion Revision	Compilation of both revisions must be complete
Generate HardCopy Handoff Report	Generate a report containing important design information files and messages generated by the Quartus II compile.	FPGA prototype design and HardCopy Companion Revision	■ Compilation of both revisions must be complete ■ Compare HardCopy Companion Revisions must have been executed
Archive HardCopy Handoff Files	Generate a Quartus II Archive File (**.qar**) specifically for submitting the design to Altera's HardCopy Design Center.	HardCopy Companion Revision	■ Compilation of both revisions must be completed ■ Compare HardCopy Companion Revisions must have been executed ■ Generate HardCopy Handoff Report must have been executed
HardCopy Advisor	Open an Advisor, similar to the Resource Optimization Advisor, helping you through the steps of creating a HardCopy project.	FPGA prototype design and HardCopy Companion Revision	None
HardCopy Design Readiness Check	Generates a reports with the design's settings, I/O check, PLL, and RAM usage checks.	FPGA prototype design and HardCopy Companion Revision.	None

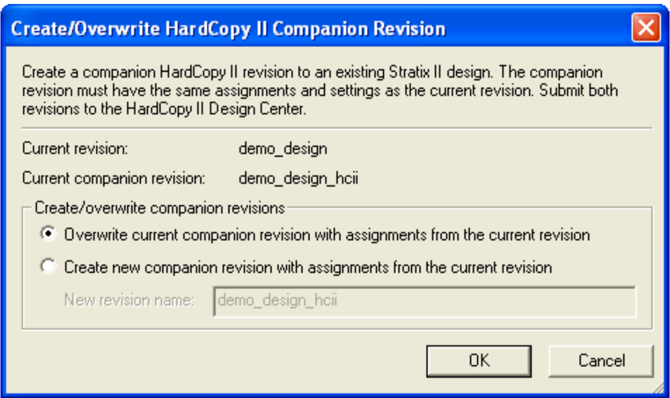
Companion Revisions

You can create multiple revisions of both the FPGA and the HardCopy device. For example, if your initial FPGA revision is called *top* and the corresponding HardCopy II revision is *top_hcii*, you could create another FPGA revision, *top_fpga*, and the corresponding HardCopy II revision would be *top_fpga_hcii*. The Quartus II software creates specific HardCopy design revisions of the project in conjunction to the regular project revisions. These parallel design revisions for HardCopy devices are called companion revisions.

Although you can create multiple project revisions, Altera recommends that you maintain only one FPGA revision once you have created the HardCopy companion revision.

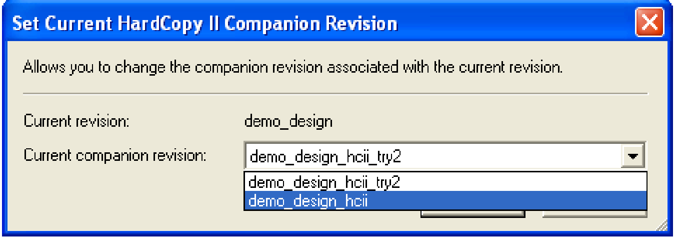
When you have successfully compiled your FPGA prototype, you can create a HardCopy companion revision of your design and proceed with compiling the HardCopy companion revision. To create a companion revision, on the Project menu, point to **HardCopy Utilities** and click **Create/Overwrite HardCopy Companion Revision**. Use the dialog box to create a new companion revision or overwrite an existing companion revision (Figure 4–10).

Figure 4–10. Create or Overwrite HardCopy Companion Revision



You can associate only one FPGA revision to one HardCopy companion revision. If you created more than one revision or more than one companion revision, set the current companion for the revision you are working on. On the Project menu, point to **HardCopy Utilities** and click **Set Current HardCopy Companion Revision** (Figure 4–11).

Figure 4–11. Set Current HardCopy Companion Revision

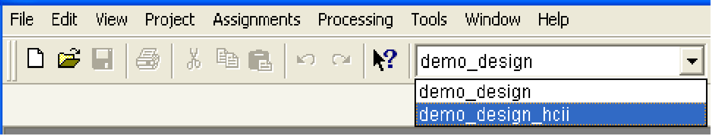


Compiling the HardCopy Companion Revision

The Quartus II software allows you to compile your HardCopy design with preliminary timing information. The timing constraints for the HardCopy companion revision can be the same as the FPGA design used to create the revision. The Quartus II software contains preliminary timing models for HardCopy devices and you can gauge how much performance improvement you can achieve in the HardCopy device compared to the FPGA. Altera verifies that the HardCopy Companion Device timing requirements are met in Altera's HardCopy Design Center.

After you create your HardCopy companion revision from your compiled FPGA design, select the companion revision in the Quartus II software design revision pull-down list (Figure 4–12) or from the **Revisions** list. Compile the HardCopy companion revision. After the Quartus II software compiles your design, you can perform a comparison check of the HardCopy companion revision to the FPGA prototype revision.

Figure 4–12. Changing Current Revision



Comparing HardCopy and FPGA Companion Revisions

Altera uses the companion revisions in a single Quartus II project to maintain compatibility between the FPGA and HardCopy ASIC. This methodology allows you to design with one set of RTL code to be used in both the FPGA and HardCopy ASIC, guaranteeing functional equivalency.

When making changes to companion revisions, use the **Compare HardCopy Companion Revisions** command to ensure that your design matches your HardCopy design functionality and compilation settings. To compare companion revisions, on the Project menu, point to **HardCopy Utilitie**s and click **Compare HardCopy Companion Revisions**.

You must perform this comparison after both the FPGA and HardCopy designs are compiled to hand off the design to Altera's HardCopy Design Center.

The Comparison Revision Summary is found in the Compilation Report and identifies where assignments were changed between revisions or if there is a change in the logic resource count due to different compilation settings.

Generate a HardCopy Handoff Report

To submit a design to Altera's HardCopy Design Center, you must generate a HardCopy Handoff Report providing important information about the design that you want Altera's HardCopy Design Center to review. To generate the HardCopy Handoff Report, you must:

- Successfully compile both FPGA and HardCopy revisions of your design
- Successfully run the **Compare HardCopy Companion Revisions** command

After you generate the HardCopy Handoff Report, you can archive the design using the **Archive HardCopy Handoff Files** command described in "Archive HardCopy Handoff Files" on page 4–21.

Archive HardCopy Handoff Files

The last step in the HardCopy design methodology is to archive the HardCopy project for submission to Altera's HardCopy Design Center for the HardCopy back-end. The **Archive HardCopy Handoff** command creates a different **.qar** file than the standard Quartus II project archive utility generates. This archive contains only the necessary data from the Quartus II project required to implement the design in Altera's HardCopy Design Center.

To use the **Archive HardCopy Handoff Files command**, you must complete the following:

- Compile both the FPGA and HardCopy revisions of your design
- Run the **Compare HardCopy Companion Revisions** command
- Generate the HardCopy Handoff Report

To select this option, on the Project menu, point to **HardCopy Utilities** and click **Archive HardCopy Handoff Files**.

HardCopy Advisor

The HardCopy Advisor provides the list of tasks you should follow to develop your FPGA prototype and your HardCopy design. To open the HardCopy Advisor, on the Project menu, point to **HardCopy Utilities** and click **HardCopy Advisor**. The following tasks highlight the checkpoints that the HardCopy Advisor reviews. These tasks include the major checkpoints in the design process, but they do not include show every step in the process for completing your FPGA and HardCopy designs:

1. Select an FPGA device.
2. Select a HardCopy device.
3. Turn on the Design Assistant.
4. Set up timing constraints.
5. Check for incompatible assignments.
6. Compile and check the FPGA design.
7. Create or overwrite the companion revision.
8. Compile and check the HardCopy companion results.
9. Compare companion revisions.
10. Generate a Handoff Report.
11. Archive Handoff Files and send to Altera.

The HardCopy Advisor shows the necessary steps that pertain to your currently selected device. The Advisor shows a slightly different view for a design with FPGA selected as compared to a design with HardCopy selected.

In the Quartus II software, you can start designing with the HardCopy device selected first, and build an FPGA companion revision second. When you use this approach, the HardCopy Advisor task list adjusts automatically to guide you from HardCopy development through FPGA prototyping, then completes the comparison archiving and handoff to Altera.

When your design uses the FPGA as your starting point, Altera recommends following the Advisor guidelines for your FPGA until you complete the prototype revision.

When the FPGA design is complete, create and switch to your HardCopy companion revision and follow the Advisor steps shown in that revision until you are finished with the HardCopy revision and are ready to submit the design to Altera for the HardCopy back-end.

Each category in the HardCopy Advisor list has an explanation of the recommended settings and constraints, as well as quick links to the features in the Quartus II software that are required for each section. The HardCopy Advisor displays:

- A green check box when you have successfully completed one of the steps
- A yellow caution sign for steps that must be completed before submitting your design to Altera for HardCopy development
- An information callout for items you must verify

Selecting an item within the HardCopy flow menu provides a description of the task and recommended action. The view in the HardCopy Advisor can vary depending on the device you select.

Figure 4–13 shows the HardCopy Advisor with the FPGA device selected.

Figure 4–13. HardCopy Advisor with FPGA Selected

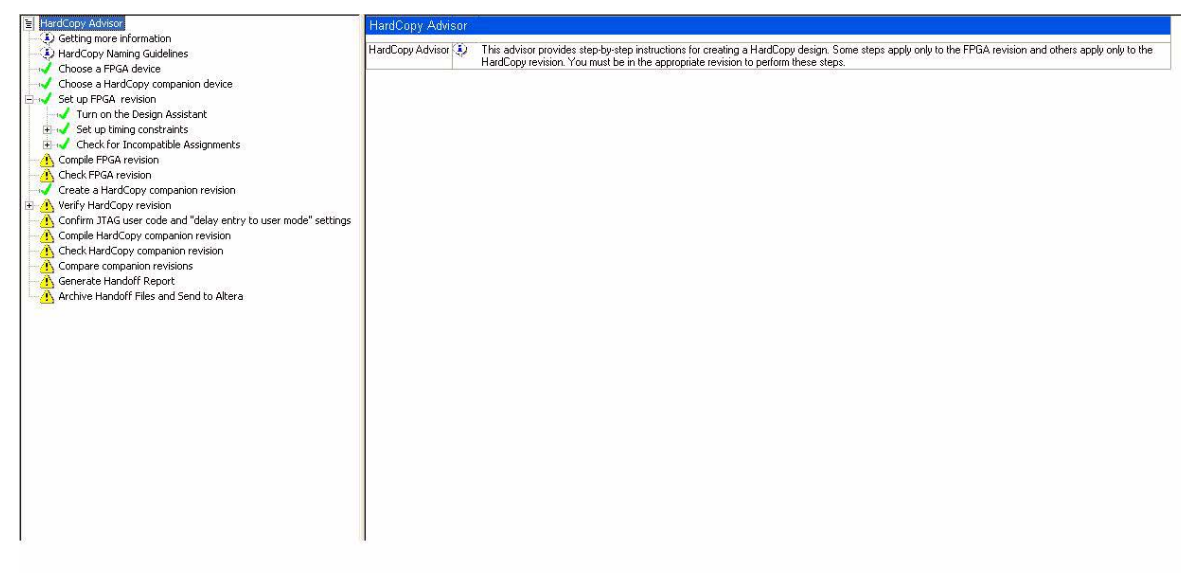
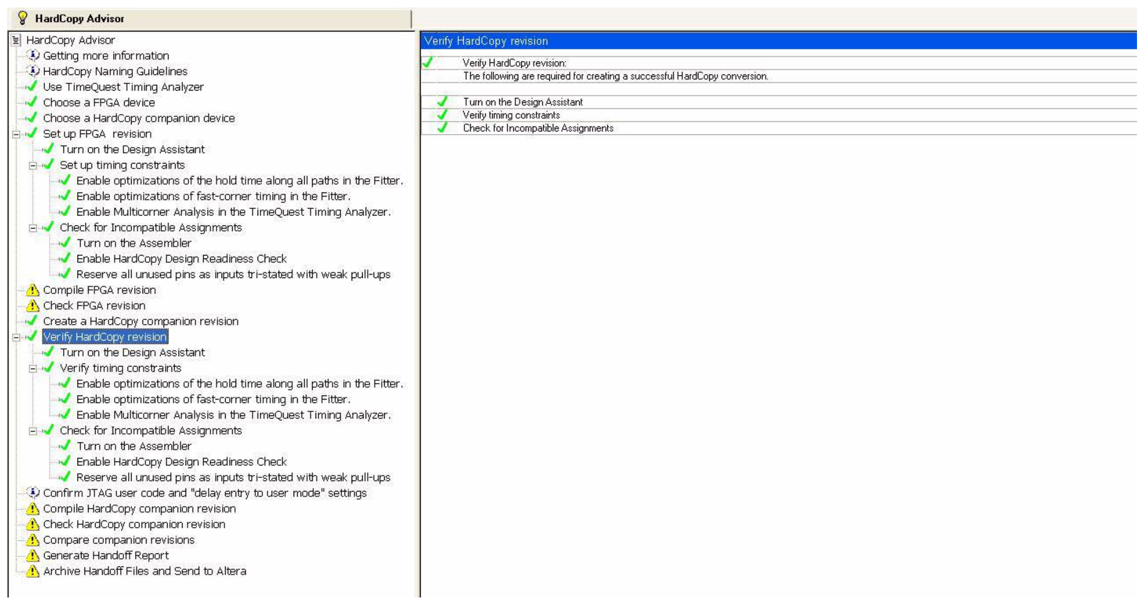


Figure 4–14 shows the HardCopy Advisor with the HardCopy device selected.

Figure 4–14. HardCopy Advisor with HardCopy Device Selected



HardCopy Design Readiness Check

Beginning in the Quartus II software version 7.2, the HardCopy Design Readiness Check (HCDRC) is available as one of the processing steps in the default compilation of either the FPGA or the HardCopy flow. This feature checks issues that must be addressed prior to handing off the HardCopy design to Altera's HardCopy Design Center for the HardCopy back-end process. This is different from the user-driven approach in HardCopy Advisor, in which you must manually open the Advisor to check for any violations.

The implemented checking in the HCDRC for the Quartus II software version 7.2 is only I/O-related. Beginning in the Quartus II software version 8.0, the checks have been extended to include other logic checks such as PLL, RAM, and Setting checks (**Global Setting**, **Instance Setting**, and **Operating Setting**).

Execution of HardCopy Design Readiness Check

The tool can be turned on through the **.qsf** file, as follows:

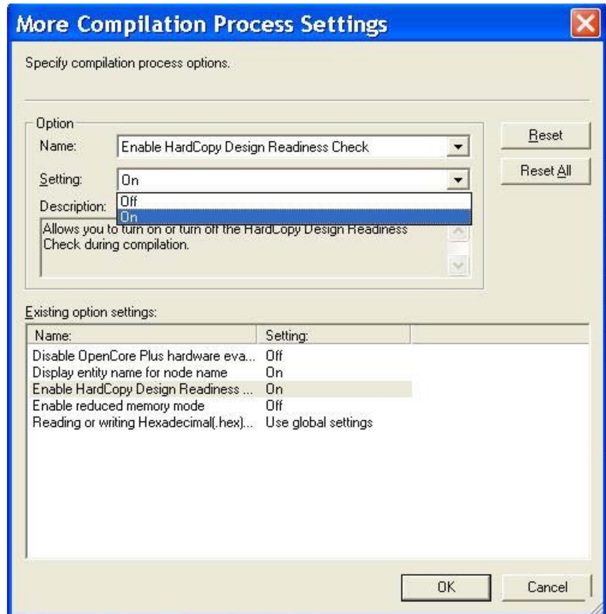
```
set_global_assignment -name \
FLOW_HARDCOPY_DESIGN_READINESS_CHECK ON
```

```
set_global_assignment -name \
FLOW_HARDCOPY_DESIGN_READINESS_CHECK OFF
```

The tool can also be turned on through the GUI, as shown in Figure 4–15.

☞ The tool is On by default.

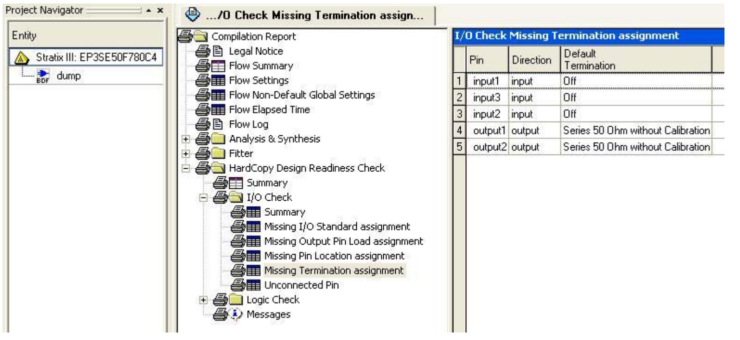
Figure 4–15. HardCopy Design Readiness Check through the GUI



Stratix III Support

Beginning in the Quartus II software version 8.0, the HCDRC enables support for Stratix III devices. This includes automated execution of HCDRC in the Stratix III design flow. However, users must select a HardCopy III companion first for HCDRC to run during the compilation. Refer to Figure 4–16.

Figure 4–16. Stratix III Support in HardCopy Design Readiness Check



All checks are the same as for other families. If the check is specific to Stratix III devices only, HCDRC dynamically runs the check exclusive to the Stratix III revision.

Setting Check

Beginning in Quartus II version 8.0, HCDRC provides the Setting Check report section. The report panels in this category are the setting checks from the HandOff Report. Setting Check consists of the following three sections.

Summary

The Summary section displays the number of settings that do not follow the recommendations. One of the following messages is displayed:

`<number> global setting(s) do not meet recommendation. Please review the recommendation and do appropriate correction as it may affect the result of the migration to HardCopy.`

or

`<number> instance setting(s) do not meet recommendation. Please review the recommendation and do appropriate correction as it may affect the result of the migration to HardCopy.`

Global Setting

The settings check in this section only displays recommendations for global settings. Global settings that currently have a different value than the recommended value are highlighted in red.

Instance Setting

This section is the same as **Global Setting**, but only checks for instances assignments.

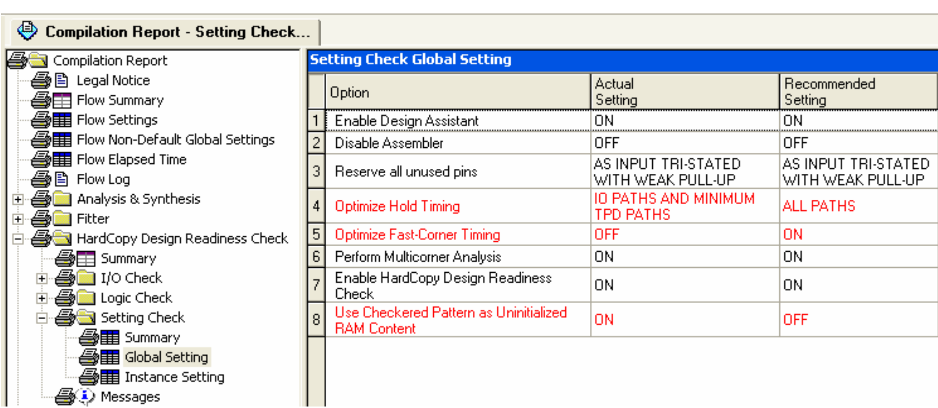
Operating Setting

In this section, checks related to the recommended operating settings for the FPGA and the HardCopy device are reported.

The Operating Setting check is primarily applicable to Stratix III devices used as prototype FPGAs because HardCopy III devices only support 0.9-V core voltage, whereas the Stratix III devices support both 1.1 V and 0.9 V.

Figure 4–17 shows the **Setting Check** category for HCDRC in the Quartus II software version 8.0.

Figure 4–17. Setting Check



Setting Check also includes checking for illegal assignments in the HardCopy design flow. The illegal assignments checks are shown in Example 4–1.

Example 4–1. Illegal Assignment Checks

```
USE_CHECKERED_PATTERN_AS_UNINITIALIZED_RAM_CONTENT ON (1)
SIGNAL_PROBE_ENABLE ON|OFF
SIGNAL_PROBE_SOURCE ON|OFF (2)
```

Notes to Example 4–1:

(1) Refer to the section "RAM Usage Check" on page 4–28.
(2) SignalProbe is not supported in HardCopy ASICs.

I/O Check

The HCDRC I/O Check ensures that you have assigned location assignments for the pins, I/O Standard, current strength assignment, output pin load assignment, termination assignments, and also checks for any unconnected pins. The tool issues a Warning if you have not specified the assignment for the I/O check.

For example, for missing I/O Standard assignments, the HCDRC issues the following warning:

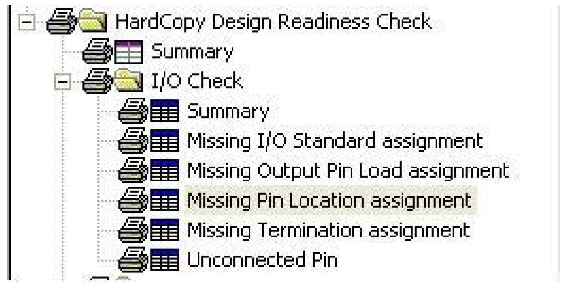
```
5 pin(s) have no explicit I/O Standard assignments provided in
the setting file and default values are being used. Please add a
specific I/O Standard assignment for these pins.
```

Input Pin Placement for Global and Regional Clock

Due to the difference in the interconnect delays between the FPGA and HardCopy, the use of non-primary clock inputs as clock inputs in a design can cause timing closure to be a problem when migrating the FPGA to HardCopy. The Input Pin Placement for Global and Regional Clock check informs you of the problem before finalizing the pin location, so that any clock inputs can be moved to the primary clock input.

This check lists all the pins that drive the global or regional clock but are not placed in a dedicated clock pad. All pins are required to have manual location assignments. This is highlighted prior to this check. See Figure 4–18.

Figure 4–18. I/O Check in the HardCopy Design Readiness Check



The following message appears in the message panel during compilation and also appears in the I/O Check Summary:

```
<number> pin(s) drives global or regional clock, but is not
placed in a dedicated clock pin position. Clock insertion delay
will be different between FPGA and HardCopy companion revisions
because of differences in local routing interconnect delays.
```

PLL Usage Check

There is a new dedicated checking category for PLLs in the HCDRC. The report folder that appears in the UI report is PLL Usage Check. This is for requirements and violations checks relating to PLL usage.

PLL Real-Time Reconfigurable Check

This check highlights the PLLs that do not have PLL reconfiguration. PLL reconfiguration allows fine tuning of the PLLs in the design after manufacturing.

The following message appears in the message panel during compilation and also appears in the Logic Check Summary:

```
<number> PLL(s) don't have real time reconfiguration. It is
highly recommended that each PLL to have PLL reconfiguration
for designs migrating to HardCopy.
```

There is a table listing the PLL elements that do not have PLL reconfiguration.

PLL Clock Outputs Driving Multiple Clock Network Types Check

This check is derived from the Design Assistant rule check for HardCopy (H102). It lists all PLL instances in the current design that have clock outputs driving multiple clock network types. The following message is displayed if the tool detects violations of this type:

```
Found <number> PLL(s) with clock outputs that drives multiple
clock network types.
```

PLL with No Compensation Mode Check

This check list all PLLs that are in "No compensation" operating mode. This setting is not recommended for a design migrating to a HardCopy device. This is due to the differences in the clock networks and the clock delays between FPGA and HardCopy devices.

The following warning message appears during compilation when a PLL is in a "No compensation mode":

```
<number> PLL(s) is operating in a "No compensation" mode.
```

PLL with Normal or Source Synchronous Mode Feeding Output Pin Check

When a PLL is directly feeding an output pin, it must be set to **Zero Delay Buffer** operating mode. However, if a PLL mode is set either in normal compensation mode or source synchronous mode, a warning message is printed during compilation.

During the runtime of HC Ready, the following warning message appears:

```
<number> PLL(s) is in normal or source synchronous mode that is
not fully compensated because it feeds an output pin -- only
PLLs in zero delay buffer mode can fully compensate output pins.
```

RAM Usage Check

HardCopy Series devices do not support initialized RAM blocks upon power-up. However, you can use the ALTMEM_INIT megafunction to initialize the RAMs in your design.

The ALTMEM_INIT megafunction initializes the RAM of a HardCopy Series device with the content of a ROM.

For more information about the ALTMEM_INIT megafunction, refer to the *RAM Initializer (ALTMEM_INIT) Megafunction User Guide*.

In HardCopy Series devices, the RAMs power up uninitialized. In the RAM Usage Check, the HCDRC tool checks to see if there are any RAMs that are initialized using a Memory Initialization File (**.mif**). Any RAM that has a **.mif** file is listed in a table with the following compilation warning message:

```
<number> RAM(s) have Memory Initialization File (MIF). HardCopy
devices do not allow initialized RAM. Please ensure that no RAM
is initialized by a MIF file.
```

Initialized Memory Dependency Testing

Beginning in Quartus II 7.2, there is an option added to the Assembler for the FPGA revision that allows you to write out an FPGA programming file with an initialized checkerboard pattern for memory contents of M4K memories. This option should not be part of your regular Stratix II revision used to migrate to the HardCopy II revision, as it creates irreconcilable revision differences between the Stratix II and HardCopy II designs (the HardCopy hand-off cannot physically have any initialized memory content. Only use this option on a parallel copy of your compiled Stratix II design that you wish to test on your board.

The recommended usage of this feature is as follows:

1. Compile your completed Stratix II design revision to use for prototype testing. This is the revision you should eventually create your HardCopy II companion revision from. On the Project menu, point to **HardCopy II Utilities** and click **Create/Overwrite HardCopy II Companion Revision**.
2. Complete the HardCopy II companion revision creation, and then compile, compare, and hand-off archive generation as usual for your design.
3. After completing the HardCopy II hand-off archive generation, switch back to your Stratix II revision, and on the Quartus II Project menu, select **Revisions** and **Create** to create a copy of your Stratix II revision.
4. In the **Create Revision** dialog box, create a new revision copy of your Stratix II design revision, selecting both the **Copy database** and **Set as current revision** check box options. This copies your Stratix II revision and sets the new revision as the current open revision in the Quartus II software.
5. On the Assignments menu, point to **Settings**, click **Assembler**, and enable **Use checkered pattern as uninitialized RAM content**, or edit the revision **.qsf** file and add the following line:

   ```
   set_global_assignment -name
   USE_CHECKERED_PATTERN_AS_UNINITIALIZED_RAM_CONTENT ON
   ```

6. Run the Assembler in the Stratix II revision to generate a new programming file for your FPGA.
7. Test the new programming file in your prototype environment to determine if your design has a dependency for initial FPGA RAM contents being initialized with zeros after power-up and configuration.

Because it is only a simple checkerboard pattern used for testing, the checkerboard patterns written into the RAMs for the new programming file are not guaranteed to detect all cases of zero-initialized RAM content dependency, but this is one way to look for it. Some designs might only be looking to detect one bit as zero (for example, the LSB of a memory word) so this is not foolproof. If a full RAM word line is expected as zeros at startup, this checkerboard pattern test exposes it.

Performing ECOs with Quartus II Engineering Change Management with the Chip Planner

As designs grow larger and larger in density, analyzing designs for performance, routing congestion, logic placement, and executing Engineering Change Orders (ECOs) becomes critical. In addition to design analysis, you can use various bottom-up and top-down flows to implement and manage the design. This becomes difficult to manage, because ECOs are often implemented as last minute changes to your design.

With the Altera Chip Planner tool, you can shorten the design cycle time significantly. When changes are made to your design as ECOs, you do not have to perform a full compilation in the Quartus II software. Instead, you make changes directly to the post place-and-route netlist, generate a new programming file, test the revised design by performing a gate-level simulation and timing analysis, and proceed to verify the fix on the system. When the fix has been verified on the FPGA, switch to the HardCopy revision, apply the same ECOs, run the timing analyzer and assembler, perform a revision compare, and then run the HardCopy Netlist Writer for design submission.

There are three scenarios from a migration point of view:

- There are changes which can map one-to-one (that is, the same change can be implemented on each architecture—FPGA and HardCopy).
- There are changes that must be implemented differently on the two architectures to achieve the same result.
- There are some changes that cannot be implemented on both architectures.

The following sections outline the methods for migrating each of these types of changes.

Migrating One-to-One Changes

One-to-one changes are implemented using identical commands in both architectures. In general, such changes include those that affect only I/O cells or PLL cells. Some examples of one-to-one changes are changes such as creating, deleting, or moving pins, changing pin or PLL properties, or changing pin connectivity (provided the source and destination of the connectivity changes are I/Os or PLLs). These can be implemented identically on both architectures.

If such changes are exported to Tcl, a direct reapplication of the generated Tcl script (with a minor text edit) on the companion revision should implement the appropriate changes as follows:

- Export the changes from the Change Manager to Tcl.
- Open the generated Tcl script, change the line `project_open <project> -revision <revision>` to refer to the appropriate companion revision.
- Apply the Tcl script to the companion revision.

The following is a partial list of examples of this type:

- I/O creation, deletion, and moves
- I/O property changes (for example, I/O standards, delay chain settings, and so forth)
- PLL property changes
- Connectivity changes between non-LCELL_COMB atoms (for example, PLL to I/O, DSP to I/O, and so forth)

Migrating Changes that Must be Implemented Differently

Some changes must be implemented differently on the two architectures. Changes affecting the logic of the design can fall into this category. Examples are LUTMASK changes, `LC_COMB/HSADDER` creation and deletion, and connectivity changes not covered in the previous section.

Another example of this would be to have different PLL settings for the FPGA and the HardCopy revisions.

For more information about how to use different PLL settings for the FPGA and HardCopy Devices, refer to *AN 432: Using Different PLL Settings Between Stratix II and HardCopy II Devices*.

Table 4–4 summarizes suggested implementation for various changes.

Table 4–4. Implementation Suggestions for Various Changes (Part 1 of 2)

Change Type	Suggested Implementation
LUTMASK changes	Because a single FPGA atom can require multiple HardCopy II atoms to implement, it might be necessary to change multiple HardCopy II atoms to implement the change, including adding or modifying connectivity
Make/Delete LC_COMB	If you are using a FPGA `LC_COMB` in extended mode (7-LUT) or are using a SHARE chain, you must create multiple atoms to implement the same logic functions in HardCopy. Additionally, the placement of the `LC_COMB` cell has no meaning in the companion revision as the underlying resources are different.

Table 4–4. Implementation Suggestions for Various Changes (Part 2 of 2)

Change Type	Suggested Implementation
Make/Delete LC_FF	The basic creation and deletion is the same on both architectures. However, as with `LC_COMB` creation and deletion, the location of an `LC_FF` in a HardCopy revision has no meaning in the FPGA revision, and vice versa.
Editing Logic Connectivity	Because a `LCELL_COMB` atom might have to be broken up into several HardCopy `LCELL_COMB` atoms, the source or destination ports for connectivity changes might have to be analyzed to properly implement the change in the companion revision.

Changes that Cannot be Migrated

A small set of changes cannot be implemented in the other architecture because they do not make sense in the other architecture. The best example of this occurs when moving logic in a design; because the logic fabric is different between the two architectures, locations in the FPGA make no sense in HardCopy, and vice versa.

Overall Migration Flow

This section outlines the migration flow and the suggested procedure for implementing changes in both revisions to ensure a successful Revision Compare such that the design can be submitted to Altera's HardCopy Design Center.

Preparing the Revisions

The general procedure for migrating changes between devices is the same, whether going from the FPGA to HardCopy or vice versa. The major steps are as follows:

1. Compile the design on the initial device.
2. Migrate the design from the initial device to the target device in the companion revision.
3. Compile the companion revision.
4. Perform a Revision Compare operation. The two revisions should pass the Revision Compare.

If testing identifies problems requiring ECO changes, equivalent changes can be applied to both FPGA and HardCopy revisions, as described in the following section.

Applying ECO Changes

The general flow for applying equivalent changes in companion revisions is as follows:

1. Make changes in one revision using the Chip Planner tools (Chip Planner, Resource Property Editor, and Change Manager), then verify and export these changes. The procedure for doing this is as follows:
   a. Make changes using the Chip Planner tool.
   b. Perform a netlist check using the **Check and Save All Netlist Changes** command.

c. Verify correctness using timing analysis, simulation, and prototyping (FPGA only). If more changes are required, repeat steps a and b.

d. Export change records from the Change Manager to Tcl scripts, or **.csv** or **.txt** file formats.

This exported file is used to assist in making the equivalent changes in the companion revision.

2. Open the companion revision in the Quartus II software.
3. Using the exported file, manually reapply the changes using the Chip Planner tool.

   As stated previously, some changes can be reapplied directly to the companion revision (either manually or by applying the Tcl commands), while others require some modifications.

4. Run the **Compare HardCopy Revision** command. The revisions should match.
5. Verify the correctness of all changes (you might have to run timing analysis).
6. Run the **HardCopy Assembler** command and the **HardCopy Netlist Writer** command for design submission along with handoff files.

The Tcl command for running the HardCopy Assembler is as follows:

```
execute_module -tool asm -args "--read_settings_files=off --
write_settings_files=off"
```

The Tcl command for the HardCopy Netlist Writer is as follows:

```
execute_module -tool cdb \
       -args "--generate_hardcopy_files"\
```

For more information about using Chip Planner, refer to the *Quartus II Engineering Change Management with Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Formal Verification of FPGA and HardCopy Revisions

Third-party formal verification software, Cadence Encounter Conformal verification software, is used for FPGA and HardCopy families, as well as several other Altera device families.

The formal verification flow for HardCopy ASIC designs is a two-step process. First, you run formal verification on the FPGA netlist to ensure that the FPGA netlist matches the RTL. Second, within the Quartus II software, use the Revision Compare tool to ensure that the HardCopy implementation matches the FPGA.

While this flow is enabled, performing formal verification is not necessary due to the one-to-one mapping of logic between the Stratix series FPGA prototype and the HardCopy series ASIC.

To use the Conformal software with the Quartus II software project for your FPGA design revision, you must enable the EDA Netlist Writer. You must turn on the EDA Netlist Writer so it can generate the necessary netlist and command files required to run the Conformal software. To automatically run the EDA Netlist Writer during the compile of your FPGA revision, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, under **EDA Tool Settings**, select **Formal Verification**, and then in the **Tool name** list, select **Conformal LEC**.
3. Compile your FPGA and HardCopy design revisions.

The Quartus II EDA Netlist Writer produces the netlist for the FPGA when it is run on that revision. You can compare your FPGA post-compilation netlist to your RTL source code using the scripts generated by the EDA Netlist Writer.

After both the FPGA and HardCopy revisions have been compiled, the Revision Compare tool can be run to ensure that the HardCopy implementation matches the FPGA.

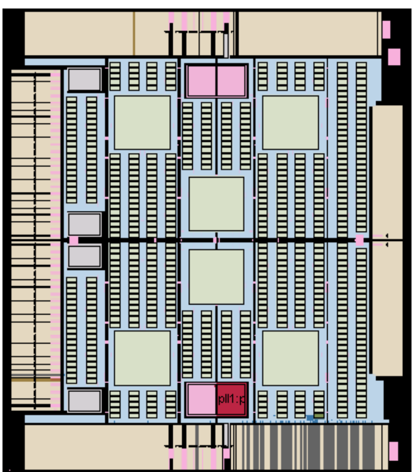
For more information about using the Cadence Encounter Conformal verification software, refer to the *Cadence Encounter Conformal Support* chapter in volume 3 of the *Quartus II Handbook*.

HardCopy Floorplan View

The Quartus II software displays the floorplan and placement of your HardCopy companion revision. This floorplan shows the preliminary placement and connectivity of all I/O pins, PLLs, memory blocks, HCell macros, and DSP HCell macros. Congestion mapping of routing connections can be viewed using the **Layers Setting** dialog box (in the View menu) settings. This is useful in analyzing densely packed areas of your floorplan that can reduce the peak performance of your design. Altera's HardCopy Design Center verifies final HCell macro timing and placement to guarantee timing closure is achieved.

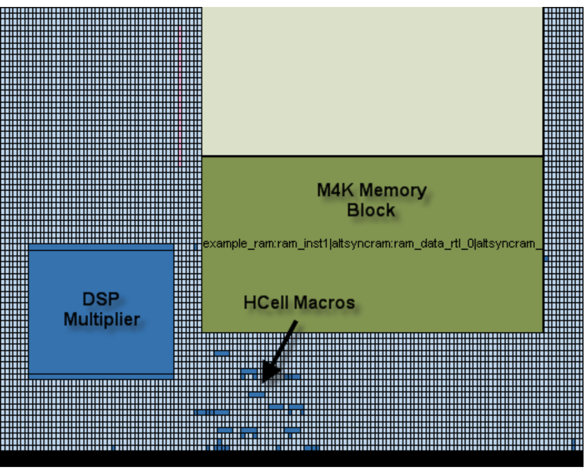
Figure 4–19 shows an example of the HC230F1020 device floorplan.

Figure 4–19. HC230F1020 Device Floorplan



In this small example design, the logic is placed near the bottom edge. You can see the placement of a DSP block constructed of HCell Macros, various logic HCell Macros, and an M4K memory block. A labeled close-up view of this region is shown in Figure 4–20.

Figure 4–20. Close-Up View of Floorplan



Altera's HardCopy Design Center performs final placement and timing closure on your HardCopy design based on the timing constraints provided in the FPGA design.

For more information about Altera's HardCopy Design Center process, refer to the respective HardCopy Series device handbook on the Altera website.

Legacy HardCopy Device Support

Altera HardCopy devices provide a comprehensive alternative to ASICs. HardCopy ASICs offer a complete solution from prototype to high-volume production, and maintain the powerful features and high-performance architecture of their equivalent FPGAs with the programmability removed. You can use the Quartus II design software to design HardCopy devices in a manner similar to the traditional ASIC design flow, and you can prototype with Altera's high density Stratix FPGAs before seamlessly migrating to the corresponding HardCopy device for high-volume production.

HardCopy ASICs provide the following key benefits:

- Improves performance, on the average, by 40% over the corresponding -6 speed grade FPGA device
- Lowers power consumption, on the average, by 40% over the corresponding FPGA
- Preserves the FPGA architecture and features and minimizes risk
- Guarantees first-silicon success through a fast and predictable turnkey backend process

Altera's Quartus II software has built-in support for HardCopy Stratix devices. The HardCopy design flow in Quartus II software offers the following advantages:

- Unified design flow from prototype to production
- Performance estimation of the HardCopy Stratix device allows you to design systems for maximum throughput
- Easy-to-use and inexpensive design tools from a single vendor
- An integrated design methodology that enables system-on-a-chip designs

The next sections discuss the following topics:

- How to design HardCopy Stratix and HardCopy APEX ASICs using the Quartus II software
- An explanation of what the HARDCOPY_FPGA_PROTOTYPE devices are and how to target designs to these devices
- Performance and power estimation of HardCopy Stratix devices
- How to generate the HardCopy design database for submitting HardCopy Stratix designs to Altera's HardCopy Design Center

Features

Beginning in Quartus II software version 4.2, the Quartus II software contains several powerful features that facilitate design of HardCopy Stratix devices:

- **HARDCOPY_FPGA_PROTOTYPE Devices**

  These are virtual Stratix FPGA devices with features identical to HardCopy Stratix devices. You must use these FPGA devices to prototype your designs and verify the functionality in silicon.

- **HardCopy Timing Optimization Wizard**

  Using this feature, you can target your design to HardCopy Stratix devices, providing an estimate of the design's performance in a HardCopy Stratix device.

- **HardCopy Stratix Floorplans and Timing Models**

  The Quartus II software supports post-migration HardCopy Stratix device floorplans and timing models and facilitates design optimization for design performance.

- **Placement Constraints**

  Location and LogicLock constraints are supported at the HardCopy Stratix floorplan level to improve overall performance.

- **Improved Timing Estimation**

  Beginning with version 4.2, the Quartus II software determines routing and associated buffer insertion for HardCopy Stratix designs, and provides the Timing Analyzer with more accurate information about the delays than was possible in previous versions of the Quartus II software. The **.qar** file automatically receives buffer insertion information, which greatly enhances the timing closure process in the back-end migration of your HardCopy Stratix device.

- **Design Assistant**

  This feature checks your design for compliance with all HardCopy device design rules and quickly establishes a seamless migration path.

- **HardCopy Files Wizard**

  This wizard allows you to deliver the design database and all the deliverables required for migration to Altera. This feature is used for HardCopy Stratix devices.

The HardCopy Stratix PowerPlay Early Power Estimator is available on the Altera website at www.altera.com.

HARDCOPY_FPGA_PROTOTYPE, HardCopy Stratix, and Stratix Devices

You must use the HARDCOPY_FPGA_PROTOTYPE virtual devices available in the Quartus II software to target your designs to the actual resources and package options available in the equivalent post-migration HardCopy Stratix device. The programming file generated for the HARDCOPY_FPGA_PROTOTYPE can be used in the corresponding Stratix FPGA device.

The purpose of the HARDCOPY_FPGA_PROTOTYPE is to guarantee seamless migration to HardCopy by making sure that your design only uses resources in the FPGA that can be used in the HardCopy device after migration. You can use the equivalent Stratix FPGAs to verify the design's functionality in-system, then generate the design database necessary to migrate to a HardCopy device. This process ensures the seamless migration of the design from a prototyping device to a production device in high volume. It also minimizes risk, assures samples in about eight weeks, and guarantees first-silicon success.

HARDCOPY_FPGA_PROTOTYPE devices are only available for HardCopy Stratix devices.

Table 4–5 compares HARDCOPY_FPGA_PROTOTYPE devices, Stratix devices, and HardCopy Stratix devices.

Table 4–5. Qualitative Comparison of HARDCOPY_FPGA_PROTOTYPE to Stratix and HardCopy Stratix Devices

Stratix Device	HARDCOPY_FPGA_PROTOTYPE Device	HardCopy Stratix Device
FPGA	Virtual FPGA	ASIC
FPGA	Architecture identical to Stratix FPGA	Architecture identical to Stratix FPGA
FPGA	Resources identical to HardCopy Stratix device	M-RAM resources different than Stratix FPGA in some devices
Ordered by Altera part number	Cannot be ordered; use the Altera Stratix FPGA part number	Ordered by Altera part number

Table 4–6 lists the resources available in each of the HardCopy Stratix devices.

Table 4–6. HardCopy Stratix Device Physical Resources

Device	LEs	ASIC Equivalent Gates (K) *(1)*	M512 Blocks	M4K Blocks	M-RAM Blocks	DSP Blocks	PLLs	Maximum User I/O Pins
HC1S25F672	25,660	250	224	138	2	10	6	473
HC1S30F780	32,470	325	295	171	2 *(2)*	12	6	597
HC1S40F780	41,250	410	384	183	2 *(2)*	14	6	615
HC1S60F1020	57,120	570	574	292	6	18	12	773
HC1S80F1020	79,040	800	767	364	6 *(2)*	22	12	773

Notes to Table 4–6:

(1) Combinational and registered logic do not include DSP blocks, on-chip RAM, or PLLs.

(2) The M-RAM resources for these HardCopy devices differ from the corresponding Stratix FPGA.

For a given device, the number of available M-RAM blocks in HardCopy Stratix devices is identical with the corresponding HARDCOPY_FPGA_PROTOTYPE devices, but can be different from the corresponding Stratix devices. Maintaining the identical resources between HARDCOPY_FPGA_PROTOTYPE and HardCopy Stratix devices facilitates seamless migration from the FPGA to the ASIC device.

For more information about HardCopy Stratix devices, refer to the *HardCopy Stratix Device Family Data Sheet* on the Altera website (www.altera.com).

The three devices, Stratix FPGA, HARDCOPY_FPGA_PROTOTYPE, and HardCopy device, are distinct devices in the Quartus II software. The HARDCOPY_FPGA_PROTOTYPE programming files are used in the Stratix FPGA for your design. The three devices are tied together with the same netlist, thus a single SRAM Object File (**.sof**) can be used to achieve the various goals at each stage. The same **.sof** file is generated in the HARDCOPY_FPGA_PROTOTYPE design, and is used to program the Stratix FPGA device, the same way that it is used to generate the HardCopy Stratix device, guaranteeing a seamless migration.

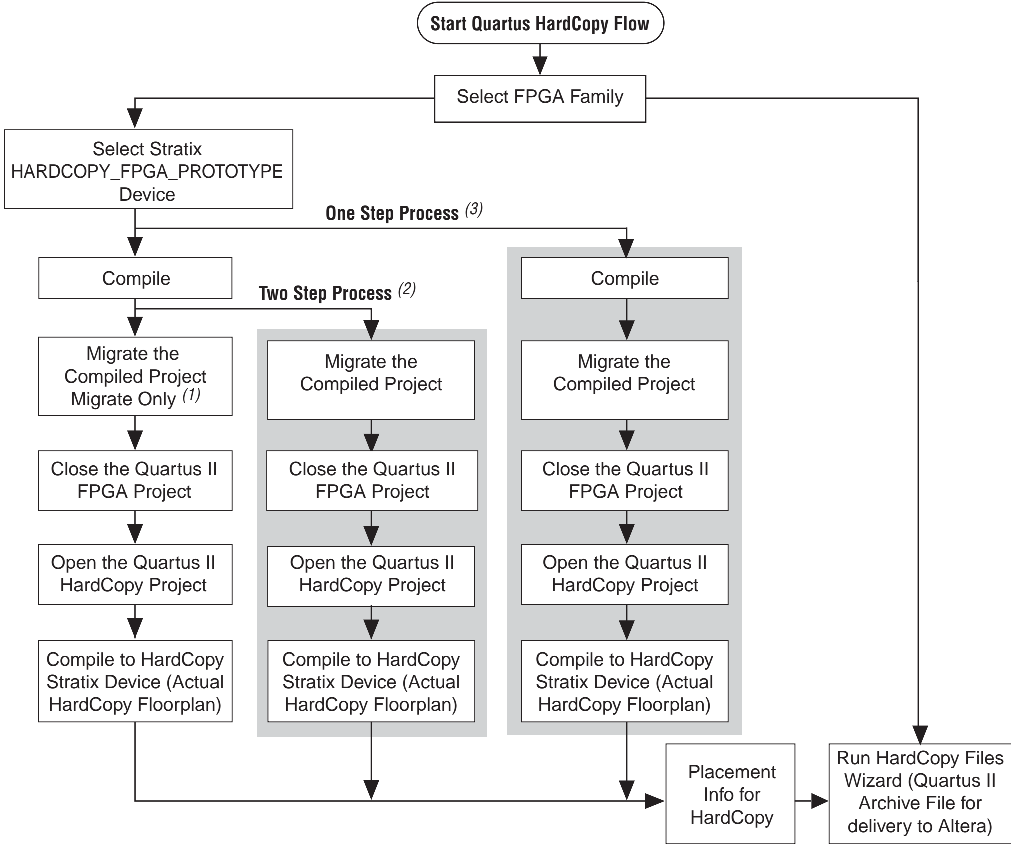
For more information about the **.sof** file and programming Stratix FPGA devices, refer to the *Programming and Configuration* chapter of the *Introduction to the Quartus II Software* manual.

HardCopy Design Flow

Figure 4–21 shows a HardCopy Stratix design flow diagram. The design steps are explained in detail in the following sections of this chapter. The HardCopy Stratix design flow utilizes the HardCopy Timing Optimization Wizard to automate the migration process into a one-step process. The remainder of this section explains the tasks performed by this automated process.

For a detailed description of the HardCopy Timing Optimization Wizard and HardCopy Files Wizard, refer to "HardCopy Timing Optimization Wizard" on page 4–40 and "Generating the HardCopy Design Database" on page 4–49.

Figure 4–21. HardCopy Stratix Design Flow Diagram



Notes to Figure 4–21:

(1) Migrate-Only Process: The displayed flow is completed manually.

(2) Two-Step Process: Migration and Compilation are done automatically (shaded area).

(3) One-Step Process: Full HardCopy Compilation. The entire process is completed automatically (shaded area).

The Design Flow Steps of the One-Step Process

The following sections describe each step of the full HardCopy compilation (the One Step Process), as shown in Figure 4–21.

Compile the Design for an FPGA

This step compiles the design for a HARDCOPY_FPGA_PROTOTYPE device and gives you the resource utilization and performance of the FPGA.

Migrate the Compiled Project

This step generates the Quartus II Project File (**.qpf**) and the other files required for HardCopy implementation. The Quartus II software also assigns the appropriate HardCopy Stratix device for the design migration.

Close the Quartus FPGA Project

Because you must compile the project for a HardCopy Stratix device, you must close the existing project which you have targeted your design to a HARDCOPY_FPGA_PROTOTYPE device.

Open the Quartus HardCopy Project

Open the Quartus II project that you created in the "Migrate the Compiled Project" step. The selected device is one of the devices from the HardCopy Stratix family that was assigned during that step.

Compile for HardCopy Stratix Device

Compile the design for a HardCopy Stratix device. After successful compilation, the Timing Analysis section of the Compilation Report shows the performance of the design implemented in the HardCopy device.

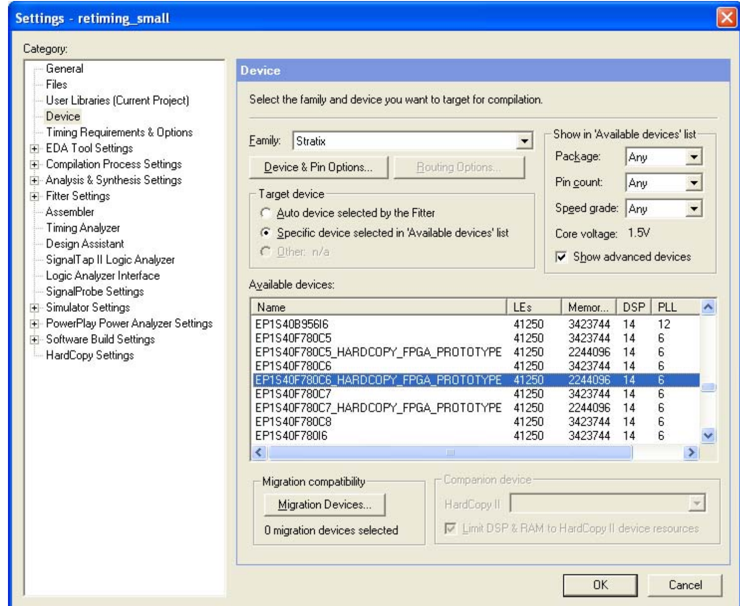
How to Design HardCopy Stratix Devices

This section describes the design process for a HardCopy Stratix device using the HARDCOPY_FPGA_PROTOTYPE as your initial selected device. To use the HardCopy Timing Optimization Wizard, you must first design with the HARDCOPY_FPGA_PROTOTYPE for the design to migrate to a HardCopy Stratix device.

To target a design to a HardCopy Stratix device in the Quartus II software, follow these steps:

1. If you have not yet done so, create a new project or open an existing project.
2. On the Assignments menu, click **Settings**. In the **Category** list, select **Device**.
3. On the **Device** page, in the **Family** list, select **Stratix**. Select the desired HARDCOPY_FPGA_PROTOTYPE device in the **Available Devices list** (Figure 4–22).

Figure 4–22. Selecting a HARDCOPY_FPGA_PROTOTYPE Device



By choosing the HARDCOPY_FPGA_PROTOTYPE device, all the design information, available resources, package option, and pin assignments are constrained to guarantee a seamless migration of your project to the HardCopy Stratix device. The netlist resulting from the HARDCOPY_FPGA_PROTOTYPE device compilation contains information about the electrical connectivity, resources used, I/O placements, and the unused resources in the FPGA device.

1. On the Assignments menu, click **Settings**. In the **Category** list, select **HardCopy Settings** and specify the input transition timing to be modeled for both clock and data input pins. These transition times are used in static timing analysis during back-end timing closure of the HardCopy device.
2. Add constraints to your HARDCOPY_FPGA_PROTOTYPE device, and on the Processing menu, click **Start Compilation** to compile the design.

HardCopy Timing Optimization Wizard

After you have successfully compiled your design in the HARDCOPY_FPGA_PROTOTYPE, you must migrate the design to the HardCopy Stratix device to get a performance estimation of the HardCopy Stratix device. This migration is required before submitting the design to Altera for the HardCopy Stratix device implementation. To perform the required migration, on the Project menu, point to **HardCopy Utilities** and click **HardCopy Timing Optimization Wizard**.

At this point, you are presented with the following three choices to target the designs to HardCopy Stratix devices (Figure 4–23):

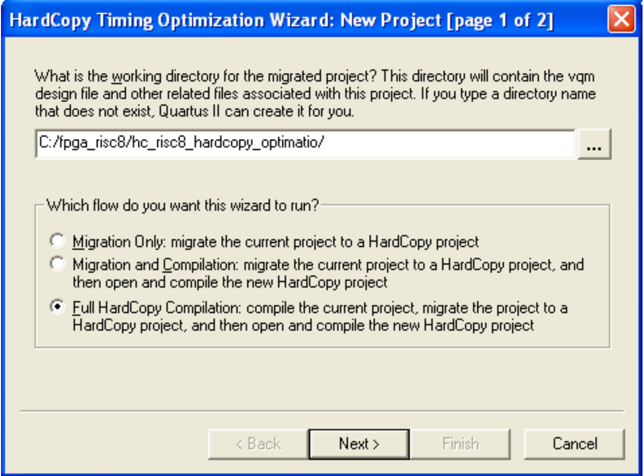
- **Migration Only**: You can select this option after compiling the HARDCOPY_FPGA_PROTOTYPE project to migrate the project to a HardCopy Stratix project.

You can now perform the following tasks manually to target the design to a HardCopy Stratix device. Refer to "Performance Estimation" on page 4–43 for additional information about how to perform these tasks.

- Close the existing project
- Open the migrated HardCopy Stratix project
- Compile the HardCopy Stratix project for a HardCopy Stratix device

- **Migration and Compilation**: You can select this option after compiling the project. This option results in the following actions:
  - Migrating the project to a HardCopy Stratix project
  - Opening the migrated HardCopy Stratix project and compiling the project for a HardCopy Stratix device
- **Full HardCopy Compilation**: Selecting this option results in the following actions:
  - Compiling the existing HARDCOPY_FPGA_PROTOTYPE project
  - Migrating the project to a HardCopy Stratix project
  - Opening the migrated HardCopy Stratix project and compiling it for a HardCopy Stratix device

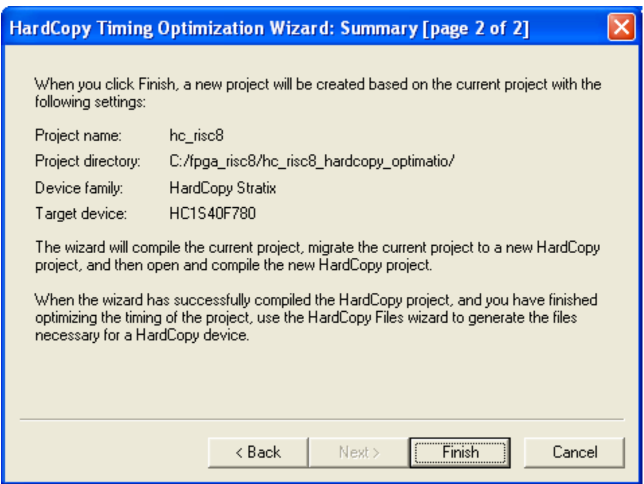
Figure 4–23. HardCopy Timing Optimization Wizard Options



The main benefit of the HardCopy Timing Wizard's three options is flexibility of the conversion process automation. The first time you migrate your HARDCOPY_FPGA_PROTOTYPE project to a HardCopy Stratix device, you might want to use **Migration Only**, and then work on the HardCopy Stratix project in the Quartus II software. As your prototype FPGA project and HardCopy Stratix project constraints stabilize and you have fewer changes, the **Full HardCopy Compilation** is ideal for one-click compiling of your HARDCOPY_FPGA_PROTOTYPE and HardCopy Stratix projects.

After selecting the wizard you want to run, the **HardCopy Timing Optimization Wizard: Summary** page shows you details about the settings you made in the wizard, as shown in Figure 4–24.

Figure 4–24. HardCopy Timing Optimization Wizard Summary Page



When either of the second two options in Figure 4–23 are selected (**Migration and Compilation** or **Full HardCopy Compilation**), designs are targeted to HardCopy Stratix devices and optimized using the HardCopy Stratix placement and timing analysis to estimate performance. For details about the performance optimization and estimation steps, refer to "Performance Estimation" on page 4–43. If the performance requirement is not met, you can modify your RTL source, optimize the FPGA design, and estimate timing until you reach timing closure.

Tcl Support for HardCopy Migration

To complement the GUI features for HardCopy migration, the Quartus II software provides the following command-line executables (which provide the Tcl shell to run the `--flow` Tcl command) to migrate the HARDCOPY_FPGA_PROTOTYPE project to HardCopy Stratix devices:

```
quartus_sh --flow migrate_to_hardcopy <project_name> [-c <revision>] ↵
```

This command migrates the project compiled for the HARDCOPY_FPGA_PROTOTYPE device to a HardCopy Stratix device:

```
quartus_sh --flow hardcopy_full_compile <project_name> [-c <revision>] ↵
```

This command performs the following tasks:

- Compiles the existing project for a HARDCOPY_FPGA_PROTOTYPE device.
- Migrates the project to a HardCopy Stratix project.
- Opens the migrated HardCopy Stratix project and compiles it for a HardCopy Stratix device.

Design Optimization and Performance Estimation

The HardCopy Timing Optimization Wizard creates the HardCopy Stratix project in the Quartus II software, where you can perform design optimization and performance estimation of your HardCopy Stratix device.

 © March 2009 Altera Corporation

Design Optimization

Beginning with version 4.2, the Quartus II software supports HardCopy Stratix design optimization by providing floorplans for placement optimization and HardCopy Stratix timing models. These features allow you to refine placement of logic array blocks (LABs) and optimize the HardCopy design further than the FPGA performance. Customized routing and buffer insertion done in the Quartus II software are then used to estimate the design's performance in the migrated device. The HardCopy device floorplan, routing, and timing estimates in the Quartus II software reflect the actual placement of the design in the HardCopy Stratix device, and can be used to see the available resources, and the location of the resources in the actual device.

Performance Estimation

Figure 4–25 illustrates the design flow for estimating performance and optimizing your design. You can target your designs to HARDCOPY_FPGA_PROTOTYPE devices, migrate the design to the HardCopy Stratix device, and get placement optimization and timing estimation of your HardCopy Stratix device.

In the event that the required performance is not met, you can:

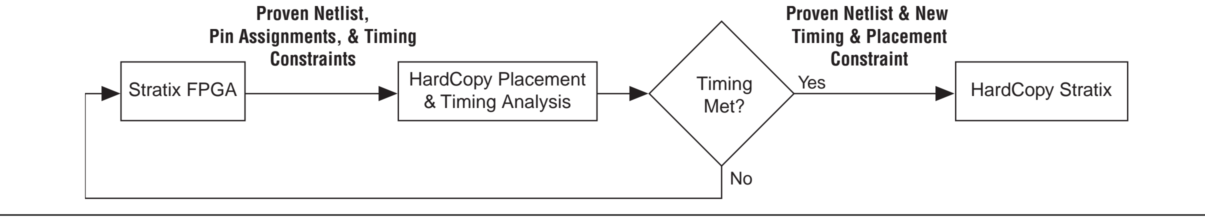
- Work to improve LAB placement in the HardCopy Stratix project.

or

- Go back to the HARDCOPY_FPGA_PROTOTYPE project and optimize that design, modify your RTL source code, repeat the migration to the HardCopy Stratix device, and perform the optimization and timing estimation steps.

On average, HardCopy Stratix devices are 40% faster than the equivalent -6 speed grade Stratix FPGA device. These performance numbers are highly design dependent, and you must obtain final performance numbers from Altera.

Figure 4–25. Obtaining a HardCopy Performance Estimation



To perform Timing Analysis for a HardCopy Stratix device, follow these steps:

1. Open an existing project compiled for a HARDCOPY_FPGA_PROTOYPE device.
2. On the Project menu, point to **HardCopy Utilities** and click **HardCopy Timing Optimization Wizard**.
3. Select a destination directory for the migrated project and complete the HardCopy Timing Optimization Wizard process.

On completion of the HardCopy Timing Optimization Wizard, the destination directory created contains the Quartus II project file, and all files required for HardCopy Stratix implementation. At this stage, the design is copied from the HARDCOPY_FPGA_PROTOTYPE project directory to a new directory to perform the timing analysis. This two-project directory structure enables you to move back and forth between the HARDCOPY_FPGA_PROTOTYPE design database and the HardCopy Stratix design database. The Quartus II software creates the <*project name*>**_hardcopy_optimization** directory.

You do not have to select the HardCopy Stratix device while performing performance estimation. When you run the HardCopy Timing Optimization Wizard, the Quartus II software selects the HardCopy Stratix device corresponding to the specified HARDCOPY_FPGA_PROTOTYPE FPGA. Thus, the information necessary for the HardCopy Stratix device is available from the earlier HARDCOPY_FPGA_PROTOTYPE device selection.

All constraints related to the design are also transferred to the new project directory. You can modify these constraints, if necessary, in your optimized design environment to achieve the necessary timing closure. However, if the design is optimized at the HARDCOPY_FPGA_PROTOTYPE device level by modifying the RTL code or the device constraints, you must migrate the project with the HardCopy Timing Optimization Wizard.

If an existing project directory is selected when the HardCopy Timing Optimization Wizard is run, the existing information is overwritten with the new compile results.

The project directory is the directory that you chose for the migrated project. A snapshot of the files inside the <*project name*>**_hardcopy_optimization** directory is shown in Table 4–7.

Table 4–7. Directory Structure Generated by the HardCopy Timing Optimization Wizard

*<project name>***_hardcopy_optimization**
*<project name>***.qsf**
*<project name>***.qpf**
*<project name>***.sof**
*<project name>***.macr**
*<project name>***.gclk**
db
hardcopy_fpga_prototype
fpga_*<project name>***_violations.datasheet**
fpga_*<project name>***_target.datasheet**
fpga_*<project name>***_rba_pt_hcpy_v.tcl**
fpga_*<project name>***_pt_hcpy_v.tcl**
fpga_*<project name>***_hcpy_v.sdo**
fpga_*<project name>***_hcpy.vo**
fpga_*<project name>***_cpld.datasheet**
fpga_*<project name>***_cksum.datasheet**
fpga_*<project name>***.tan.rpt**
fpga_*<project name>***.map.rpt**
fpga_*<project name>***.map.atm**
fpga_*<project name>***.fit.rpt**
fpga_*<project name>***.db_info**
fpga_*<project name>***.cmp.xml**
fpga_*<project name>***.cmp.rcf**
fpga_*<project name>***.cmp.atm**
fpga_*<project name>***.asm.rpt**
fpga_*<project name>***.qarlog**
fpga_*<project name>***.qar**
fpga_*<project name>***.qsf**
fpga_*<project name>***.pin**
fpga_*<project name>***.qpf**
db_export
*<project name>***.map.atm**
*<project name>***.map.hdbx**
*<project name>***.db_info**

1. Open the migrated Quartus II project created in step 3, "Select a destination directory for the migrated project and complete the HardCopy Timing Optimization Wizard process."
2. Perform a full compilation.

After successful compilation, the Timing Analysis section of the Compilation Report shows the performance of the design.

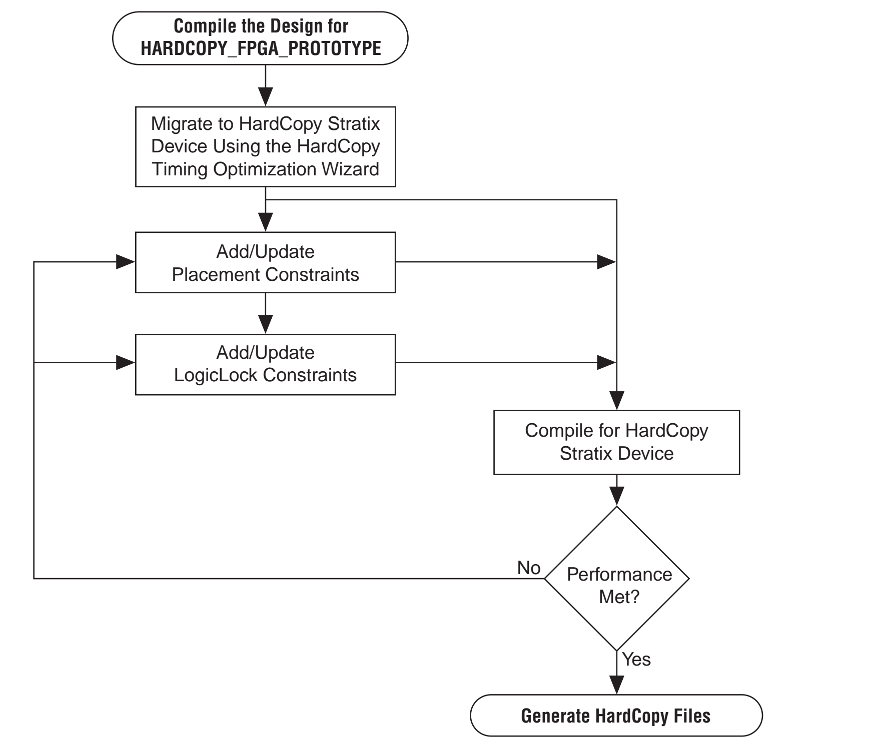
Buffer Insertion

Beginning with version 4.2, the Quartus II software provides improved HardCopy Stratix device timing closure and estimation, to more accurately reflect the results expected after back-end migration. The Quartus II software performs the necessary buffer insertion in your HardCopy Stratix device during the Fitter process, and stores the location of these buffers and necessary routing information in the **.qaf** file. This buffer insertion improves the estimation of the Quartus II Timing Analyzer for the HardCopy Stratix device.

Placement Constraints

Beginning with version 4.2, the Quartus II software supports placement constraints and LogicLock regions for HardCopy Stratix devices. Figure 4–26 shows an iterative process to modify the placement constraints until the best placement for the HardCopy Stratix device is achieved.

Figure 4–26. Placement Constraints Flow for HardCopy Stratix Devices



Location Constraints

This section provides information about HardCopy Stratix logic location constraints.

LAB Assignments

Logic placement in HardCopy Stratix is limited to LAB placement and optimization of the interconnecting signals between them. In a Stratix FPGA, individual logic elements (LEs) are placed by the Quartus II Fitter into LABs. The HardCopy Stratix migration process requires that LAB contents cannot change after the Timing Optimization Wizard task is done. Therefore, you can only make LAB-level placement optimization and location assignments after migrating the HARDCOPY_FPGA_PROTOTYPE project to the HardCopy Stratix device.

The Quartus II software supports these LAB location constraints for HardCopy Stratix devices. The entire contents of a LAB is moved to an empty LAB when using LAB location assignments. If you want to move the logic contents of LAB A to LAB B, the entire contents of LAB A are moved to an empty LAB B. For example, the logic contents of LAB_X33_Y65 can be moved to an empty LAB at LAB_X43_Y56 but individual logic cell LC_X33_Y65_N1 cannot be moved by itself in the HardCopy Stratix Timing Closure Floorplan.

LogicLock Assignments

The LogicLock feature of the Quartus II software provides a block-based design approach. Using this technique you can partition your design and create each block of logic independently, optimize placement and area, and integrate all blocks into the top level design.

To learn more about this methodology, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

LogicLock constraints are supported when you migrate the project from a HARDCOPY_FPGA_PROTOTYPE project to a HardCopy Stratix project. If the LogicLock region was specified as “Size=Fixed“ and “Location=Locked“ in the HARDCOPY_FPGA_PROTOTYPE project, it is converted to have “Size=Auto“ and “Location=Floating“ as shown in the following LogicLock examples. This modification is necessary because the floorplan of a HardCopy Stratix device is different from that of the Stratix device, and the assigned coordinates in the HARDCOPY_FPGA_PROTOTYPE do not match the HardCopy Stratix floorplan. If this modification did not occur, LogicLock assignments would lead to incorrect placement in the Quartus II Fitter. Making the regions auto-size and floating maintains your LogicLock assignments, allowing you to easily adjust the LogicLock regions as required and lock their locations again after HardCopy Stratix placement.

Example 4–2 and Example 4–3 show two examples of LogicLock assignments.

Example 4–2. LogicLock Region Definition in the HARDCOPY_FPGA_PROTOTYPE Quartus II Settings File

```
set_global_assignment -name LL_HEIGHT 15 -entity risc8 -section_id test
set_global_assignment -name LL_WIDTH 15 -entity risc8 -section_id test
set_global_assignment -name LL_STATE LOCKED -entity risc8 -section_id test
set_global_assignment -name LL_AUTO_SIZE OFF -entity risc8 -section_id test
```

Example 4–3. LogicLock Region Definition in the Migrated HardCopy Stratix Quartus II Settings File

```
set_global_assignment -name LL_HEIGHT 15 -entity risc8 -section_id test
set_global_assignment -name LL_WIDTH 15 -entity risc8 -section_id test
set_global_assignment -name LL_STATE FLOATING -entity risc8 -section_id test
set_global_assignment -name LL_AUTO_SIZE ON -entity risc8 -section_id test
```

Checking Designs for HardCopy Design Guidelines

When you develop a design with HardCopy migration in mind, you must follow Altera-recommended design practices to ensure a straightforward migration process or the design will not be able to be implemented in a HardCopy device. Prior to starting migration of the design to a HardCopy device, you must review the design and identify and address all the design issues. Any design issues that have not been addressed can jeopardize silicon success.

Altera-Recommended HDL Coding Guidelines

Designing for Altera PLD, FPGA, and HardCopy ASIC devices requires that certain specific design guidelines and HDL coding style recommendations be followed.

For more information about design recommendations and HDL coding styles, refer to the *Design Guidelines* section in volume 1 of the *Quartus II Handbook*.

Design Assistant

The Quartus II software includes the Design Assistant feature to check your design against the HardCopy design guidelines. Some of the design rule checks performed by the Design Assistant include the following rules:

- Design should not contain combinational loops
- Design should not contain delay chains
- Design should not contain latches

To use the Design Assistant, you must run Analysis and Synthesis on the design in the Quartus II software. Altera recommends that you run the Design Assistant to check for compliance with the HardCopy design guidelines early in the design process and after every compilation.

Design Assistant Settings

You must select the design rules on the **Design Assistant** page prior to running the design. On the Assignments menu, click **Settings**. In the **Settings** dialog box, in the **Category** list, select **Design Assistant** and turn on **Run Design Assistant during compilation**. Altera recommends enabling this feature to run the Design Assistant automatically during compilation of your design.

Running Design Assistant

To run Design Assistant independently of other Quartus II features, on the Processing menu, point to **Start** and click **Start Design Assistant**.

The Design Assistant automatically runs in the background of the Quartus II software when the HardCopy Timing Optimization Wizard is launched, and does not display the Design Assistant results immediately to the display. The design is checked before the Quartus II software migrates the design and creates a new project directory for performing timing analysis.

Also, the Design Assistant runs automatically whenever you generate the HardCopy design database with the HardCopy Files Wizard. The Design Assistant report generated is used by Altera's HardCopy Design Center to review your design.

Reports and Summary

The results of running the Design Assistant on your design are available in the Design Assistant Results section of the Compilation Report. The Design Assistant also generates the summary report in the <*project name*>**hardcopy** subdirectory of the project directory. This report file is titled <*project name*>**_violations.datasheet**. Reports include the settings, run summary, results summary, and details of the results and messages. The Design Assistant report indicates the rule name, severity of the violation, and the circuit path where any violation occurred.

To learn about the design rules and standard design practices to comply with HardCopy design rules, refer to the Quartus II Help and the *Design Guidelines for HardCopy Series Devices* chapter in the *HardCopy Stratix Device Handbook* on the Altera website.

Generating the HardCopy Design Database

You can use the HardCopy Files Wizard to generate the complete set of deliverables required for migrating the design to a HardCopy device in a single click. The HardCopy Files Wizard asks questions related to the design and archives your design, settings, results, and database files for delivery to Altera. Your responses to the design details are stored in <*project name*>**_hardcopy_optimization**<*project name*>**.hps.txt**.

You can generate the archive of the HardCopy design database only after compiling the design to a HardCopy Stratix device. The **.qaf** file is generated at the same directory level as the targeted project, either before or after optimization.

The Design Assistant automatically runs when the HardCopy Files Wizard is started.

Table 4–8 shows the archive directory structure and files collected by the HardCopy Files Wizard.

Table 4–8. HardCopy Stratix Design Files Collected by the HardCopy Files Wizard

- *<project name>***_hardcopy_optimization**
  - *<project name>***.flow.rpt**
  - *<project name>***.qpf**
  - *<project name>***.asm.rpt**
  - *<project name>***.blf**
  - *<project name>***.fit.rpt**
  - *<project name>***.gclk**
  - *<project name>***.hps.txt**
  - *<project name>***.macr**
  - *<project name>***.pin**
  - *<project name>***.qsf**
  - *<project name>***.sof**
  - *<project name>***.tan.rpt**
  - **hardcopy**
    - *<project name>***.apc**
    - *<project name>***_cksum.datasheet**
    - *<project name>***_cpld.datasheet**
    - *<project name>***_hcpy.vo**
    - *<project name>***_hcpy_v.sdo**
    - *<project name>***_pt_hcpy_v.tcl**
    - *<project name>***_rba_pt_hcpy_v.tcl**
    - *<project name>***_target.datasheet**
    - *<project name>***_violations.datasheet**
  - **hardcopy_fpga_prototype**
    - fpga_*<project name>***.asm.rpt**
    - fpga_*<project name>***.cmp.rcf**
    - fpga_*<project name>***.cmp.xml**
    - fpga_*<project name>***.db_info**
    - fpga_*<project name>***.fit.rpt**
    - fpga_*<project name>***.map.atm**
    - fpga_*<project name>***.map.rpt**
    - fpga_*<project name>***.pin**
    - fpga_*<project name>***.qsf**
    - fpga_*<project name>***.tan.rpt**
    - fpga_*<project name>***_cksum.datasheet**
    - fpga_*<project name>***_cpld.datasheet**
    - fpga_*<project name>***_hcpy.vo**
    - fpga_*<project name>***_hcpy_v.sdo**
    - fpga_*<project name>***_pt_hcpy_v.tcl**
    - fpga_*<project name>***_rba_pt_hcpy_v.tcl**
    - fpga_*<project name>***_target.datasheet**
    - fpga_*<project name>***_violations.datasheet**
  - **db_export**
    - *<project name>***.db_info**
    - *<project name>***.map.atm**
    - *<project name>***.map.hdbx**

After creating the migration database with the HardCopy Timing Optimization Wizard, you must compile the design before generating the project archive. You will receive an error if you create the archive before compiling the design.

Static Timing Analysis

In addition to performing timing analysis, the Quartus II software also provides all of the requisite netlists and Tcl scripts to perform static timing analysis (STA) using the Synopsys STA tool, PrimeTime. The following files, necessary for timing analysis with the PrimeTime tool, are generated by the HardCopy Files Wizard:

- <*project name*>**_hcpy.vo**—Verilog HDL output format
- <*project name*>**_hpcy_v.sdo**—Standard Delay Format Output File
- <*project name*>**_pt_hcpy_v.tcl**—Tcl script

These files are available in the <*project name*>**\hardcopy** directory. PrimeTime libraries for the HardCopy Stratix and Stratix devices are included with the Quartus II software.

Use the HardCopy Stratix libraries for PrimeTime to perform STA during timing analysis of designs targeted to HARDCOPY_FPGA_PROTOTYPE device.

For more information about static timing analysis, refer to the *Quartus II Classic Timing Analyzer* and the *Synopsys PrimeTime Support* chapters in volume 3 of the *Quartus II Handbook*.

Early Power Estimation

You can use PowerPlay Early Power Estimation to estimate the amount of power your HardCopy Stratix or HardCopy APEX device will consume. This tool is available on the Altera website. Using the Early Power Estimator requires some knowledge of your design resources and specifications, including:

- Target device and package
- Clock networks used in the design
- Resource usage for LEs, DSP blocks, PLL, and RAM blocks
- High speed differential interfaces (HSDI), general I/O power consumption requirements, and pin counts
- Environmental and thermal conditions

HardCopy Stratix Early Power Estimation

The PowerPlay Early Power Estimator provides an initial estimate of I_{CC} for any HardCopy Stratix device based on typical conditions. This calculation saves significant time and effort in gaining a quick understanding of the power requirements for the device. No stimulus vectors are necessary for power estimation, which is established by the clock frequency and toggle rate in each clock domain.

This calculation should only be used as an estimation of power, not as a specification. The actual I_{CC} should be verified during operation because this estimate is sensitive to the actual logic in the device and the environmental operating conditions.

For more information about simulation-based power estimation, refer to the *Power Estimation and Analysis* section in volume 3 of the *Quartus II Handbook*.

On average, HardCopy Stratix devices are expected to consume 40% less power than the equivalent FPGA.

Tcl Support for HardCopy Stratix

The Quartus II software also supports the HardCopy Stratix design flow at the command prompt using Tcl scripts.

For details about Quartus II support for Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Conclusion

The methodology for designing HardCopy devices using the Quartus II software is the same as that for designing the Stratix FPGA equivalent. You can use the familiar Quartus II software tools and design flow, target designs to HardCopy devices, optimize designs for higher performance and lower power consumption than the Stratix FPGAs, and deliver the design database for migration to a HardCopy device. Submit the files to Altera's HardCopy Design Center to complete the back-end migration.

Referenced Documents

This chapter references the following documents:

- *ALTMEM_INIT Megafunction User Guide*
- *AN 432: Using Different PLL Settings Between Stratix II and HardCopy II Devices*
- *Cadence Encounter Conformal Support* chapter in volume 3 of the *Quartus II Handbook*
- *Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Description, Architecture and Features* chapter in the *HardCopy II Device Family Data Sheet*
- *Design Guidelines for HardCopy Series Devices* chapter in the *HardCopy Stratix Device Handbook*
- *Design Guidelines* section in volume 1 of the *Quartus II Handbook*
- *HardCopy Stratix Device Family Data Sheet*
- *Introduction to the Quartus II Software* manual
- *Introduction to HardCopy II Devices* chapter in the *HardCopy II Device Family Data Sheet*

- *Power Estimation and Analysis* section in volume 3 of the *Quartus II Handbook*
- *Programming and Configuration* chapter of the *Introduction to the Quartus II Software* manual
- *Quartus II Analyzing and Optimizing Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *RAM Initializer (ALTMEM_INIT) Megafunction User Guide*
- *Synopsys PrimeTime Support* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 4–9 shows the revision history for this chapter.

Table 4–9. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Updated "RAM Usage Check" on page 4–28 ■ Updated "Referenced Documents"	Updated for Quartus II software version 9.0
November 2008 v8.1.0	■ Added HardCopy IV E support information ■ Added notes for Initialized Memory Dependency testing ■ Changed page size to 8.5" × 11"	Updated for Quartus II software version 8.1
May 2008 v8.0.0	■ Added new section "HardCopy Design Readiness Check". ■ Updated the tables and figures for HardCopy Series devices.	Updated for Quartus II software version 8.0

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

Section II. Design Guidelines

When designing large and complex FPGAs, your design and coding styles can impact your quality of results significantly. Designs following synchronous design practices behave in a predictable and reliable manner, even when re-targeted to different device families or speed grades. Using recommended HDL coding styles ensures that synthesis tools can infer the optimal device hardware to implement your design. Following best practices when creating your design hierarchy and logic provides the most flexibility when partitioning the design for incremental compilation, and leads to the best results. If you create floorplan location assignments to control the placement of different design blocks (useful in team-based designs so each designer can target a different area of the device floorplan), following best practices is important to maintaining good design performance.

This section presents design and coding style recommendations for your Altera® design, and includes the following chapters:

- Chapter 5, Design Recommendations for Altera Devices and the Quartus II Design Assistant

  This chapter describes synchronous design practices, and provides guidelines for combinational logic structures and clocking schemes. It also explains how to check design “rules” using the Quartus® II Design Assistant. Finally, it discusses targeting your design to use the clock and register-control features in the device architecture.

  Use this chapter at the start of your design process to guide the design.

- Chapter 6, Recommended HDL Coding Styles

  This chapter discusses Altera megafunctions and provides specific Verilog HDL and VHDL coding examples for inferring Altera dedicated logic such as memory and DSP blocks. It also provides device-specific coding recommendations for registers and certain logic functions such as tri-state signals, multiplexers, and cyclic redundancy check (CRC) functions, and includes references to other Altera documentation for low-level logic design.

  Use this chapter when you code specific design blocks to ensure you create HDL code that infers the optimal Altera device architecture.

- Chapter 8, Best Practices for Incremental Compilation Partitions and Floorplan Assignments

  This chapter provides a set of guidelines to help you set up and partition your design to take advantage of the compilation time savings, performance preservation, and hierarchical design features offered by Quartus II incremental compilation, and to help you create a design floorplan (using LogicLock™ regions) to support the flow when required.

Use this chapter when setting up your design hierarchy and determining the interfaces between logic blocks in your design, as well as if/when you create a design floorplan. You can also use this chapter to make changes to a design that was not originally set up to take advantage of incremental compilation, because it provides tips on changing a design to work better with an incremental design flow.

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

5. Design Recommendations for Altera Devices and the Quartus II Design Assistant

QII51006-9.0.0

Introduction

Current FPGA applications have reached the complexity and performance requirements of ASICs. In the development of such complex system designs, good design practices have an enormous impact on your device's timing performance, logic utilization, and system reliability. Well-coded designs behave in a predictable and reliable manner even when re-targeted to different families or speed grades. Good design practices also aid in successful design migration between FPGA and HardCopy® or ASIC implementations for prototyping and production.

For optimal performance, reliability, and faster time-to-market when designing with Altera® devices, adhere to the following guidelines:

- Understand the impact of synchronous design practices
- Follow recommended design techniques including hierarchical design partitioning
- Take advantage of the architectural features in the targeted device

This chapter presents design recommendations in these areas and describes the Quartus® II Design Assistant that can help you check your design for violations of design recommendations.

This chapter contains the following sections:

- "Synchronous FPGA Design Practices" on page 5–2
- "Design Guidelines" on page 5–4
- "Checking Design Violations Using the Design Assistant" on page 5–13
- "Targeting Clock and Register-Control Architectural Features" on page 5–41
- "Targeting Embedded RAM Architectural Features" on page 5–43

For specific HDL coding examples and recommendations, including coding guidelines for targeting dedicated device hardware, such as memory and digital signal processing (DSP) blocks, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

For information about migrating designs to HardCopy devices, refer to the *Design Guidelines for HardCopy Series Devices* chapter in volume 1 of the *HardCopy Series Handbook*.

For guidelines on partitioning a hierarchical design for incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Synchronous FPGA Design Practices

The first step in good design methodology is to understand the implications of your design practices and techniques. This section outlines some of the benefits of optimal synchronous design practices and the hazards involved in other techniques. Good synchronous design practices can help you meet your design goals consistently. Problems with other design techniques can include reliance on propagation delays in a device, incomplete timing analysis, and possible glitches.

In a synchronous design, a clock signal triggers all events. As long as all the registers' timing requirements are met, a synchronous design behaves in a predictable and reliable manner for all process, voltage, and temperature (PVT) conditions. You can easily target synchronous designs to different device families or speed grades. In addition, synchronous design practices help ensure successful migration if you plan to migrate your design to a high-volume solution such as an Altera HardCopy device or if you are prototyping an ASIC.

Fundamentals of Synchronous Design

In a synchronous design, everything is related to the clock signal. On every active edge of the clock (usually the rising edge), the data inputs of registers are sampled and transferred to outputs. Following an active clock edge, the outputs of combinational logic feeding the data inputs of registers change values. This change triggers a period of instability due to propagation delays through the logic as the signals go through a number of transitions and finally settle to new values. Changes happening on data inputs of registers do not affect the values of their outputs until the next active clock edge.

Because the internal circuitry of registers isolates data outputs from inputs, instability in the combinational logic does not affect the operation of the design as long as the following timing requirements are met:

- Before an active clock edge, the data input has been stable for at least the setup time of the register
- After an active clock edge, the data input remains stable for at least the hold time of the register

When you specify all of your clock frequencies and other timing requirements, the Quartus II Classic Timing Analyzer reports actual hardware requirements for the setup times (t_{SU}) and hold times (t_H) for every pin of your design. By meeting these external pin requirements and following synchronous design techniques, you ensure that you satisfy the setup and hold times for all registers within the Altera device.

To meet setup and hold time requirements on all input pins, any inputs to combinational logic that feeds a register should have a synchronous relationship with the clock of the register. If signals are asynchronous, you can register the signals at the input of the Altera device to help prevent a violation of the required setup and hold times.

When the setup or hold time of a register is violated, the output can be set to an intermediate voltage level between the high and low levels, called a metastable state. In this unstable state, small perturbations such as noise in power rails can cause the register to assume either the high or low voltage level, resulting in an unpredictable valid state. Various undesirable effects can occur, including increased propagation delays and incorrect output states. In some cases, the output can even oscillate between the two valid states for a relatively long period of time.

For details about timing requirements and analysis in the Quartus II software, refer to the *Quartus II Classic Timing Analyzer* or the *Quartus II TimeQuest Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*.

Hazards of Asynchronous Design

In the past, designers have often used asynchronous techniques such as ripple counters or pulse generators in programmable logic device (PLD) designs, enabling them to take “short cuts” to save device resources. Asynchronous design techniques have inherent problems such as relying on propagation delays in a device, which can result in incomplete timing constraints and possible glitches and spikes. Because current FPGAs provide many high-performance logic gates, registers, and memory, resource and performance trade-offs have changed. Now it is more important to focus on design practices that help you meet design goals consistently than to save device resources using problematic asynchronous techniques.

Some asynchronous design structures rely on the relative propagation delays of signals to function correctly. In these cases, race conditions can arise where the order of signal changes can affect the output of the logic. PLD designs can have varying timing delays, depending on how the design is placed and routed in the device with each compilation. Therefore, it is almost impossible to determine the timing delay associated with a particular block of logic ahead of time. As devices become faster because of device process improvements, the delays in an asynchronous design may decrease, resulting in a design that does not function as expected. Specific examples are provided in “Design Guidelines” on page 5–4. Relying on a particular delay also makes asynchronous designs very difficult to migrate to different architectures, devices, or speed grades.

The timing of asynchronous design structures is often difficult or impossible to model with timing assignments and constraints. If you do not have complete or accurate timing constraints, the timing-driven algorithms used by your synthesis and place-and-route tools may not be able to perform the best optimizations and the reported results may not be complete.

Some asynchronous design structures can generate harmful glitches, which are pulses that are very short compared with clock periods. Most glitches are generated by combinational logic. When the inputs of combinational logic change, the outputs exhibit a number of glitches before they settle to their new values. These glitches can propagate through the combinational logic, leading to incorrect values on the outputs in asynchronous designs. In a synchronous design, glitches on the data inputs of registers are normal events that have no negative consequences because the data is not processed until the clock edge.

Design Guidelines

When designing with HDL code, it is important to understand how a synthesis tool interprets different HDL design techniques and what results to expect. Your design techniques can affect logic utilization and timing performance, as well as the design's reliability. This section describes some basic design techniques that ensure optimal synthesis results for designs targeted to Altera devices while avoiding several common causes of unreliability and instability. Design your combinational logic carefully to avoid potential problems and pay attention to your clocking schemes so you can maintain synchronous functionality and avoid timing problems.

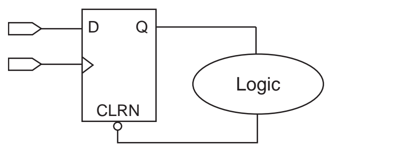
Combinational Logic Structures

Combinational logic structures consist of logic functions that depend only on the current state of the inputs. In Altera FPGAs, these functions are implemented in the look-up tables (LUTs) of the device's architecture, using either logic elements (LEs) or adaptive logic modules (ALMs). For some cases in which combinational logic feeds registers, the register control signals can also be used to implement part of the logic function to save LUT resources. By following the recommendations in this section, you can improve the reliability of your combinational design.

Combinational Loops

Combinational loops are among the most common causes of instability and unreliability in digital designs. They should be avoided whenever possible. In a synchronous design, feedback loops should include registers. Combinational loops generally violate synchronous design principles by establishing a direct feedback loop that contains no registers. For example, a combinational loop occurs when the left-hand side of an arithmetic expression also appears on the right-hand side in HDL code. A combinational loop also occurs when you feed back the output of a register to an asynchronous pin of the same register through combinational logic, as shown in Figure 5–1.

Figure 5–1. Combinational Loop through Asynchronous Control Pin



Use recovery and removal analysis to perform timing analysis on asynchronous ports such as `clear` or `reset` in the Quartus II software.

- If you are using the Classic Timing Analyzer, on the Assignments menu, click **Settings**. In the **Settings** dialog box, under **Timing Analysis Settings**, select **Classic Timing Analyzer Settings**. On the **Classic Timing Analyzer Settings** page, click **More Settings** and turn on the **Enable Recovery/Removal Analysis** option.
- If you are using the TimeQuest Timing Analyzer, refer to the "Recovery and Removal" section in the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* for details about how the TimeQuest Timing Analyzer performs recovery and removal analysis.

 © March 2009 Altera Corporation

Combinational loops are inherently high-risk design structures for the following reasons:

- Combinational loop behavior generally depends on relative propagation delays through the logic involved in the loop. As discussed, propagation delays can change, which means the behavior of the loop is unpredictable.
- Combinational loops can cause endless computation loops in many design tools. Most tools break open combinational loops to process the design. The various tools used in the design flow may open a given loop in a different manner, processing it in a way that is inconsistent with the original design intent.

Latches

A latch is a small circuit with combinational feedback that holds a value until a new value is assigned. Latches can be implemented directly with primitives, using `LPM_LATCH`, or inferred from HDL code. It is common for mistakes in HDL code to cause unintended latch inference. Quartus II Synthesis issues a warning message if this occurs.

Unlike other technologies, a latch in an FPGA architecture is not significantly smaller than a register. The architecture is not optimized for latch implementation and latches generally have slower timing performance compared to equivalent registered circuitry.

Latches have a transparent mode in which data flows continuously from input to output. A positive latch is in transparent mode when the enable signal is high (low for negative latch). In transparent mode, glitches on the input can pass through the output because of the direct path created. This presents significant complexity for timing analysis. Typical latch schemes use multiple enable phases to prevent long transparent paths from occurring. However, timing analysis is generally not able to identify these safe applications.

The Quartus II software setting **Analyze latches as Synchronous Elements** allows you to treat latches as having nontransparent start and end points. Bear in mind that even an instantaneous transition through transparent mode can lead to glitch propagation. The Quartus II software does not perform cycle-borrowing analysis, such as that performed by third-party timing analysis tools (such as the Synopsys PrimeTime software).

Due to various timing complexities, latches have limited support in formal verification tools. Therefore, it is very important that you do not use latches when using formal verification.

Altera recommends you avoid using latches to ensure that you can completely analyze the timing performance and reliability of your design.

Delay Chains

Delay chains occur when two or more consecutive nodes with a single fan-in and a single fan-out are used to cause delay. Inverters are often chained together to add delay. Delay chains are sometimes used to resolve race conditions created by other asynchronous design practices.

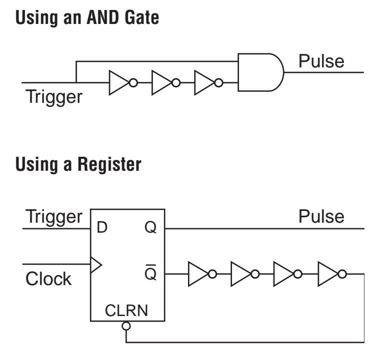
Delays in PLD designs can change with each place-and-route cycle. Effects such as rise and fall time differences and on-chip variation mean that delay chains, especially those placed on clock paths, can cause significant problems in your design. Refer to "Hazards of Asynchronous Design" on page 5–3 for examples of the kinds of problems that delay chains can cause. Avoid using delay chains to prevent these kind of problems.

In some ASIC designs, delays are used for buffering signals as they are routed around the device. This functionality is not required in FPGA devices because the routing structure provides buffers throughout the device.

Pulse Generators and Multivibrators

Delay chains are sometimes used to generate either one pulse (pulse generators) or a series of pulses (multivibrators). There are two common methods for pulse generation, as shown in Figure 5–2. These techniques are purely asynchronous and need to be avoided.

Figure 5–2. Asynchronous Pulse Generators



In "Using an AND Gate" (Figure 5–2), a trigger signal feeds both inputs of a 2-input AND gate, but the design inverts or adds a delay chain to one of the inputs. The width of the pulse depends on the relative delays of the path that feeds the gate directly and the one that goes through the delay. This is the same mechanism responsible for the generation of glitches in combinational logic following a change of input values. This technique artificially increases the width of the glitch by using a delay chain.

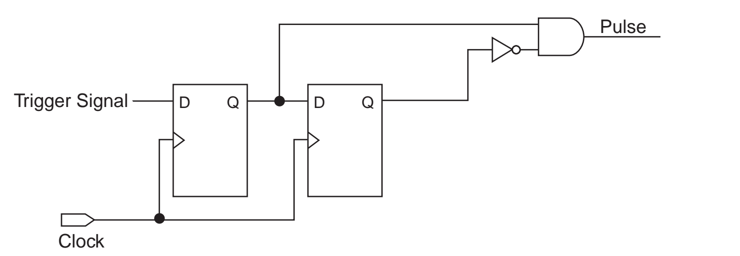
In "Using a Register" (Figure 5–2), a register's output drives the same register's asynchronous reset signal through a delay chain. The register resets itself asynchronously after a certain delay.

The width of pulses generated in this way are difficult for synthesis and place-and-route software to determine, set, or verify. The actual pulse width can only be determined after placement and routing, when routing and propagation delays are known. You cannot reliably determine the width of the pulse when creating HDL code and it cannot be set by EDA tools. The pulse may not be wide enough for the application under all PVT conditions. Also, the pulse width changes if you change to a different device. In addition, static timing analysis cannot be used to verify the pulse width, so verification is very difficult.

Multivibrators use a glitch generator to create pulses, together with a combinational loop that turns the circuit into an oscillator. This creates additional problems because of the number of pulses involved. In addition, when the structures generate multiple pulses, they also create a new artificial clock in the design that has to be analyzed by the design tools.

When you must use a pulse generator, use synchronous techniques, as shown in Figure 5–3.

Figure 5–3. Recommended Pulse-Generation Technique



In this design, the pulse width is always equal to the clock period. This pulse generator is predictable, can be verified with timing analysis, and is easily moved to other architectures, devices, or speed grades.

Clocking Schemes

Like combinational logic, clocking schemes have a large effect on your design's performance and reliability. Avoid using internally generated clocks wherever possible because they can cause functional and timing problems in the design. Clocks generated with combinational logic can introduce glitches that create functional problems and the delay inherent in combinational logic can lead to timing problems.

Specify all clock relationships in the Quartus II software to allow for the best timing-driven optimizations during fitting and to allow correct timing analysis. Use clock setting assignments on any derived or internal clocks to specify their relationship to the base clock.

Altera recommends using global device-wide, low-skew dedicated routing for all internally-generated clocks, instead of routing clocks on regular routing lines. For a detailed explanation, refer to "Clock Network Resources" on page 5–42.

Avoid data transfers between different clocks wherever possible. If you require a data transfer between different clocks, use FIFO circuitry. You can use the clock uncertainty features in the Quartus II software to compensate for the variable delays between clock domains. Consider setting a Clock Setup Uncertainty and Clock Hold Uncertainty value of 10% to 15% of the clock delay.

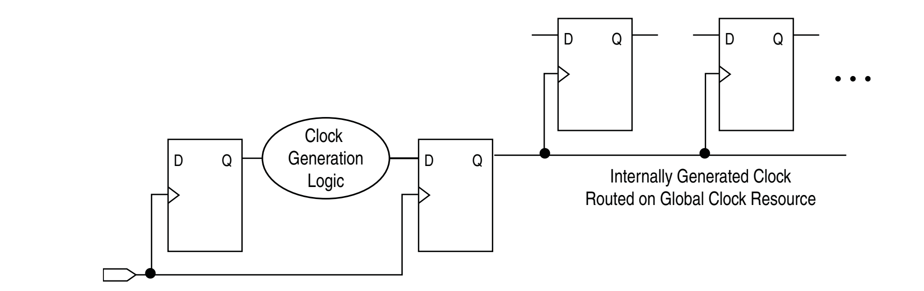
The following sections provide some specific examples and recommendations for avoiding these problems.

Internally Generated Clocks

If you use the output from combinational logic as a clock signal or as an asynchronous reset signal, expect to see glitches in your design. In a synchronous design, glitches on data inputs of registers are normal events that have no consequences. However, a glitch or a spike on the clock input (or an asynchronous input) to a register can have significant consequences. Narrow glitches can violate the register's minimum pulse width requirements. Setup and hold times may also be violated if the data input of the register is changing when a glitch reaches the clock input. Even if the design does not violate timing requirements, the register output can change value unexpectedly and cause functional hazards elsewhere in the design.

Because of these problems, Altera recommends that you always register the output of combinational logic before you use it as a clock signal (Figure 5–4).

Figure 5–4. Recommended Clock-Generation Technique



Registering the output of combinational logic ensures that the glitches generated by the combinational logic are blocked at the data input of the register.

Divided Clocks

Designs often require clocks created by dividing a master clock. Most Altera FPGAs provide dedicated phase-locked loop (PLL) circuitry for clock division. Using dedicated PLL circuitry can help you to avoid many of the problems that can be introduced by asynchronous clock division logic.

When you must use logic to divide a master clock, always use synchronous counters or state machines. In addition, create your design so that registers always directly generate divided clock signals, as described in "Internally Generated Clocks" on page 5–8, and route the clock on global clock resources. To avoid glitches, do not decode the outputs of a counter or a state machine to generate clock signals.

Ripple Counters

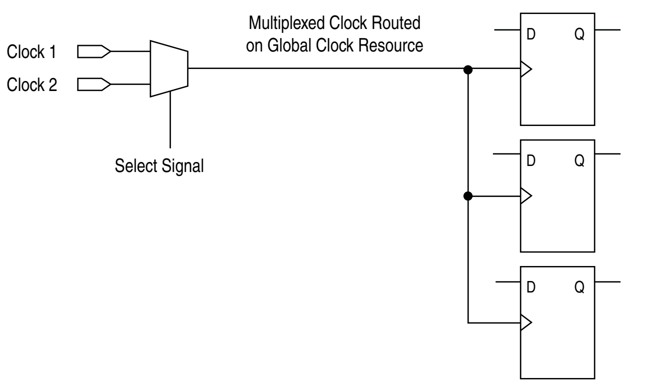
To simplify verification, Altera recommends avoiding ripple counters in your design. In the past, FPGA designers implemented ripple counters to divide clocks by a power of two because the counters are easy to design and may use fewer gates than their synchronous counterparts. Ripple counters use cascaded registers, in which the output pin of each register feeds the clock pin of the register in the next stage. This cascading can cause problems because the counter creates a ripple clock at each stage. These ripple clocks have to be handled properly during timing analysis, which can be difficult and may require you to make complicated timing assignments in your synthesis and place-and-route tools.

Ripple clock structures are often used to make ripple counters out of the smallest amount of logic possible. However, in all Altera devices supported by the Quartus II software, using a ripple clock structure to reduce the amount of logic used for a counter is unnecessary because the device allows you to construct a counter using one logic element per counter bit. Altera recommends that you avoid using ripple counters under any circumstances.

Multiplexed Clocks

Use clock multiplexing to operate the same logic function with different clock sources. In these designs, multiplexing selects a clock source, as in Figure 5–5. For example, telecommunications applications that deal with multiple frequency standards often use multiplexed clocks.

Figure 5–5. Multiplexing Logic and Clock Sources



Adding multiplexing logic to the clock signal can create the problems addressed in the previous sections, but requirements for multiplexed clocks vary widely, depending on the application. Clock multiplexing is acceptable when the clock signal uses global clock routing resources and if the following criteria are met:

- The clock multiplexing logic does not change after initial configuration
- The design uses multiplexing logic to select a clock for testing purposes
- Registers are always reset when the clock switches
- A temporarily incorrect response following clock switching has no negative consequences

If the design switches clocks in real time with no reset signal, and your design cannot tolerate a temporarily incorrect response, you must use a synchronous design so that there are no timing violations on the registers, no glitches on clock signals, and no race conditions or other logical problems. By default, the Quartus II software optimizes and analyzes all possible paths through the multiplexer and between both internal clocks that may come from the multiplexer. This may lead to more restrictive analysis than required if the multiplexer is always selecting one particular clock. If you do not require the more complete analysis, you can assign the output of the multiplexer as a base clock in the Quartus II software, so that all register-register paths are analyzed using that clock.

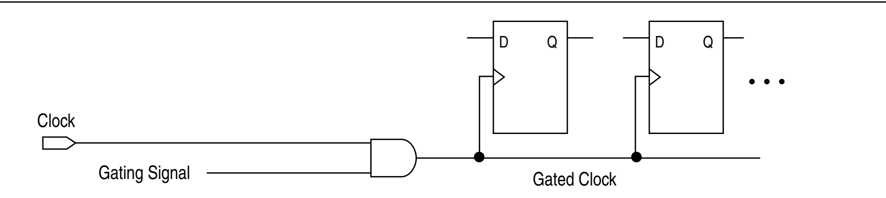
Altera recommends using dedicated hardware to perform clock multiplexing when it is available, instead of using multiplexing logic. For example, you can use the Clock Switchover feature or Clock Control Block available in certain Altera devices. These dedicated hardware blocks ensure that you use global low-skew routing lines and avoid any possible hold time problems on the device due to logic delay on the clock line.

Refer to the appropriate device data sheet or handbook for device-specific information about clocking structures.

Gated Clocks

Gated clocks turn a clock signal on and off using an enable signal that controls some sort of gating circuitry, as shown in Figure 5–6. When a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive.

Figure 5–6. Gated Clock



You can use gated clocks to reduce power consumption in some device architectures by effectively shutting down portions of a digital circuit when they are not in use. When a clock is gated, both the clock network and the registers driven by it stop toggling, thereby eliminating their contributions to power consumption. However, gated clocks are not part of a synchronous scheme and therefore can significantly increase the effort required for design implementation and verification. Gated clocks contribute to clock skew and make device migration difficult. These clocks are also sensitive to glitches, which can cause design failure.

Altera recommends that you use dedicated hardware to perform clock gating rather than using an AND or OR gate. For example, you can use the clock control block in newer Altera devices to shut down an entire clock network. Dedicated hardware blocks ensure that you use global routing with low skew and avoid any possible hold time problems on the device due to logic delay on the clock line.

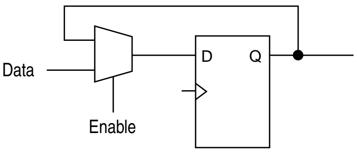
Refer to the appropriate device data sheet or handbook for device-specific information about clocking structures.

From a functional point of view, you can shut down a clock domain in a purely synchronous manner using a synchronous clock enable signal. However, when using a synchronous clock enable scheme, the clock network continues toggling. This practice does not reduce power consumption as much as gating the clock at the source does. In most cases, use a synchronous scheme such as those described in "Synchronous Clock Enables". For improved power reduction when gating clocks with logic, refer to "Recommended Clock-Gating Methods" on page 5–11.

Synchronous Clock Enables

To turn off a clock domain in a synchronous manner, use a synchronous clock enable signal. FPGAs efficiently support clock enable signals because there is a dedicated clock enable signal available on all device registers. This scheme does not reduce power consumption as much as gating the clock at the source because the clock network keeps toggling, but it performs the same function as a gated clock by disabling a set of registers. Insert a multiplexer in front of the data input of every register to either load new data or copy the output of the register (Figure 5–7).

Figure 5–7. Synchronous Clock Enable

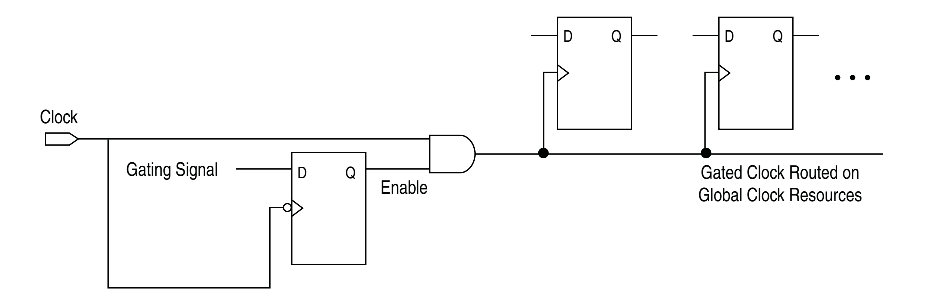


Recommended Clock-Gating Methods

Use gated clocks only when your target application requires power reduction and when gated clocks are able to provide the required reduction in your device architecture. If you must use clocks gated by logic, implement these clocks using the robust clock-gating technique shown in Figure 5–8 and ensure that the gated clock signal uses dedicated global clock routing.

You can gate a clock signal at the source of the clock network, at each register, or somewhere in between. Because the clock network contributes to switching power consumption, gate the clock at the source whenever possible, so you can shut down the entire clock network instead of gating it further along the clock network at the registers.

Figure 5–8. Recommended Clock-Gating Technique



In the technique shown in Figure 5–8, a register generates the enable signal to ensure that the signal is free of glitches and spikes. The register that generates the enable signal is triggered on the inactive edge of the clock to be gated (use the falling edge when gating a clock that is active on the rising edge, as shown in Figure 5–8). Using this technique, only one input of the gate that turns the clock on and off changes at a time. This prevents any glitches or spikes on the output. Use an AND gate to gate a clock that is active on the rising edge. For a clock that is active on the falling edge, use an OR gate to gate the clock and register the enable command with a positive edge-triggered register.

When using this technique, pay attention to the duty cycle of the clock and the delay through the logic that generates the enable signal because the enable command must be generated in one-half the clock cycle. This situation might cause problems if the logic that generates the enable command is particularly complex, or if the duty cycle of the clock is severely unbalanced. However, careful management of the duty cycle and logic delay may be an acceptable solution when compared with problems created by other methods of gating clocks.

Ensure that you apply a clock setting to the gated clock in the Quartus II software. As shown in Figure 5–8 on page 5–11, apply a clock setting to the output of the AND gate. Otherwise, the timing analyzer may analyze the circuit using the clock path through the register as the longest clock path and the path that skips the register as the shortest clock path, resulting in artificial clock skew.

In certain cases, converting the gated clocks to clock enables may help to reduce glitch and clock skew, and eventually produce a more accurate timing analysis. You can set the Quartus II software to automatically convert gated clocks to clock enables by turning on the **Auto Gated Clock Conversion** option. The conversion applies to two types of gated clocking schemes: single-gated clock and cascaded-gated clock. This option is only available for devices that are supported by the TimeQuest Timing Analyzer, except for Stratix® and Cyclone® devices.

For information about the settings and limitations of this option, refer to the "Auto Gated Clock Conversion" section of the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Design Techniques to Save Power

The total FPGA power consumption is comprised of I/O power, core static power, and core dynamic power. Knowledge of the relationship between these components is fundamental in calculating the overall total power consumption. You can use various optimization techniques and tools to minimize power consumption when applied during FPGA design implementation. The Quartus II software offers power-driven compilation features to fully optimize device power consumption. Power-driven compilation focuses on reducing your design's total power consumption using power-driven synthesis and power-driven place-and-route.

For information about power-driven compilation flow and low-power design guidelines, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*.

For information about power optimization techniques available for Stratix III devices, refer to *AN 437: Power Optimization in Stratix III FPGAs*. For information about power optimization techniques available for Stratix IV devices, refer to *AN 514: Power Optimization in Stratix IV FPGAs*.

In addition, you can use the Quartus II PowerPlay Power Analyzer to aid you during the design process by delivering fast and accurate estimations of power consumption. For information about the PowerPlay Power Analyzer, refer to the *PowerPlay Power Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

 © March 2009 Altera Corporation

Checking Design Violations Using the Design Assistant

To improve the reliability, timing performance, and logic utilization of your design, practicing good design methodology and understanding how to avoid design rule violations are important. The Quartus II software provides a tool that automatically checks for design rule violations and reports their location.

The Design Assistant is a design rule checking tool that allows you to check for design issues early in the design flow. The Design Assistant checks your design for adherence to Altera-recommended design guidelines. You can specify which rules you want the Design Assistant to apply to your design. This is useful if you know that your design violates particular rules that are not critical, so you can allow these rule violations. The Design Assistant generates design violation reports with clear details about each violation, based on the settings you specified.

The first parts in this section provide an introduction to the Quartus II design flow with Design Assistant, message severity levels, and an explanation about how to set up the Design Assistant. The last parts of the section describe the design rules and the reports generated by the Design Assistant.

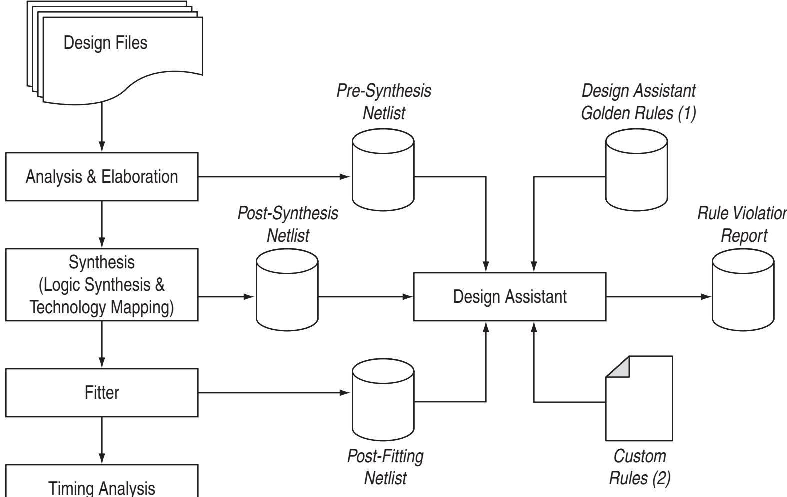
Quartus II Design Flow with the Design Assistant

You can run the Design Assistant after Analysis and Elaboration, Analysis and Synthesis, fitting, or a full compilation. To run the Design Assistant, on the Processing menu, point to **Start**, and click **Start Design Assistant**.

To set the Design Assistant to run automatically during compilation, on the Assignments menu, click **Settings**. In the **Category** list, select **Design Assistant**. Turn on **Run Design Assistant during compilation**. This enables the Design Assistant to perform a post-fitting netlist analysis of your design. The default is to apply all of the rules to your project. But if there are some rules that are unimportant to your design, you can turn off the rules that you do not want the Design Assistant to use. Refer to “The Design Assistant Settings Page” on page 5–15.

Figure 5–9 shows the Quartus II software design flow with the Design Assistant.

Figure 5–9. Quartus II Design Flow with the Design Assistant



Notes to Figure 5–9:

(1) Database of the default rules for the Design Assistant.

(2) A file that contains the **.xml** codes of the custom rules for the Design Assistant. For more details about how to create this file, refer to "Custom Rules" on page 5–35.

The Design Assistant analyzes your design netlist at different stages of the compilation flow and may yield different warnings or errors, even though the netlists are functionally the same. Your pre-synthesis, post-synthesis, and post-fitting netlists may be different due to optimizations performed by the Quartus II software. For example, a warning message in a pre-synthesis netlist may be removed after the netlist has been synthesized into a post-synthesis or post-fitting netlist.

- When you run the Design Assistant after running a full compilation or fitting, the Design Assistant performs a post-fitting analysis on the design.
- When you start the Design Assistant after performing Analysis and Synthesis, the Design Assistant performs post-synthesis analysis on the design.
- When you start the Design Assistant after performing Analysis and Elaboration, the Design Assistant performs a pre-synthesis analysis on the design. You can also perform pre-synthesis analysis with the Design Assistant using the command-line. You can use the `-rtl` option with the `quartus_drc` executable, as shown in the following example:

  quartus_drc <*project_name*> --rtl=on ↵

The Design Assistant generates warning messages when your design violates design rules and generates information messages to provide information regarding the rules. The Design Assistant supports all Altera devices supported by the Quartus II software.

The Design Assistant Settings Page

To apply design rules in the Design Assistant, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Settings** dialog box, in the **Category** list, select **Design Assistant**.
3. In the **Design Assistant** page, turn on the rules that you want the Design Assistant to apply during analysis. By default, all of the rules except the finite state machine rules are turned on.

To specify the file path to the custom rule file of the user-defined rules, refer to "Specifying the Path to the Custom Rules File" on page 5–37.

In the **Timing Closure** category, if **Nodes with more than specified number of fan-outs** or **Top nodes with highest fan-out** are turned on, you can use the **High Fan-Out Net Settings** dialog box to specify the number of fan-out a node must have to be reported by the Design Assistant. To open the **High Fan-Out Net Settings** dialog box, in the **Design Assistant** page, in the **Timing Closure** category, select **Nodes with more than specified number of fan-outs** or **Top nodes with highest fan-out**. Click **High Fan-Out Net Settings**.

In the **Clock** category, if you turn on **Clock signal should be a global signal**, you can use the **Global Clock Threshold Settings** dialog box to specify the number of nodes with the highest fan-out that you want the Design Assistant to report. To open the **Global Clock Threshold Settings** dialog box, on the **Design Assistant** page, in the **Clock** category, select **Clock signal should be a global signal**. Click **Global Clock Threshold Settings**.

To specify the maximum number of messages reported by the Design Assistant, on the **Design Assistant** page, click **Report Settings** and enter the maximum number of violation messages and detail messages to be reported.

Message Severity Levels

The Design Assistant classifies messages and rules using the four severity levels described in Table 5–1. Following Altera guidelines is very important for designs that are migrated to the HardCopy series of devices; therefore, the table highlights the impact of a rule violation on a HardCopy migration. Designs that adhere to Altera-recommended design guidelines do not produce any messages with critical-, high-, or medium-level severity.

Table 5–1. Design Assistant Message Severity Levels

Severity Level	Explanation
Critical	A violation of the rule critically affects the reliability of the design. Altera may not be able to implement the design successfully without closely reviewing the violations with the designer for HardCopy device conversions.
High	A violation of the rule affects the reliability of the design. Altera must review the violation before implementing the design for HardCopy device conversions.
Medium	The rule violation may result in implementation complexity that may have an impact for HardCopy device conversions.
Information Only	The rule provides information regarding the design.

Design Assistant Rules

This section describes the Design Assistant rules and details some of the reasons that Altera recommends following certain guidelines. Many of the Design Assistant rules enforce the design guidelines described in previous sections of this chapter.

Every rule is represented by a rule ID and has its own severity level. The rule ID is normally used in Tcl commands for rule suppression. The letter in each rule ID corresponds to the group of rules based on the following scheme:

- **A**—Asynchronous design structure rules
- **C**—Clock rules
- **R**—Reset rules
- **S**—Signal race rules
- **T**—Timing closure rules
- **D**—Asynchronous clock domain rules
- **H**—HardCopy rules
- **M**—Finite state machine rules

For example, the rule "Design Should Not Contain Combinational Loops" is the first rule in the asynchronous design structure rules; therefore, it is represented by rule ID A101.

Finite state machine rules are applicable only to register transfer level (RTL) check.

Summary of Rules and IDs

Table 5–2 lists the rules, their rule IDs, and their severity level.

Table 5–2. Summary of Rules and IDs (Part 1 of 2)

Rule ID	Rule Name	Severity Level
A101	Design Should Not Contain Combinational Loops	Critical
A102	Register Output Should Not Drive Its Own Control Signal Directly or through Combinational Logic	Critical
A103	Design Should Not Contain Delay Chains	High
A104	Design Should Not Contain Ripple Clock Structures	Medium
A105	Pulses Should Not Be Implemented Asynchronously	Critical
A106	Multiple Pulses Should Not Be Generated in the Design	Critical
A107	Design Should Not Contain SR Latches	High
A108	Design Should Not Contain Latches	High
A109	Combinational Logic Should Not Directly Drive Write Enable Signal of Asynchronous RAM	Medium
A110	Design Should Not Contain Asynchronous Memory	Medium
C101	Gated Clocks Should Be Implemented According to Altera Standard Scheme	Critical
C102	Logic Cell Should Not Be Used to Generate Inverted Clock	High
C103	Gated Clock Is Not Feeding At Least A Pre-Defined Number Of Clock Ports to Effectively Save Power: <n>	Medium
C104	Clock Signal Source Should Drive Only Input Clock Ports	Medium

Table 5–2. Summary of Rules and IDs (Part 2 of 2)

Rule ID	Rule Name	Severity Level
C105	Clock Signal Should Be a Global Signal	High
C106	Clock Signal Source Should Not Drive Registers that Are Triggered by Different Clock Edges	Medium
R101	Combinational Logic Used as a Reset Signal Should Be Synchronized	High
R102	External Reset Should Be Synchronized Using Two Cascaded Registers	Medium
R103	External Reset Should Be Synchronized Correctly	High
R104	Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized Correctly	High
R105	Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized	Medium
S101	Output Enable and Input of the Same Tri-State Nodes Should Not Be Driven by the Same Signal Source	High
S102	Synchronous Port and Asynchronous Port of the Same Register Should Not Be Driven by the Same Signal Source	High
S103	More Than One Asynchronous Signal Source of the Same Register Should Not Be Driven by the Same Source	High
S104	Clock Port and Any Other Signal Port of the Same Register Should Not Be Driven by the Same Signal Source	High
T101	Nodes with More Than Specified Number of Fan-outs: <n>	Information Only
T102	Top Nodes with Highest Fan-out: <n>	Information Only
D101	Data Bits Are Not Synchronized When Transferred between Asynchronous Clock Domains	High
D102	Multiple Data Bits Transferred Across Asynchronous Clock Domains Are Synchronized, But Not All Bits May Be Aligned in the Receiving Clock Domain	Medium
D103	Data Bits Are Not Correctly Synchronized When Transferred Between Asynchronous Clock Domains	High
H101	Only One VREF Pin Should Be Assigned to HardCopy Test Pin in an I/O Bank	Medium
H102	A PLL Drives Multiple Clock Network Types	Medium
M101	Data Bits Are Not Synchronized When Transferred to the State Machine of Asynchronous Clock Domains	High
M102	No Reset Signal Defined to Initialize the State Machine	Medium
M103	State Machine Should Not Contain Unreachable State	Medium
M104	State Machine Should Not Contain a Deadlock State	Medium
M105	State Machine Should Not Contain a Dead Transition	Medium

Design Should Not Contain Combinational Loops

Severity Level: Critical
Rule ID: A101

A combinational loop is created by establishing a direct feedback loop on combinational logic that is not synchronized by a register. A combinational loop also occurs when the output of a register is fed back to an asynchronous pin of the same register through combinational logic. Combinational loops are among the most common causes of instability and reliability in your designs and should be avoided whenever possible. Refer to "Combinational Loops" on page 5–4 for examples of the kinds of problems that combinational loops can cause.

Register Output Should Not Drive Its Own Control Signal Directly or through Combinational Logic

Severity Level: Critical
Rule ID: A102

A combinational loop occurs when you feed back the output of a register to an asynchronous pin of the same register (for example, the register's preset or asynchronous load signal), or the register drives combinational logic that drives one of the control signals on the same register. Combinational loops are among the most common causes of instability and reliability in your designs and should be avoided whenever possible. Refer to "Combinational Loops" on page 5–4 for examples of the kinds of problems that combinational loops can cause.

Design Should Not Contain Delay Chains

Severity Level: High
Rule ID: A103

Delay chains are created when one or more consecutive nodes with a single fan-in and a single fan-out are used to cause delay. Delay chains are sometimes used to create intentional delay to resolve race conditions. Delay chains may cause significant problems because they affect the rise and fall time differences in your design.

This rule applies only for delay chains implemented in logic cells and is limited to the clock and reset path of your design. This rule does not apply to delay chains in the data path. Altera recommends that you do not instantiate a cell that does not benefit the design and is used only to delay the signal. Refer to "Delay Chains" on page 5–5 for examples of the kinds of problems that delay chains can cause.

Design Should Not Contain Ripple Clock Structures

Severity Level: Medium
Rule ID: A104

Designs should not contain ripple clock structures. These structures use two or more cascaded registers in which the output of each register feeds the clock pin of the register in the next stage. Cascading structures cause large skew in the output signal because each stage of the structure causes a new clock domain to be defined. The additional clock domains from each stage of the ripple clock are difficult for static timing analysis tools to analyze. Refer to "Ripple Counters" on page 5–8 for examples of the kinds of problems that ripple clock structures can cause.

Pulses Should Not Be Implemented Asynchronously

Severity Level: Critical
Rule ID: A105

There are two common methods for pulse generation:

- Increasing the width of a glitch using a 2-input AND, NAND, OR, or NOR gate, where the source for the two gate inputs are the same, but one of the gate inputs is inverted
- Using a register where the register output drives the register's own asynchronous reset signal through a delay chain (refer to "Delay Chains" on page 5–5 for more details).

These techniques are purely asynchronous and therefore need to be avoided. Refer to "Pulse Generators and Multivibrators" on page 5–6 for recommended pulse generation guidelines.

Multiple Pulses Should Not Be Generated in the Design

Severity Level: Critical
Rule ID: A106

A common asynchronous multiple-pulse-generation technique consists of a combinational logic gate in which the inverted output feeds back to one of the inputs of the same gate. This feedback path causes a combinational loop that forces the output to change state and therefore, oscillate. Sometimes multiple pulse generators or multivibrator circuits are built out of a series of cascaded inverters in a structure called a "ring oscillator." Oscillation creates a new artificial clock in your design that is difficult for the Quartus II software to determine, set, or verify.

Structures that generate multiple pulses cause more problems than pulse generators because of the number of pulses involved. In addition, multiple pulse generators increase the frequency of the design. Refer to "Pulse Generators and Multivibrators" on page 5–6 for recommended pulse generation guidelines.

Design Should Not Contain SR Latches

Severity Level: High
Rule ID: A107

A latch is a combinational loop that holds the value of a signal until a new value is assigned. Combinational loops are hazardous to your design and are the most common causes of instability and unreliability. Refer to "Combinational Loops" on page 5–4 for examples of the kinds of problems that combinational loops can cause.

Rule A107 triggers only when your design contains SR latches. An SR latch can cause glitches and ambiguous timing, which complicates the timing analysis of your design. Refer to "Latches" on page 5–5 for details about latches and for more examples of the kinds of problems that latches can cause.

Design Should Not Contain Latches

Severity Level: High
Rule ID: A108

The Design Assistant generates warnings when it identifies one or more structures as latches.

Refer to "Latches" on page 5–5 for details about latches and for examples of the kinds of problems that latches can cause.

The difference between A107 ("Design Should Not Contain SR Latches") and A108 is that A107 triggers only when an SR latch is detected. A108 triggers when an unidentified latch exists in your design.

Combinational Logic Should Not Directly Drive Write Enable Signal of Asynchronous RAM

Severity Level: Medium
Rule ID: A109

Altera FPGA devices contain flexible embedded memory structures that can be configured into many different modes. One possible mode is asynchronous RAM. The definition of an asynchronous RAM circuit is one in which the write-enable signal driving into the RAM causes data to be written into it without a clock being required.

Do not use combinational logic to directly drive the write-enable signal of an asynchronous RAM. Any glitches that exist on the write-enable signal can cause the asynchronous RAM to be corrupted. Also, the data and write address ports of the RAM needs to be stable before the write pulse is asserted and must remain stable until the write pulse is de-asserted. Because of the limitations to using memory structures in this asynchronous mode, synchronous memories are always preferred. In addition, synchronous memories provide higher design performance.

As a guideline, a register should be used between combinational logic and asynchronous RAM or the asynchronous RAM should be replaced with synchronous memory. Refer to “Hazards of Asynchronous Design” on page 5–3 for examples of the kinds of problems asynchronous techniques can cause.

This rule applies only to device families that support asynchronous RAM.

Design Should Not Contain Asynchronous Memory

Severity Level: Medium
Rule ID: A110

Avoid using asynchronous memory (for example, asynchronous RAM) in your design because asynchronous memory can become corrupted by glitches created in the combinational logic that drives the write-enable signal of the memory. Asynchronous memory requires that the data and write address ports of the memory be stable before the write pulse is asserted and must remain stable until the write pulse is de-asserted. In addition, asynchronous memory has lower performance than synchronous memory.

As a guideline, a register should be used between combinational logic and asynchronous RAM or the asynchronous RAM should be replaced with synchronous memory. Immediately registering both input and output of the RAM improves performance and timing closure. Refer to “Hazards of Asynchronous Design” on page 5–3 for examples of the kinds of problems asynchronous techniques can cause.

This rule applies only to device families that support asynchronous RAM.

 © March 2009 Altera Corporation

Gated Clocks Should Be Implemented According to Altera Standard Scheme

Severity Level: Critical
Rule ID: C101

Clock gating is sometimes used to turn parts of a circuit on and off to reduce the total power consumption of a device. Clock gating is implemented using an enable signal that controls some sort of gating circuitry. The gated clock signal prevents any of the logic driven by it from switching so the logic does not consume any power. For example, when a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive. However, the disadvantage of using this type of circuit is that it can lead to unexpected glitches on the resultant gated clock signal if certain rules are not followed.

Refer to "Gated Clocks" on page 5–10 for examples of the kinds of problems gated clocks can cause. Refer to "Recommended Clock-Gating Methods" on page 5–11 for a recommended clock gating technique. However, when following the recommended clock gating techniques, your design must have a minimum number of fan-outs to meet rule C103, "Gated Clock Is Not Feeding At Least A Pre-Defined Number Of Clock Ports to Effectively Save Power: <n>."

Logic Cell Should Not Be Used to Generate Inverted Clock

Severity Level: High
Rule ID: C102

Your design may require both positive and negative edges of a clock to operate. However, do not implement an inverter to drive the clock input of a register in your design with a logic cell. Implementing the inverter with a logic cell can lead to clock insertion delay and skew, which is hazardous to your design and can cause problems with the timing closure of the design.

In addition, using a logic cell to implement an inverter is unnecessary. Use the programmable clock inversion featured in the register to generate the inverted clock signal. Refer to "Clocking Schemes" on page 5–7 for details about different types of clocking methods.

Gated Clock Is Not Feeding At Least A Pre-Defined Number Of Clock Ports to Effectively Save Power: <n>

Severity Level: Medium
Rule ID: C103

Your design can contain an input clock pin that fans out to more than one gated clock. However, to effectively reduce power consumption, Altera recommends that the gated clock feed at least a pre-defined number of clock ports (*n* ports). The default value for *n* is 30. You can set the number of clock ports (*n*) by performing the following steps:

1. Click **Settings** on the Assignments menu.
2. In the **Category** list, select **Design Assistant**.
3. On the **Design Assistant** page, expand the **Clock** category and turn on **Gated clock is not feeding at least a pre-defined number of clock port to effectively save power: <n>**.

4. Click on the **Gated Clock Settings** button, and in the **Gated Clock Settings** dialog box, set the number of clock ports a gated clock should feed.

Refer to "Clocking Schemes" on page 5–7, and "Recommended Clock-Gating Methods" on page 5–11 for proper clock-gating techniques.

Clock Signal Source Should Drive Only Input Clock Ports

Severity Level: Medium
Rule ID: C104

Clock signal sources in a design should drive only input clock ports of registers. Rule C104 triggers when a design contains a clock signal source of a register that connects to the port rather than the clock port of another register. Note that if the clock signal source and ports are of the same register, rule S104 "Clock Port and Any Other Signal Port of the Same Register Should Not Be Driven by the Same Signal Source" is triggered instead. Such a design is considered asynchronous and should be avoided. Asynchronous design structures can be hazardous to your design because some of them rely on the relative propagation delays of signals to function correctly, which can result in incomplete timing constraints and possible glitches and spikes.

Refer to "Hazards of Asynchronous Design" on page 5–3 for examples of the kinds of problems that asynchronous design structures can cause. Also refer to "Clocking Schemes" on page 5–7 for proper clocking techniques.

This rule does not apply in the following conditions:

- When the clock signal source drives combinational logic that is used as a clock signal and the combinational logic is implemented according to the Altera standard scheme
- When the clock signal source drives only a clock multiplexer that selects one clock source from a number of different clock sources

  This type of multiplexer adds complexity to the timing analysis of a design. Avoid using the multiplexer in the design.

- Using a clock multiplexer causes the "Gated Clocks Should Be Implemented According to Altera Standard Scheme" rule (C101) to be applied; refer to "Multiplexed Clocks" on page 5–9 for recommended clock multiplexing techniques

Clock Signal Should Be a Global Signal

Severity Level: High
Rule ID: C105

Ensure that all clock signals in your design use the global clock networks that exist in the target FPGA. Mapping clock signals to use non-dedicated clock networks can negatively affect the performance of your design. A non-global signal can be slower and have larger skew than a global signal because the clock must be distributed using regular FPGA routing resources.

To specify the number of minimum fan-outs that you want the Design Assistant to report, on the **Design Assistant** page, in the **Clock** category, select **Clock signal should be a global signal**. Click **Global Clock Threshold Settings** and enter the number in the dialog box.

If a design contains more clock signals than are available in the target device, consider reducing the number of clock signals in the design, such that only dedicated clock resources are used for clock distribution. However, if the design must use more clock signals than you can specify as global signals, implement the clock signals with the lowest fan-out using regular routing resources. Also, implement the fastest clock signals as global signals. Refer to “Clock Network Resources” on page 5–42 for detailed information about clock resources.

Clock Signal Source Should Not Drive Registers that Are Triggered by Different Clock Edges

Severity Level: Medium
Rule ID: C106

This rule triggers an error message if your design contains a clock signal source that drives the clock inputs of both positive and negative edge-sensitive registers. This error also triggers if your design contains an inverted clock signal that drives the clock inputs of either positive or negative edge-sensitive registers.

These two scenarios can cause an increase in timing requirement complexity and difficulties in design optimization. Also, synchronous resetting may not be possible because registers are not clocked on the same edge in the design. Refer to “Clocking Schemes” on page 5–7 for some specific examples and recommended clocking methods.

Combinational Logic Used as a Reset Signal Should Be Synchronized

Severity Level: High
Rule ID: R101

All combinational logic used to drive reset signals in your design needs to be synchronized. This means that a register is required between the combinational logic that drives the reset signal and input reset pin. Unsynchronized combinational logic can cause glitches and spikes that lead to unintentional reset signals. Synchronizing the combinational logic that drives the reset signal delays the resulting reset signal by an extra clock cycle and avoids unintentional reset. You must consider the extra clock cycle delay when using this method in your design.

Rule R101 does not trigger if the combinational logic used is either a 2-input AND or NOR that feeds active low reset port, or either a 2-input OR or NAND that feeds active high reset port.

External Reset Should Be Synchronized Using Two Cascaded Registers

Severity Level: Medium
Rule ID: R102

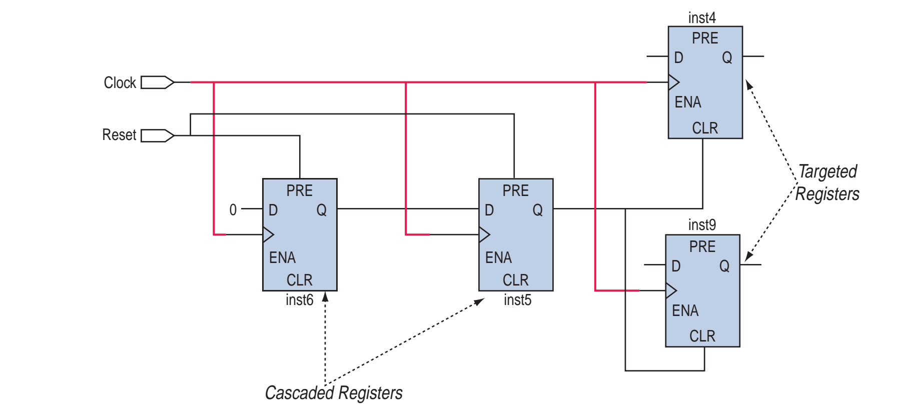
The only way to put your design into a reset state in the absence of a clock signal is to use an asynchronous reset or external reset. However, an asynchronous reset can affect the recovery time of a register, cause design stability problems, and unintentionally reset the state machines in your design to incorrect states.

As a guideline, you can synchronize an external reset signal by using a double-buffer circuit, which consists of two cascaded registers triggered on the same clock edge and on the same clock domain as the targeted registers.

This rule does not apply in the following two conditions:

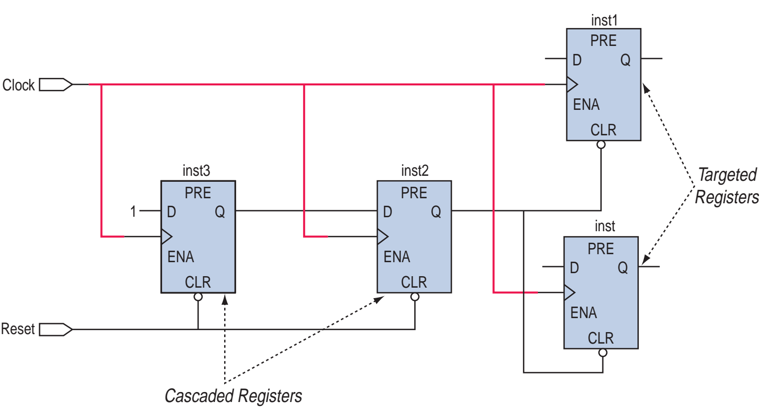
- When the targeted registers use active-high reset ports and the external reset signal drives the PRE ports on the cascaded registers with the input port of the first cascaded registers is fed to GND. Refer to Figure 5–10.

Figure 5–10. Active-High Reset Ports



- When the targeted registers use active-low reset ports and the external reset signal drives the CLR ports on the cascaded registers with the input port of the first cascaded registers is fed to V_{CC}. Refer to Figure 5–11.

Figure 5–11. Active-Low Reset Ports



External Reset Should Be Synchronized Correctly

Severity Level: High
Rule ID: R103

The only way to put your design into a reset state in the absence of a clock signal is to use an asynchronous reset or external reset. However, asynchronous reset can affect the recovery time of a register, cause design stability problems, and unintentionally reset the state machines in your design to incorrect states.

As a guideline, you can synchronize an external reset signal by using two cascaded registers. The registers need to be triggered on the same clock edge and should be in the same clock domain as the targeted registers.

This rule applies when an asynchronous reset or external reset signal is synchronized but fails to follow the recommended guidelines, as described in rule R102 ("External Reset Should Be Synchronized Using Two Cascaded Registers"). This violation happens when the external reset is synchronized with only one register or the cascaded synchronization registers are triggered on different clock edges.

R102 triggers when you don't use two cascaded registers to synchronize the external reset. R103 triggers when the external reset is synchronized but fails to follow the recommended guidelines.

Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized Correctly

Severity Level: High
Rule ID: R104

If your design uses an internally generated reset signal generated in one clock domain and used in one or more other asynchronous clock domains, the reset signal needs to be synchronized. An unsynchronized reset signal can cause metastability issues. To synchronize reset signals across clock domains, use the following guidelines:

- The reset signal needs to be synchronized with two or more cascading registers in the receiving asynchronous clock domain.
- The cascading registers needs to be triggered on the same clock edge.
- There must be no logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain. The synchronization registers may sample unintended data due to the glitches caused by the logic.

This rule applies when the internal reset signal is synchronized but fails to follow the recommended guidelines. This happens when the external reset is only synchronized with one register, when the cascaded synchronization registers are triggered on different clock edges, or when there is logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain. Synchronizing the reset signal delays the signal by an extra clock cycle. Consider this delay when using the reset signal in a design.

Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized

Severity Level: Medium
Rule ID: R105

If your design uses an internally generated reset signal that is generated in one clock domain and used in one or more other asynchronous clock domain, the reset signal needs to be synchronized. An unsynchronized reset signal can cause metastability issues. To synchronize reset signals across clock domains, follow the guidelines described in Rule R104 ("Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized Correctly").

This rule applies when the internally generated reset signal is not being synchronized.

Output Enable and Input of the Same Tri-State Nodes Should Not Be Driven by the Same Signal Source

Severity Level: High
Rule ID: S101

This rule applies when your design contains a tri-state node in which the input and output enable are driven by the same signal source. Signal race occurs between the input and output enable signals of the tri-state when they are propagated simultaneously. Race conditions lead to incorrect design function and unpredictable results. To avoid violation of this rule, the input and output enable of the tri-state should be driven by separate signal sources.

Synchronous Port and Asynchronous Port of the Same Register Should Not Be Driven by the Same Signal Source

Severity Level: High
Rule ID: S102

A purely synchronous design is free of signal race conditions as long as the clock signal is properly distributed and the timing requirements of the registers are met. However, race conditions can occur typically when the synchronous and asynchronous input pins of the register are driven by the same signal source. Race conditions can cause incorrect design function and unpredictable results. Rule S102 triggers when the synchronous and asynchronous pins of a register are driven by the same signal source. Rule S102 does not trigger if the signal source is from a negative-edge sensitive register of the same clock and if the source register is directly feeding the reset port, provided there is no combinational logic in-between the signal and register.

More Than One Asynchronous Signal Source of the Same Register Should Not Be Driven by the Same Source

Severity Level: High
Rule ID: S103

To avoid race conditions in your design, Altera recommends that you avoid using the same signal source to drive more than one port on a register. The following ports are affected: `ALOAD`, `ADATA`, `Preset`, and `Clear`.

Clock Port and Any Other Signal Port of the Same Register Should Not Be Driven by the Same Signal Source

Severity Level: High
Rule ID: S104

Any clock signal source in your design needs to drive only input clock ports of registers. Rule S104 triggers only when your design contains clock signal sources that connect to ports other than the clock ports of the same register. Rule S104 is a subset of C104, "Clock Signal Source Should Drive Only Input Clock Ports" on page 5–22. Such a design is considered asynchronous and should be avoided.

Refer to "Hazards of Asynchronous Design" for examples of the kinds of problems that asynchronous design structures can cause. Refer to "Clocking Schemes" for proper clocking techniques.

 © March 2009 Altera Corporation

Nodes with More Than Specified Number of Fan-outs: <n>

Severity Level: Information Only
Rule ID: T101

This rule reports nodes that have more than a specified number of fan-outs, which can create timing challenges for your design.

To specify the number of fan-outs, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Design Assistant**.
3. On the **Design Assistant** page, expand the **Timing closure** category by clicking the ⊞ icon next to **Timing closure**.
4. Turn on **Nodes with more than specified number of fan-outs**.
5. Click **High Fan-Out Net Settings**. In the **High Fan-Out Net Settings** dialog box, enter the number of fan-outs a node must have to be reported by the Design Assistant.

Top Nodes with Highest Fan-out: <n>

Severity Level: Information Only
Rule ID: T102

This rule reports the specified number of nodes with the highest fan-out, which can create timing challenges for your design.

To specify the number of fan-outs, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Design Assistant**.
3. On the **Design Assistant** page, click the ⊞ icon next to **Timing closure** to expand the folder.
4. Select **Nodes with more than specified number of fan-outs**.
5. Click **High Fan-Out Net Settings**.
6. In the **High Fan-Out Net Settings** dialog box, enter the number of nodes with the highest fan-out to be reported by the Design Assistant.

Data Bits Are Not Synchronized When Transferred between Asynchronous Clock Domains

Severity Level: High
Rule ID: D101

The data bits transferred between asynchronous clock domains in a design need to be synchronized to avoid metastability problems.

If the data bits belong to single-bit data, each data bit needs to be synchronized with two cascading registers in the receiving asynchronous clock domain, in which the cascaded registers are triggered on the same clock edge. Do not put any logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain.

If the data bits belong to multiple-bit data, use a handshake protocol to guarantee that all bits of the data bus are stable when the receiving clock domain samples the data. If you use a handshake protocol, only the data bits that act as `REQ` (request) and `ACK` (acknowledge) signals must be synchronized. The data bits that belong to multiple-bit data do not need to be synchronized. Ignore the violation on the data bits that use a handshake protocol.

Multiple Data Bits Transferred Across Asynchronous Clock Domains Are Synchronized, But Not All Bits May Be Aligned in the Receiving Clock Domain

Severity Level: Medium
Rule ID: D102

This rule applies when data bits from a multiple-bit data that are transferred between asynchronous clock domains are synchronized. However, not all data bits may be aligned in the receiving clock domain. Propagation delays may cause skew when the data reaches the receiving clock domain.

If the data bits belong to multiple-bit data and you use a handshake protocol, only the data bits that act as `REQ`, `ACK`, or both signals for the transfer need to be synchronized with two or more cascading registers in the receiving asynchronous clock domain.

If all of the data bits belong to single-bit data, the synchronization of the data bits does not cause problems in the design.

Data Bits Are Not Correctly Synchronized When Transferred Between Asynchronous Clock Domains

Severity Level: High
Rule ID: D103

The data bits that are transferred between asynchronous clock domains in a design need to be synchronized to avoid metastability problems.

If the data bits belong to single-bit data, each data bit needs to be synchronized with two cascading registers in the receiving asynchronous clock domain. In this case, the cascaded registers are triggered on the same clock edge. Do not put any logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain.

This rule only applies when the data bits across asynchronous clock domains are synchronized but fail to follow the guidelines.

Only One VREF Pin Should Be Assigned to HardCopy Test Pin in an I/O Bank

Severity Level: Medium
Rule ID: H101

If your design targets a HardCopy APEX™ 20K device, do not assign more than one VREF pin to a HardCopy test pin in an I/O bank in that targeted device. The assignment of more than one VREF pin to a HardCopy test pin can cause contention of the VREF bus.

You can find the list of HardCopy test pins that are in each of a HardCopy APEX 20K device’s I/O banks in the Messages window, the Design Assistant Messages report, and the Design Assistant HardCopy Test Pins report. Use this information to ensure that only one VREF pin is assigned to a HardCopy test pin.

However, the Fitter may have assigned the VREF pins to the HardCopy test pins during compilation. To prevent the Fitter from making these assignments during the next compilation, create and assign VREF pins manually instead of allowing the Fitter to do so automatically.

This rule applies only to designs that target HardCopy APEX 20K devices.

A PLL Drives Multiple Clock Network Types

Severity Level: Medium
Rule ID: H102

A PLL can compensate only one of the clock network types; therefore, the other non-compensated clock network types have a non-zero delay. However, the non-zero delay for the non-compensated clock network types can change between a Stratix device and its corresponding HardCopy Stratix device, or a Stratix II device and its corresponding HardCopy II device.

Therefore, if a Stratix FPGA design relies on the relative offset between the compensated clock network type and the non-compensated clock network types driven by a PLL, an error can occur in the corresponding HardCopy Stratix design because the relative offset in the HardCopy Stratix design may differ from the relative offset in the original Stratix FPGA design.

This rule reports only nodes in a design where a PLL drives multiple clock network types.

Data Bits Are Not Synchronized When Transferred to the State Machine of Asynchronous Clock Domains

Severity Level: High
Rule ID: M101

Data bits that are transferred between asynchronous clock domains in your design need to be synchronized to avoid metastability problems. Rule M101 is a state-machine-specific rule that triggers when input signals of a state machine across asynchronous clock domains are not synchronized or improperly synchronized. Rule M101 applies to state machines only, while the “Data Bits Are Not Synchronized When Transferred between Asynchronous Clock Domains” rule (D101) and the “Data Bits Are Not Correctly Synchronized When Transferred Between Asynchronous Clock Domains” rule (D103) apply only to data synchronization between registers.

No Reset Signal Defined to Initialize the State Machine

Severity Level: Medium
Rule ID: M102

A finite state machine (FSM) needs to have a reset signal that initializes the state machine to its initial state. A finite state machine without a proper initialization state is susceptible to functional problems and can introduce extra difficulty in analysis, verification, and maintenance.

State Machine Should Not Contain Unreachable State

Severity Level: Medium
Rule ID: M103

An unreachable state is a state that can never be reached from the initial state. Having an unreachable state in your design causes logic redundancy and affects your design functionality. Rule M103 triggers when the initial state cannot traverse to a state through any of the reachable states and transitions.

State Machine Should Not Contain a Deadlock State

Severity Level: Medium
Rule ID: M104

A deadlock state is a state that does not have any transitions to another state except to loop to itself. When the state machine enters a deadlock state, it stays in that state until the state machine is reset. Your design may have a single state, or a few states forming a deadlock structure. Having a deadlock state in your design leads to design functionality problems, and theoretically may consume more power.

You can change the maximum number of states to be detected as a deadlock structure by clicking **Settings** on the Assignments menu, and in the **Settings** dialog box, in the **Category** list, select **Design Assistant**. In the **Design Assistant** page, click **Finite State Machine Deadlock Settings**. In the **Finite State Machine Deadlock Settings** dialog box, specify the maximum number of states to be reported as a deadlock structure. The default setting is `2`.

State Machine Should Not Contain a Dead Transition

Severity Level: Medium
Rule ID: M105

A dead transition is a redundant transition that never occurs regardless of the event sequence input to the state machine. A dead transition indicates logic redundancy in your design, although it may not affect your design functionality. Rule M105 triggers when the condition required to trigger a transition is not possible.

Enabling and Disabling Design Assistant Rules

You can selectively enable or disable Design Assistant rules on individual nodes by making an assignment in the Assignment Editor or by using the `altera_attribute` synthesis attribute in Verilog HDL or VHDL, or using a Tcl command.

For a list of the types of nodes that allow Design Assistant rule suppression, refer to *Node Types Eligible for Rule Suppression* in the Quartus II Help.

Assignments made with Assignment Editor, the Quartus Settings File (**.qsf**), and Tcl scripts and commands, take precedence over assignments made with the `altera_attribute` synthesis attribute. Assignments made to nodes, entities, or instances, take precedence over global assignments.

Using the Assignment Editor

You can enable or disable a Design Assistant rule on selected nodes in your design by using the Assignments Editor. You must first compile your design if you have not already done so.

 © March 2009 Altera Corporation

To enable or disable a Design Assistant rule, follow these steps:

1. On the Assignments menu, click **Assignment Editor**.
2. In the spreadsheet, for the desired node, entity, or instance, double-click the cell in the **Assignment Name** column and select **Enable Design Assistant Rule** or **Disable Design Assistant Rule** in the pull-down menu.
3. Double-click the **Value** cell and type in the Rule ID.

   or

   Click **Browse** to open the **Design Assistant Rules** dialog box. In the **Design Assistant Rules** dialog box, select the rule you want to enable or disable for that particular node.

You can enable or disable multiple rules by typing more than one Rule ID in the cell and separating each Rule ID with a comma.

Using Verilog HDL

You can use the `altera_attributes` synthesis attribute in your Verilog HDL code to enable or disable a Design Assistant rule on the selected nodes in your design.

To enable the rule on the selected node, the syntax is shown in the following example:

```
<entity class> <object> /* synthesis altera_attribute="enable_da_rule=<ruleID>" */
```

You can enable more than one rule on a selected node as shown in the following example:

```
<entity class> <object> /* synthesis altera_attribute="enable_da_rule=\"<ruleID>,
<ruleID>, <ruleID>\""*/
```

To disable the rule on the selected node, the syntax is shown in the following example:

```
<entity class> <object> /* synthesis altera_attribute="disable_da_rule=<ruleID>" */
```

You can disable more than one rule on a selected node as shown in the following example:

```
<entity class> <object> /* synthesis altera_attribute="disable_da_rule=\"<ruleID>,
<ruleID>, <ruleID>\""*/
```

When enabling or disabling multiple rules in Verilog HDL, you must separate the Rule ID strings with commas and spaces only and they must be enclosed within the `\"` and `\"` characters.

Using VHDL

You can use the `altera_attributes` synthesis attribute in your VHDL code to enable or disable a Design Assistant rule on the selected nodes in your design.

To enable the rule on the selected node, use the following syntax:

```
attribute altera_attribute : string;attribute altera_attribute of <object>: <entity
class> is "enable_da_rule=<ruleID>"
```

You can enable more than one rule on a selected node as shown in the following example:

```
attribute altera_attribute : string;attribute altera_attribute of <object>: <entity
class> is "enable_da_rule=""<ruleID>, <ruleID>, <ruleID>"""
```

To disable the rule on the selected node, use the following syntax:

```
attribute altera_attribute : string;attribute altera_attribute of <object>: <entity
class> is "disable_da_rule=<ruleID>"
```

You can disable more than one rule on a selected node as shown in the following example:

```
attribute altera_attribute : string;attribute altera_attribute of <object>: <entity
class> is "disable_da_rule=""<ruleID>, <ruleID>, <ruleID>"""
```

When enabling or disabling multiple rules in VHDL, you must separate the Rule ID strings with commas and spaces only and they must be enclosed with double quotation mark (" ") characters.

Using TCL Commands

To enable a Design Assistant rule on the selected node in your design using Tcl in a script or at a Tcl prompt, use the following Tcl command:

```
set_instance_assignment -name enable_da_rule "<rule ID>" -to <design element> ↵
```

To enable more than one rule on a selected node, use the following Tcl command:

```
set_instance_assignment -name enable_da_rule "<rule ID>, <rule ID>"
-to <design element> ↵
```

To disable a Design Assistant rule on a selected node in your design using Tcl in a script, or at a command or Tcl prompt, use the following Tcl command:

```
set_instance_assignment -name disable_da_rule "<rule ID>" -to <design element> ↵
```

To disable more than one rule on a selected node, use the following Tcl command:

```
set_instance_assignment -name disable_da_rule "<rule ID>, <rule ID>"
-to <design element> ↵
```

Viewing Design Assistant Results

If your design violates a design rule, the Design Assistant generates warning messages and information messages about the violated design rule. The Design Assistant displays these messages in the Messages window, in the Design Assistant Messages report, and in the Design Assistant report files. You can find the Design Assistant report files called *<project_name>*.**drc.rpt** in the *<project_name>* subdirectory of the project directory.

The Design Assistant generates the following reports based on the settings specified in the Design Assistant page:

- Summary Report
- Settings Report
- Detailed Results Report
- Messages Report
- HardCopy Test Pins Report
- Rule Suppression Assignments Report

- Ignored Design Assistant Assignments Report
- Custom Rules Report

Summary Report

The Design Assistant Summary report contains a summary of the Design Assistant process on a particular project. The Design Assistant Summary report provides the following information:

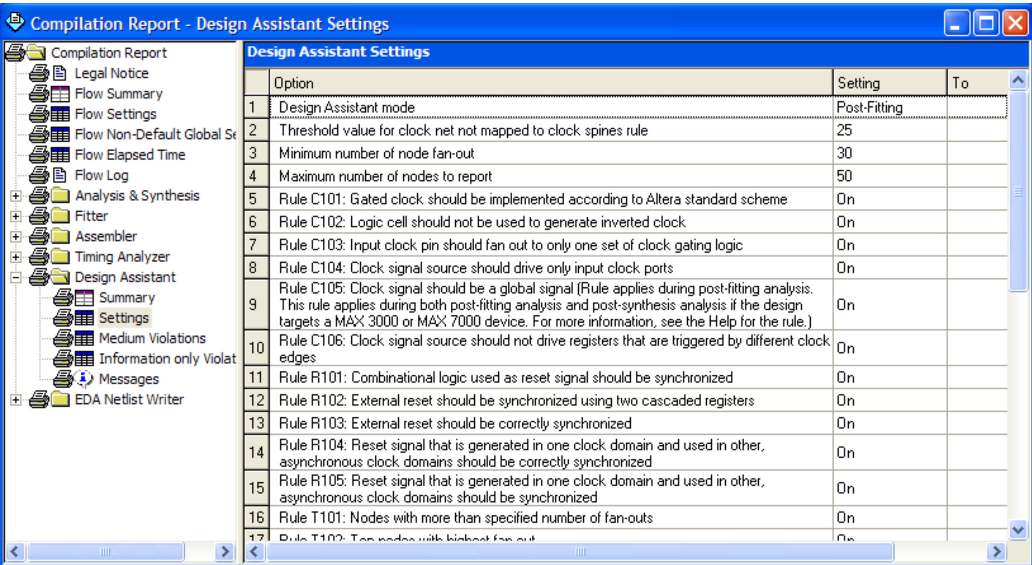
- **Design Assistant Status—**the status, end date, and end time of the Design Assistant operation
- **Revision Name—**the revision name specified in the **Revisions** dialog box
- **Top-level Entity Name—**the top-level entity of your design
- **Family—**the device family name specified in the **Device** page of the **Settings** dialog box
- **Total Critical Violations**, **Total High Violations**, **Total Medium Violations**, and **Total Information Only Violations—**the total violations of the rules organized by level, some of which might affect the reliability of the design

Review the violations closely before converting your design for HardCopy devices to achieve a successful conversion.

Settings Report

The Design Assistant Settings report contains a list of enabled Design Assistant rules and options that you specified in the **Design Assistant Settings** page, as shown in Figure 5–12.

Figure 5–12. The Design Assistant Settings Report

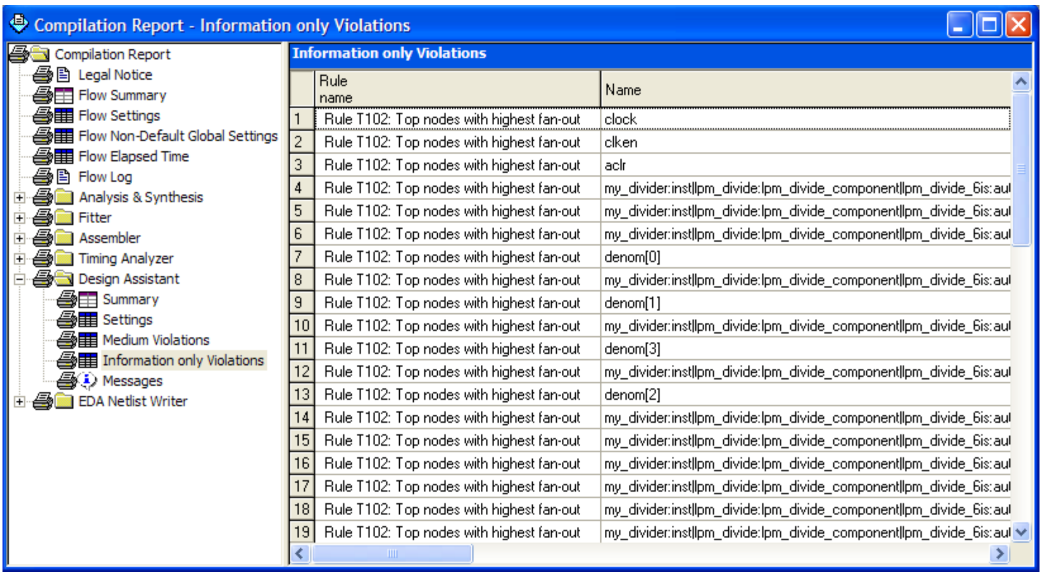


Detailed Results Report

The Detailed Results report contains detailed information of every rule violation including the rule name, node name, and fan-out. This report appears only if you specify settings in the **Design Assistant Settings** page. For more information about how to specify the settings, refer to “The Design Assistant Settings Page” on page 5–15.

Separate Detailed Results reports are generated for critical, high, medium, and information only results. Figure 5–13 shows the **Information Only Violations** report.

Figure 5–13. The Design Detailed Results Report, Information Only



Messages Report

The Messages report contains current information, warning, and error messages generated during the Design Assistant process. You can right-click a message in the Messages report and click **Help** to display the Quartus II software Help with details about the selected message, or click **Locate** to trace or cross-probe the selected node and locate the source of the violation.

HardCopy Test Pins Report

The HardCopy Test Pins report appears only if you turn on **Run Design Assistant during compilation** in the **Design Assistant** page and if your design violates the “Only One VREF Pin Should Be Assigned to HardCopy Test Pin in an I/O Bank” rule (H101). The report lists all the HardCopy design rule violations and all of the test pins in the HardCopy device.

Rule Suppression Assignments Report

The Rule Suppression Assignments report contains detailed information about which Design Assistant rules are enabled or disabled, as explained in the “Enabling and Disabling Design Assistant Rules” on page 5–30. The report shows the following information:

- **Assignment**—shows the name of the assignment

- **Value**—identifies the rule
- **To**—shows the name of the node where the rule is being applied

Ignored Design Assistant Assignments Report

The Ignored Design Assistant Assignments report lists detailed information about the invalid and conflicting rule assignments reported by the Design Assistant. This report is generated only if you specify an invalid rule ID in the rule suppression or a conflicting rule assignment. The following information appears in the report:

- **Assignment**—shows the name of the assignment
- **Value**—identifies the rule
- **To**—shows the name of the node where the rule is being applied
- **Comment**—shows why the assignment is being ignored

Custom Rules Report

The Design Assistant Custom Rules report contains the names of the custom rules used in the design checking, the path to the custom rules files from which the custom rules are read, and the list of ignored custom rules.

Custom Rules

In addition to the existing design rules that the Design Assistant offers, you can also create your own rules and specify your own reporting format in a text file (with any file extension) using the XML format. You then specify the path to that file in the Design Assistant settings page and run the Design Assistant for violations checking.

The file that contains the default rules for the Design Assistant is located at <*Quartus II install path*>**\quartus\libraries\design-assistant\da_golden_rule.xml**.

For details about how to set the file path to your custom rules, refer to "Specifying the Path to the Custom Rules File" on page 5–37.

This section explains the basics of writing a custom rule, the Design Assistant settings, and provides coding examples on how to check for clock relationship and node relationship in a design.

XML File Format for Custom Rules

All XML commands in custom rules file must be written within the `<ROOT>` and `</ROOT>` tags. Every user-defined rule consists of three main sections:

- Rule Attribute
- Rule Definition
- Reporting

The Rule Definition and Reporting sections must be defined inside the Rule Attribute section. Example 5–1 shows all three sections in a pre-defined custom rule file.

XML commands and attributes are case sensitive. However, attribute values are *not* case sensitive.

Example 5–1. Predefined XML File Format for a Custom Rule

```
<ROOT>
<!-Start creating a rule here -->

<!--Define rule attribute for a rule here -->
<DA_RULE ID=<rule id> NAME=<rule name> SEVERITY=<rule severity> DEFAULT_RUN=<default run> >

<RULE_DEFINITION>
      <!--Define rule definition here -->
   </RULE_DEFINITION>

   <REPORTING>
      <!--Define report settings here -->
   </REPORTING>

</DA_RULE>

</ROOT>
```

The Rule Attribute section contains the name, ID, severity level, and enable value of a rule. The order of these attributes is not important. This section is enclosed within `<DA_RULE>` and `</DA_RULE>` tags. Table 5–3 describes the attributes of the Rule Attribute section.

Table 5–3. Attributes for the Rule Attribute Section

Attribute	Description
ID	The value for this attribute is string type and must be unique. This attribute is required. For the list of IDs of the default rules, refer to Table 5–2 on page 5–16.
NAME	The value for this attribute is string type. This attribute is optional.
SEVERITY	This attribute presents the severity level of the rule. The value is string type and can be `CRITICAL`, `HIGH`, `MEDIUM`, or `INFORMATION`. This attribute is required. For details about rule severity level, refer to "Message Severity Levels" on page 5–15.
DEFAULT_RUN	The value is string type and can only be `YES`, or `NO`. If the value is `YES`, the rule is included in the design rule check, and vice versa. By default, the value is `YES`. This attribute is optional.

All string-type values must be enclosed within double quotes.

Command lines that begin with a single XML tag must end with the "`/>`" sign before another command begins.

The Rule Definition section is where you declare the node properties and rule triggering conditions, enclosed by `<RULE_DEFINITION>` and `</RULE_DEFINITION>` tags.

There are four subsections within the Rule Definition section that you can use to declare the properties and conditions:

- `<DECLARE>`—Global nodes that are used in the file are declared in this subsection. Every node name must be unique.

A node declared outside of the `<DECLARE>` subsection is considered a local node. You can perform local node declaration at any place in the `<BASIC>`, `<REQUIRED>`, and `<FORBID>` subsections, and can be performed using the node declaration command directly without being enclosed within the `<DECLARE>` tag.

- `<BASIC>`—This subsection contains the condition that acts like a trigger point which the Design Assistant continuously checks for a match. If the condition is fulfilled, the Design Assistant checks the remaining conditions in the `<REQUIRED>` and `<FORBID>` subsections.
- `<REQUIRED>`—This subsection contains the acceptable conditions that your design must meet. If the condition is not fulfilled, the Design Assistant reports a rule violation.
- `<FORBID>`—This subsection contains the undesirable condition for a design. If the condition is fulfilled, the Design Assistant highlights a rule violation. This subsection may be optional, depending on your rule situation.

The Reporting section is where you describe the settings for rule violation reporting, enclosed by `<REPORTING>` and `</REPORTING>` tags. This section is optional. If there is no Reporting section defined, the violated rule is not reported. If the Reporting section is defined, the Design Assistant reports the name of the violated rule and the nodes that violated the rule according to the reporting format that you defined.

Specifying the Path to the Custom Rules File

To specify the path to the custom rule file, follow these steps:

1. To specify the path, on the Assignments menu, click **Settings**.
2. In the Category list, click **Design Assistant** and select **Custom Rules Settings**.
3. In the **Custom Rules Settings** dialog box, in the **Project custom rules file name** field, specify the path to your custom rules file.
4. Click **OK**.

Your rules are now included in the list of default Design Assistant rules.

The default file extension for a Design Assistant custom rules file is **.dacr**, but the file can have any file name or extension.

To specify the rules that you want the Design Assistant to check for violations, refer to “The Design Assistant Settings Page” on page 5–15.

Custom Rules Coding Examples

The following examples of custom rules show how to check node relationships and clock relationships in a design.

Checking SR Latch Structures In a Design

Example 5–2 shows the XML codes for checking SR latch structures in a design.

Example 5–2. Detecting SR Latches in a Design

```
<DA_RULE ID="EX01" SEVERITY="CRITICAL" NAME="Checking Design for SR Latch"
DEFAULT_RUN="YES">
<RULE_DEFINITION>
   <FORBID>
   <OR>
      <NODE NAME="NODE_1" TYPE="SRLATCH" />
      <HAS_NODE NODE_LIST="NODE_1" />
      <NODE NAME="NODE_1" TOTAL_FANIN="EQ2" />
      <NODE NAME="NODE_2" TOTAL_FANIN="EQ2" />
      <AND>
         <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NAND" TO_NAME="NODE_2"
TO_TYPE="NAND" />
         <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NAND" TO_NAME="NODE_1"
TO_TYPE="NAND" />
      </AND>
      <AND>
         <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NOR" TO_NAME="NODE_2"
TO_TYPE="NOR" />
         <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NOR" TO_NAME="NODE_1"
TO_TYPE="NOR" />
      </AND>
      </OR>
   </FORBID>
</RULE_DEFINITION>

<REPORTING_ROOT>
   <MESSAGE NAME="Rule %ARG1%: Found %ARG2% node(s) related to this rule.">
      <MESSAGE_ARGUMENT NAME="ARG1" TYPE="ATTRIBUTE" VALUE="ID" />
      <MESSAGE_ARGUMENT NAME="ARG2" TYPE="TOTAL_NODE" VALUE="NODE_1" />
   </MESSAGE>
</REPORTING_ROOT>
</DA_RULE>
```

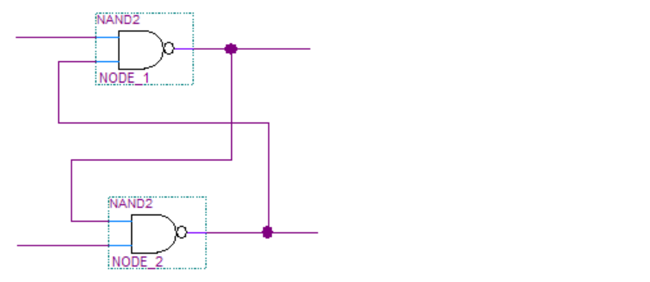
In Example 5–2, the possible SR latch structures are specified in the rule definition section. Codes defined in the `<AND></AND>` block are tied together, meaning that each statement in the block must be true for the block to be fulfilled (AND gate similarity). In the `<OR></OR>` block, as long as one statement in the block is true, the block is fulfilled (OR gate similarity). If no `<AND></AND>` or `<OR></OR>` block is specified, the default is `<AND></AND>`.

The `<FORBID></FORBID>` section contains the undesirable condition for the design, which in this case is the SR latch structures. If the condition is fulfilled, the Design Assistant highlights a rule violation.

The following examples are the undesired conditions from Example 5–2 with their equivalent block diagrams (Figure 5–14 and Figure 5–15):

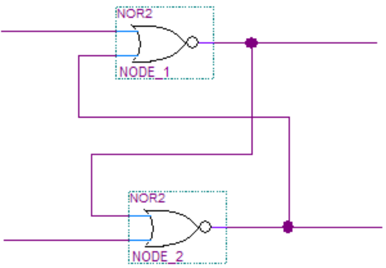
```
<AND>
  <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NAND" TO_NAME="NODE_2"
TO_TYPE="NAND" />
  <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NAND" TO_NAME="NODE_1"
TO_TYPE="NAND" />
</AND>
```

Figure 5–14. Undesired Condition 1




```
<AND>
 <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NOR" TO_NAME="NODE_2" TO_TYPE="NOR" />
 <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NOR" TO_NAME="NODE_1" TO_TYPE="NOR" />
</AND>
```

Figure 5–15. Undesired Condition 2



Relating Nodes to a Clock Domain

Example 5–3 shows how to use the CLOCK_RELATIONSHIP attribute to relate nodes to clock domains. This example checks for correct synchronization in data transfer between asynchronous clock domains. Synchronization is done using cascaded registers, also called synchronizers, at the receiving clock domain. The code in Example 5–3 checks for the synchronizer configuration based on the following guidelines:

- The cascading registers need to be triggered on the same clock edge
- Do not put any logic between the register output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain

Example 5–3. Detecting Incorrect Synchronizer Configuration

```
<DA_RULE ID="EX02" SEVERITY="HIGH" NAME="Data Transfer Not Synch Correctly"
DEFAULT_RUN="YES">

<RULE_DEFINITION>
<DECLARE>
   <NODE NAME="NODE_1" TYPE="REG" />
   <NODE NAME="NODE_2" TYPE="REG" />
   <NODE NAME="NODE_3" TYPE="REG" />
</DECLARE>
<FORBID>
   <NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="ASYN" />
   <NODE_RELATIONSHIP FROM_NAME="NODE_2" TO_NAME="NODE_3" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="!ASYN" />
   <OR>
      <NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
REQUIRED_THROUGH="YES" THROUGH_TYPE="COMB" CLOCK_RELATIONSHIP="ASYN" />
      <CLOCK_RELATIONSHIP NAME="SEQ_EDGE|ASYN" NODE_LIST="NODE_2, NODE_3" />
   </OR>
</FORBID>
</RULE_DEFINITION>

<REPORTING_ROOT>
<MESSAGE NAME="Rule %ARG1%: Found %ARG2% node(s) related to this rule.">
   <MESSAGE_ARGUMENT NAME="ARG1" TYPE="ATTRIBUTE" VALUE="ID" />
   <MESSAGE_ARGUMENT NAME="ARG2" TYPE="TOTAL_NODE" VALUE="NODE_1" />
   <MESSAGE NAME="Source node(s): %ARG3%, Destination node(s): %ARG4%">
      <MESSAGE_ARGUMENT NAME="ARG3" TYPE="NODE" VALUE="NODE_1" />
      <MESSAGE_ARGUMENT NAME="ARG4" TYPE="NODE" VALUE="NODE_2" />
   </MESSAGE>
</MESSAGE>
</REPORTING_ROOT>
</DA_RULE>
```

The codes differentiate the clock domains. ASYN means asynchronous, and !ASYN means non-asynchronous. This notation is useful for describing nodes that are in different clock domains. The following lines from Example 5–3 state that NODE_2 and NODE_3 are in the same clock domain, but NODE_1 is not.

```
<NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="ASYN" />

<NODE_RELATIONSHIP FROM_NAME="NODE_2" TO_NAME="NODE_3" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="!ASYN" />
```

The next line of code states that NODE_2 and NODE_3 have a clock relationship of either sequential edge or asynchronous.

```
 <CLOCK_RELATIONSHIP NAME="SEQ_EDGE|ASYN" NODE_LIST="NODE_2, NODE_3" />
```

The <FORBID></FORBID> section contains the undesirable condition for the design, which in this case is the undesired configuration of the synchronizer. If the condition is fulfilled, the Design Assistant highlights a rule violation.

The following examples are the undesired conditions from Example 5–3 with their equivalent block diagrams (Figure 5–16 and Figure 5–17):

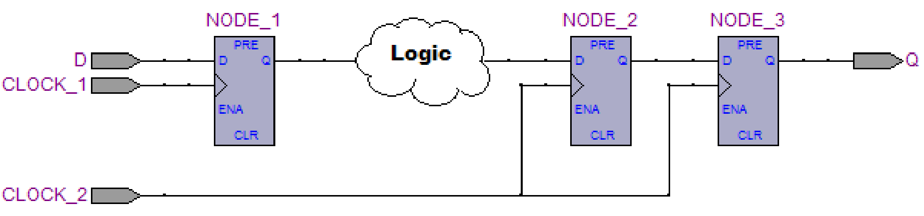
Example 5–4.

```
<NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="ASYN" />

<NODE_RELATIONSHIP FROM_NAME="NODE_2" TO_NAME="NODE_3" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="!ASYN" />

<NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
REQUIRED_THROUGH="YES" THROUGH_TYPE="COMB" CLOCK_RELATIONSHIP="ASYN" />
```

Figure 5–16. Undesired Condition 3



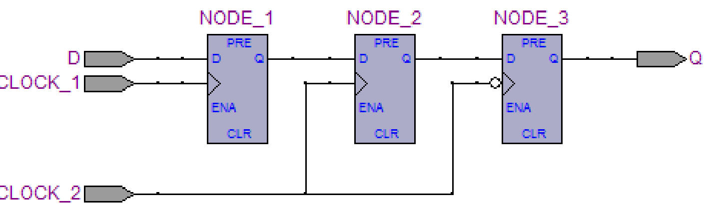
Example 5–5.

```
<NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="ASYN" />

<NODE_RELATIONSHIP FROM_NAME="NODE_2" TO_NAME="NODE_3" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="!ASYN" />

<CLOCK_RELATIONSHIP NAME="SEQ_EDGE|ASYN" NODE_LIST="NODE_2, NODE_3" />
```

Figure 5–17. Undesired Condition 4



Targeting Clock and Register-Control Architectural Features

In addition to following general design guidelines, it is important to code your design with the device architecture in mind. FPGAs provide device-wide clocks and register control signals that can improve performance.

Clock Network Resources

Altera FPGAs provide device-wide global clock routing resources and dedicated inputs. Use the FPGA's low-skew, high fan-out dedicated routing where available. By assigning a clock input to one of these dedicated clock pins or using a Quartus II logic option to assign global routing, you can take advantage of the dedicated routing available for clock signals.

In ASIC design, balancing clock delay as it is distributed across the device is important. Because Altera FPGAs provides device-wide global clock routing resources and dedicated inputs, there is no need to manually balance delays on the clock network.

Altera recommends limiting the number of clocks in your design to the number of dedicated global clock resources available in your FPGA. Clocks feeding multiple locations that do not use global routing may exhibit clock skew across the device that could lead to timing problems. In addition, when you use combinational logic to generate an internal clock, it adds delays on the clock line. In some cases, delay on a clock line can result in a clock skew greater than the data path length between two registers. If the clock skew is greater than the data delay, the timing parameters of the register (such as hold time requirements) are violated and the design will not function correctly.

Current FPGAs offer increasing numbers of global clocks to address large designs with many clock domains. Many large FPGA devices provide dedicated global clock networks, regional clock networks, and dedicated fast regional clock networks. These clocks are typically organized into a hierarchical clock structure that allows many clocks in each device region with low skew and delay. There are typically a number of dedicated clock pins to drive either the global or regional clock networks and both PLL outputs and internal clocks can drive various clock networks.

To reduce clock skew within a given clock domain and ensure that hold times are met within that clock domain, assign each clock signal to one of the global high fan-out, low-skew clock networks in the FPGA device. The Quartus II software automatically uses global routing for high fan-out control signals, PLL outputs, and signals feeding the global clock pins on the device. You can make explicit Global Signal logic option settings by turning on the **Global Signal** option settings. On the Assignments menu, click **Assignment Editor**. Use this option when it is necessary to force the software to use the global routing for particular signals.

To take full advantage of these routing resources, the sources of clock signals in a design (input clock pins or internally-generated clocks) need to drive only the clock input ports of registers. In older Altera device families (such as FLEX® 10K and ACEX® 1K), if a clock signal feeds the data ports of a register, the signal may not be able to use dedicated routing, which can lead to decreased performance and clock skew problems. In general, allowing clock signals to drive the data ports of registers is not considered synchronous design and can complicate timing analysis. Altera does not recommend this practice.

Reset Resources

ASIC designs may use local resets to avoid long routing delays on the signal. Take advantage of the device-wide asynchronous reset pin available on most FPGAs to eliminate these problems. This reset signal provides low-skew routing across the device.

 © March 2009 Altera Corporation

Register Control Signals

Avoid using an asynchronous load signal if the design target device architecture does not include registers with dedicated circuitry for asynchronous loads. Also, avoid using both asynchronous clear and preset if the architecture provides only one of those control signals. Stratix III devices, for example, directly support an asynchronous clear function, but not a preset or load function. When the target device does not directly support the signals, the synthesis or place-and-route software must use combinational logic to implement the same functionality. In addition, if you use signals in a priority other than the inherent priority in the device architecture, combinational logic may be required to implement the desired control signals. Combinational logic is less efficient and can cause glitches and other problems; it is best to avoid these implementations.

For Verilog HDL and VHDL examples of registers with various control signals, and information about the inherent priority order of register control signals in Altera device architecture, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Targeting Embedded RAM Architectural Features

Altera's dedicated memory architecture offers many advanced features that you can target easily using the MegaWizard™ Plug-In Manager or using the recommended HDL coding styles that infer the appropriate RAM megafunction (ALTSYNCRAM or ALTDPRAM). Altera recommends using synchronous memory blocks for your design, so the blocks can be mapped directly into the device dedicated memory blocks. You can choose to use single-port, dual-port, or three-port RAM with a single- or dual-clocking method. Asynchronous memory logic is not inferred as a memory block or placed in the dedicated memory block, but is implemented in regular logic cells.

Altera memory blocks have differing read-during-write behaviors, depending on the targeted device family, memory mode, and block type. Read-during-write behavior refers to read and write from the same memory address in the same clock cycle; for example, you read from the same address to which you write in the same clock cycle.

It is important to check how you specify the memory in your HDL code when you use read-during-write behavior. The HDL code describes that the read returns either the old data at the memory location, or the new data being written to the memory location. The old data refers to the data stored in the memory location. The new data refers to the data that is being written to the memory location.

In some cases, when the device architecture cannot implement the memory behavior described in your HDL code, the memory block is not mapped to the dedicated RAM blocks, or the memory block is implemented using extra logic in addition to the dedicated RAM block. Altera recommends that you implement the read-during-write behavior using single-port RAM in Arria® GX devices and the Stratix and Cyclone series of devices to avoid this extra logic implementation.

For Verilog HDL and VHDL examples and guidelines for inferring RAM functions that match the dedicated memory architecture in Altera devices, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

In many synthesis tools, you can specify that the read-during-write behavior is not important to your design; if, for example, you never read and write from the same address in the same clock cycle. For Quartus II integrated synthesis, add the synthesis attribute `ramstyle="no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM, rather than using the read-during-write behavior specified in your HDL code. Using this type of attribute prevents the synthesis tool from using extra logic to implement the memory block and, in some cases, can allow memory inference when it would otherwise be impossible.

For details about using the `ramstyle` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about the synthesis attributes in other synthesis tools, refer to your synthesis tool documentation, or to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Conclusion

Following the design practices described in this chapter can help you to consistently meet your design goals. Asynchronous design techniques may result in incomplete timing analysis, may cause glitches on data signals, and may rely on propagation delays in a device leading to race conditions and unpredictable results. Taking advantage of the architectural features in your FPGA device can also improve the quality of your results.

Referenced Documents

This chapter references the following documents:

- *AN 437: Power Optimization in Stratix III FPGAs*
- *AN 514: Power Optimization in Stratix IV FPGAs*
- *Design Guidelines for HardCopy Series Devices* chapter in the *HardCopy Series Handbook*
- *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *PowerPlay Power Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*

Document Revision History

Table 5–4 shows the revision history for this chapter.

Table 5–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change to content.	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	■ Changed to 8-1/2 x 11 page size. ■ Added new section "Custom Rules Coding Examples" on page 5–37 ■ Added paragraph to "Recommended Clock-Gating Methods" on page 5–11. ■ Added new section: "Design Techniques to Save Power" on page 5–12.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Updated Figure 5–9 on page 5–13; added custom rules file to the flow ■ Added notes to Figure 5–9 on page 5–13 ■ Added new section: "Custom Rules Report" on page 5–34 ■ Added new section: "Custom Rules" on page 5–34 ■ Added new section: "Targeting Embedded RAM Architectural Features" on page 5–38 ■ Minor editorial updates throughout the chapter ■ Added hyperlinks to referenced documents throughout the chapter	Updated for the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

6. Recommended HDL Coding Styles

QII51007-9.0.0

Introduction

HDL coding styles can have a significant effect on the quality of results that you achieve for programmable logic designs. Synthesis tools optimize HDL code for both logic utilization and performance. However, sometimes the best optimizations require human understanding of the design, and synthesis tools have no information about the purpose or intent of the design. You are often in the best position to improve your quality of results.

This chapter addresses HDL coding style recommendations to ensure optimal synthesis results when targeting Altera® devices, including the following sections:

- "Quartus II Language Templates" on page 6–1
- "Using Altera Megafunctions" on page 6–2
- "Instantiating Altera Megafunctions in HDL Code" on page 6–3
- "Inferring Multiplier and DSP Functions from HDL Code" on page 6–6
- "Inferring Memory Functions from HDL Code" on page 6–12
- "Coding Guidelines for Registers and Latches" on page 6–36
- "General Coding Guidelines" on page 6–46
- "Designing with Low-Level Primitives" on page 6–71

For additional guidelines about structuring your design, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*. For additional hand-crafted techniques, you can use to optimize design blocks for the adaptive logic modules (ALMs) in many Altera devices, including a collection of circuit building blocks and related discussions, refer to the *Advanced Synthesis Cookbook: A Design Guide for Stratix II, Stratix III, and Stratix IV Devices*.

Altera's website also provides design examples for other types of functions and to target specific applications. Refer to the Design Examples page and the Reference Designs page.

For style recommendations, options, or HDL attributes specific to your synthesis tool (including Quartus® II integrated synthesis and other EDA tools), refer to the tool vendor's documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

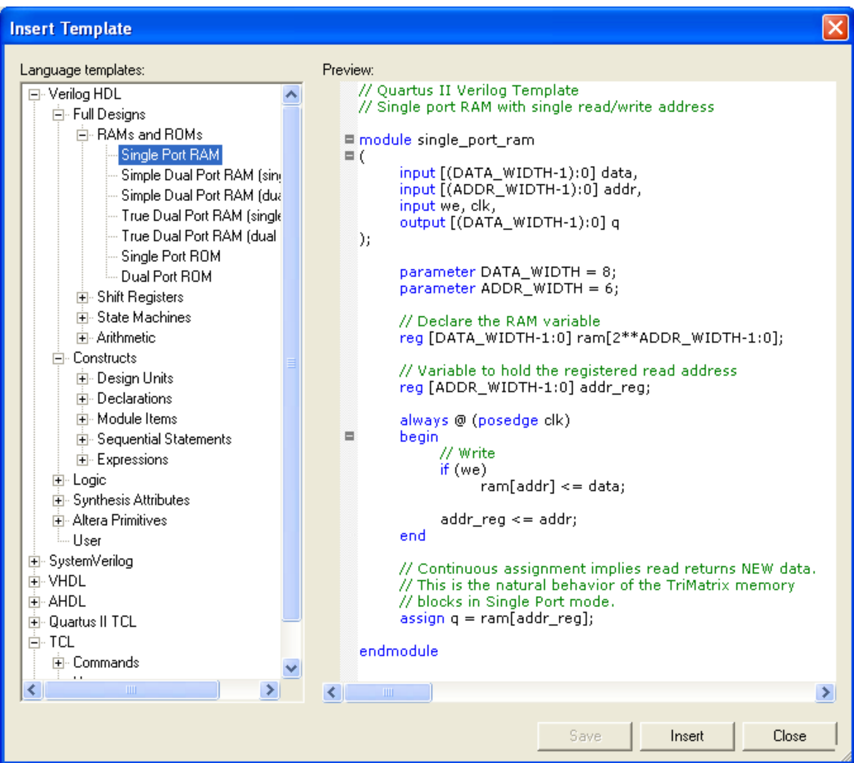
Quartus II Language Templates

The Quartus II software provides Verilog HDL, VHDL, AHDL, Tcl script, and megafunction language templates that can help you with your design.

Many of the Verilog HDL and VHDL examples in this document correspond with examples in the templates. You can easily insert examples from this document into your HDL source code using the **Insert Template** dialog box in the Quartus II user interface, shown in Figure 6–1.

To open the **Insert Template** dialog box when you have a file open in the Quartus II Text Editor, on the Edit menu, click **Insert Template**. Alternatively, you can right-click in the Text Editor window and choose **Insert Template**.

Figure 6–1. Insert Template Dialog Box



Using Altera Megafunctions

Altera provides parameterizable megafunctions that are optimized for Altera device architectures. Using megafunctions instead of coding your own logic saves valuable design time. Additionally, the Altera-provided megafunctions may offer more efficient logic synthesis and device implementation. You can scale the megafunction's size and set various options by setting parameters. Megafunctions include the library of parameterized modules (LPM) and Altera device-specific megafunctions.

To use megafunctions in your HDL code, you can instantiate them as described in "Instantiating Altera Megafunctions in HDL Code" on page 6–3.

© March 2009 Altera Corporation

Sometimes it is preferable to make your code independent of device family or vendor. In this case, you might not want to instantiate megafunctions directly. For some types of logic functions, such as memories and DSP functions, you can infer a megafunction instead of instantiating it. Synthesis tools, including Quartus II integrated synthesis, recognize certain types of HDL code and automatically infer the appropriate megafunction. The synthesis tool uses the Altera megafunction code when compiling your design—even when you do not specifically instantiate the megafunction. Synthesis tools infer megafunctions to take advantage of logic that is optimized for Altera devices or to target dedicated architectural blocks.

In cases where you prefer to use generic HDL code instead of instantiating a megafunction, follow the guidelines and coding examples in "Inferring Multiplier and DSP Functions from HDL Code" on page 6–6 and "Inferring Memory Functions from HDL Code" on page 6–12 to ensure your HDL code infers the appropriate Altera megafunction.

You must use megafunctions to access some Altera device-specific architecture features. You can infer or instantiate megafunctions to target some features such as memory and DSP blocks. You must instantiate megafunctions to target certain device and high-speed features such as LVDS drivers, phase-locked loops (PLLs), transceivers, and double-data rate input/output (DDIO) circuitry.

For some designs, generic HDL code can provide better results than instantiating a megafunction. Refer to the following general guidelines and examples that describe when to use standard HDL code and when to use megafunctions:

- For simple addition or subtraction functions, use the + or − symbol instead of an LPM function. Instantiating an LPM function for simple arithmetic operations can result in a less efficient result because the function is hard coded and the synthesis algorithms cannot take advantage of basic logic optimizations.
- For simple multiplexers and decoders, use array notation (such as `out = data[sel]`) instead of an LPM function. Array notation works very well and has simple syntax. You can use the `lpm_mux` function to take advantage of architectural features such as cascade chains in APEX™ series devices, but use the LPM function only if you understand the device architecture in detail and want to force a specific implementation.
- Avoid division operations where possible. Division is an inherently slow operation. Many designers use multiplication creatively to produce division results.

Instantiating Altera Megafunctions in HDL Code

The following sections describe how to use megafunctions by instantiating them in your HDL code with the following methods:

- "Instantiating Megafunctions Using the MegaWizard Plug-In Manager"—You can use the MegaWizard™ Plug-In Manager to parameterize the function and create a wrapper file.
- "Creating a Netlist File for Other Synthesis Tools"—You can optionally create a netlist file instead of a wrapper file.

- "Instantiating Megafunctions Using the Port and Parameter Definition"—You can instantiate the function directly in your HDL code.

Instantiating Megafunctions Using the MegaWizard Plug-In Manager

Use the MegaWizard Plug-In Manager as described in this section to create megafunctions in the Quartus II GUI that you can instantiate in your HDL code. The MegaWizard Plug-In Manager provides a GUI to customize and parameterize megafunctions, and ensures that you set all megafunction parameters properly. When you finish setting parameters, you can specify which files you want generated. Depending on which language you choose, the MegaWizard Plug-In Manager instantiates the megafunction with the correct parameters and generates a megafunction variation file (wrapper file) in Verilog HDL (**.v**), VHDL (**.vhd**), or AHDL (**.tdf**), along with other supporting files.

The MegaWizard Plug-In Manager provides options to create the following files:

- A sample instantiation template for the language of the variation file (**_inst.v** | **vhd** | **tdf**).
- Component Declaration File (**.cmp**) that can be used in VHDL Design Files
- ADHL Include File (**.inc**) that can be used in Text Design Files (**.tdf**)
- Quartus II Block Symbol File (**.bsf**) for schematic designs
- Verilog HDL module declaration file that can be used when instantiating the megafunction as a black box in a third-party synthesis tool (**_bb.v**).
- If you enable the option to generate a synthesis area and timing estimation netlist, the MegaWizard Plug-In Manager generates an additional synthesis netlist file (**_syn.v**). Refer to "Creating a Netlist File for Other Synthesis Tools" on page 6–5 for details.

Table 6–1 lists and describes the files generated by the MegaWizard Plug-In Manager.

Table 6–1. MegaWizard Plug-In Manager Generated Files (Part 1 of 2)

File	Description
*<output file>***.v** (1)	Verilog HDL Variation Wrapper File—Megafunction wrapper file for instantiation in a Verilog HDL design.
*<output file>***.vhd** (1)	VHDL Variation Wrapper File—Megafunction wrapper file for instantiation in a VHDL design.
*<output file>***.tdf** (1)	AHDL Variation Wrapper File—Megafunction wrapper file for instantiation in an AHDL design.
*<output file>***.inc**	ADHL Include File—Used in AHDL designs.
*<output file>***.cmp**	Component Declaration File—Used in VHDL designs.
*<output file>***.bsf**	Block Symbol File—Used in Quartus II Block Design Files (**.bdf**).
*<output file>***_inst.v**	Verilog HDL Instantiation Template—Sample Verilog HDL instantiation of the module in the megafunction wrapper file.
*<output file>***_inst.vhd**	VHDL Instantiation Template—Sample VHDL instantiation of the entity in the megafunction wrapper file.
*<output file>***_inst.tdf**	Text Design File Instantiation Template—Sample AHDL instantiation of the subdesign in the megafunction wrapper file.

Table 6–1. MegaWizard Plug-In Manager Generated Files (Part 2 of 2)

File	Description
*<output file>*_**bb.v**	Black box Verilog HDL Module Declaration—Hollow-body module declaration that can be used in Verilog HDL designs to specify port directions when creating black boxes in third-party synthesis tools.
*<output file>*_**syn.v** (2)	Synthesis area and timing estimation netlist—Megafunction netlist may be used by third-party synthesis tools to improve area and timing estimations.

Notes to Table 6–1:

(1) The MegaWizard Plug-In Manager generates either the **.v**, **.vhd**, or **.edf** ile, depending on the language you select for the output file on the megafunction-selection page of the wizard.

(2) The MegaWizard Plug-In Manager generates this file only if you turn on the **Generate netlist** option under **Timing and resource estimation** on the **EDA** page of the wizard.

Creating a Netlist File for Other Synthesis Tools

When you use certain megafunctions with third-party EDA synthesis tools (that is, tools other than Quartus II integrated synthesis), you can optionally create a netlist for area and timing estimation instead of a wrapper file.

The netlist file is a representation of the customized logic used in the Quartus II software. The file provides the connectivity of architectural elements in the megafunction but may not represent true functionality. This information enables certain third-party synthesis tools to better report area and timing estimates. In addition, synthesis tools can use the timing information to focus timing-driven optimizations and improve the quality of results.

To generate the netlist, turn on **Generate a synthesis area and timing estimation netlist** on the EDA page of the MegaWizard Plug-In Manager. The netlist file is called <*output file*>**_syn.v**. If you use this netlist for synthesis, you must include the megafunction wrapper file <*output file*>**.v** | **vhd** in your Quartus II project for placement and routing.

Your synthesis tool may call the Quartus II software in the background to generate this netlist, so you might not be required to perform the extra step of turning on this option.

For information about support for area and timing estimation netlists in your synthesis tool, refer to the tool vendor's documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Instantiating Megafunctions Using the Port and Parameter Definition

You can instantiate the megafunction directly in your Verilog HDL, VHDL, or AHDL code by calling the megafunction and setting its parameters as you would any other module, component, or subdesign.

Refer to the specific megafunction in the Quartus II Help for a list of the megafunction ports and parameters. The Quartus II Help also provides a sample VHDL component declaration and AHDL function prototype for each megafunction.

Altera strongly recommends that you use the MegaWizard Plug-In Manager for complex megafunctions such as PLLs, transceivers, and LVDS drivers. For details about using the MegaWizard Plug-In Manager, refer to "Instantiating Megafunctions Using the MegaWizard Plug-In Manager" on page 6–4.

Inferring Multiplier and DSP Functions from HDL Code

The following sections describe how to infer multiplier and DSP functions from generic HDL code, and, if applicable, how to target the dedicated DSP block architecture in Altera devices:

- "Multipliers—Inferring the LPM_MULT Megafunction from HDL Code"
- "Multiply-Accumulators and Multiply-Adders—Inferring ALTMULT_ACCUM and ALTMULT_ADD Megafunctions from HDL Code" on page 6–8

For synthesis tool features and options, refer to your synthesis tool documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook.*

For more design examples involving advanced multiply functions and complex DSP functions, refer to the DSP Design Examples page on Altera's website.

Multipliers—Inferring the LPM_MULT Megafunction from HDL Code

To infer multiplier functions, synthesis tools look for multipliers and convert them to LPM_MULT or ALTMULT_ADD megafunctions, or may map them directly to device atoms. For devices with DSP blocks, the software can implement the function in a DSP block instead of logic, depending on device utilization. The Quartus II Fitter can also place input and output registers in DSP blocks (that is, perform register packing) to improve performance and area utilization.

For additional information about the DSP block and the supported functions, refer to the appropriate Altera device family handbook and Altera's DSP Solutions Center website.

Example 6–1 and Example 6–2 show Verilog HDL code examples, and Example 6–3 and Example 6–4 show VHDL code examples, for unsigned and signed multipliers that synthesis tools can infer as an LPM_MULT or ALTMULT_ADD megafunction. Each example fits into one DSP block 9-bit element. In addition, when register packing occurs, no extra logic cells for registers are required.

The `signed` declaration in Verilog HDL is a feature of the Verilog 2001 Standard.

Example 6–1. Verilog HDL Unsigned Multiplier

```
module unsigned_mult (out, a, b);
   output [15:0] out;
   input  [7:0] a;
   input  [7:0] b;
   assign out = a * b;
endmodule
```


© March 2009 Altera Corporation

Example 6–2. Verilog HDL Signed Multiplier with Input and Output Registers (Pipelining = 2)

```
module signed_mult (out, clk, a, b);
   output [15:0] out;
   input clk;
   input signed [7:0] a;
   input signed [7:0] b;

   reg signed [7:0] a_reg;
   reg signed [7:0] b_reg;
   reg signed [15:0] out;
   wire signed [15:0] mult_out;

   assign mult_out = a_reg * b_reg;

   always @ (posedge clk)
   begin
      a_reg <= a;
      b_reg <= b;
      out <= mult_out;
   end
endmodule
```

Example 6–3. VHDL Unsigned Multiplier with Input and Output Registers (Pipelining = 2)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY unsigned_mult IS
   PORT (
      a: IN UNSIGNED (7 DOWNTO 0);
      b: IN UNSIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      result: OUT UNSIGNED (15 DOWNTO 0)
   );
END unsigned_mult;

ARCHITECTURE rtl OF unsigned_mult IS
   SIGNAL a_reg, b_reg: UNSIGNED (7 DOWNTO 0);
BEGIN
   PROCESS (clk, aclr)
   BEGIN
      IF (aclr ='1') THEN
         a_reg <= (OTHERS => '0');
         b_reg <= (OTHERS => '0');
         result <= (OTHERS => '0');
      ELSIF (clk'event AND clk = '1') THEN
         a_reg <= a;
         b_reg <= b;
         result <= a_reg * b_reg;
      END IF;
   END PROCESS;
END rtl;
```

Example 6–4. VHDL Signed Multiplier

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY signed_mult IS
   PORT (
      a: IN SIGNED (7 DOWNTO 0);
      b: IN SIGNED (7 DOWNTO 0);
      result: OUT SIGNED (15 DOWNTO 0)
   );
END signed_mult;

BEGIN
   result <= a * b;
END rtl;
```

Multiply-Accumulators and Multiply-Adders—Inferring ALTMULT_ACCUM and ALTMULT_ADD Megafunctions from HDL Code

Synthesis tools detect multiply-accumulators or multiply-adders and convert them to ALTMULT_ACCUM or ALTMULT_ADD megafunctions, respectively, or may map them directly to device atoms. The Quartus II software then places these functions in DSP blocks during placement and routing.

Synthesis tools infer multiply-accumulator and multiply-adder functions only if the Altera device family has dedicated DSP blocks that support these functions.

A simple multiply-accumulator consists of a multiplier feeding an addition operator. The addition operator feeds a set of registers that then feeds the second input to the addition operator. A simple multiply-adder consists of two to four multipliers feeding one or two levels of addition, subtraction, or addition/subtraction operators. Addition is always the second-level operator, if it is used. In addition to the multiply-accumulator and multiply-adder, the Quartus II Fitter also places input and output registers into the DSP blocks to pack registers and improve performance and area utilization.

Some device families offer additional advanced multiply-add and accumulate functions, such as complex multiplication, input shift register, or larger multiplications.

For details about advanced DSP block features, refer to the appropriate device handbook. For more design examples involving DSP functions and inferring advanced features in the multiply-add and multiply-accumulate circuitry, refer to the DSP Design Examples page on Altera’s website.

The Verilog HDL and VHDL code samples shown in Example 6–5 through Example 6–8 infer multiply-accumulators and multiply-adders with input, output, and pipeline registers as well as an optional asynchronous clear signal. Using the three sets of registers provides the best performance through the function, with a latency of 3. You can remove the registers in your design to reduce the latency.

© March 2009 Altera Corporation

Example 6–5. Verilog HDL Unsigned Multiply-Accumulator

```
module unsig_altmult_accum (dataout, dataa, datab, clk, aclr, clken);
   input [7:0] dataa;
   input [7:0] datab;
   input clk;
   input aclr;
   input clken;
   output [31:0] dataout;

   reg [31:0] dataout;
   reg [7:0] dataa_reg;
   reg [7:0] datab_reg;
   reg [15:0] multa_reg;
   wire [15:0] multa;
   wire [31:0] adder_out;
   assign multa = dataa_reg * datab_reg;
   assign adder_out = multa_reg + dataout;

   always @ (posedge clk or posedge aclr)
   begin
      if (aclr)
      begin
         dataa_reg <= 8'b0;
         datab_reg <= 8'b0;
         multa_reg <= 16'b0;
         dataout <= 32'b0;
      end
      else if (clken)
      begin
         dataa_reg <= dataa;
         datab_reg <= datab;
         multa_reg <= multa;
         dataout <= adder_out;
      end
   end
endmodule
```

Example 6–6. Verilog HDL Signed Multiply-Adder

```
module sig_altmult_add (dataa, datab, datac, datad, clock, aclr,
result);
   input signed [15:0] dataa, datab, datac, datad;
   input clock, aclr;
   output reg signed [32:0] result;

   reg signed [15:0] dataa_reg, datab_reg, datac_reg, datad_reg;
   reg signed [31:0] mult0_result, mult1_result;

   always @ (posedge clock or posedge aclr) begin
       if (aclr) begin
           dataa_reg <= 16'b0;
           datab_reg <= 16'b0;
           datac_reg <= 16'b0;
           datad_reg <= 16'b0;
           mult0_result <= 32'b0;
           mult0_result <= 32'b0;
           result <= 33'b0;
        end
        else begin
           dataa_reg <= dataa;
           datab_reg <= datab;
           datac_reg <= datac;
           datad_reg <= datad;
           mult0_result <= dataa_reg * datab_reg;
           mult1_result <= datac_reg * datad_reg;
           result <= mult0_result + mult1_result;
       end
   end
endmodule
```

Example 6–7. VHDL Signed Multiply-Accumulator

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY sig_altmult_accum IS
   PORT (
      a: IN SIGNED(7 DOWNTO 0);
      b: IN SIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      accum_out: OUT SIGNED (15 DOWNTO 0)
   ) ;
END sig_altmult_accum;

ARCHITECTURE rtl OF sig_altmult_accum IS
   SIGNAL a_reg, b_reg: SIGNED (7 DOWNTO 0);
   SIGNAL pdt_reg: SIGNED (15 DOWNTO 0);
   SIGNAL adder_out: SIGNED (15 DOWNTO 0);
BEGIN
   PROCESS (clk, aclr)
   BEGIN
      IF (aclr = '1') then
          a_reg <= (others => '0');
          b_reg <= (others => '0');
          pdt_reg <= (others => '0');
          adder_out <= (others => '0');
      ELSIF (clk'event and clk = '1') THEN
          a_reg <= (a);
          b_reg <= (b);
          pdt_reg <= a_reg * b_reg;
          adder_out <= adder_out + pdt_reg;
      END IF;
   END process;
   accum_out <= adder_out;
END rtl;
```

Example 6–8. VHDL Unsigned Multiply-Adder

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY unsignedmult_add IS
   PORT (
      a: IN UNSIGNED (7 DOWNTO 0);
      b: IN UNSIGNED (7 DOWNTO 0);
      c: IN UNSIGNED (7 DOWNTO 0);
      d: IN UNSIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      result: OUT UNSIGNED (15 DOWNTO 0)
   );
END unsignedmult_add;

ARCHITECTURE rtl OF unsignedmult_add IS
   SIGNAL a_reg, b_reg, c_reg, d_reg: UNSIGNED (7 DOWNTO 0);
   SIGNAL pdt_reg, pdt2_reg: UNSIGNED (15 DOWNTO 0);
   SIGNAL result_reg: UNSIGNED (15 DOWNTO 0);
BEGIN
   PROCESS (clk, aclr)
   BEGIN
      IF (aclr = '1') THEN
         a_reg <= (OTHERS => '0');
         b_reg <= (OTHERS => '0');
         c_reg <= (OTHERS => '0');
         d_reg <= (OTHERS => '0');
         pdt_reg <= (OTHERS => '0');
         pdt2_reg <= (OTHERS => '0');

      ELSIF (clk'event AND clk = '1') THEN
         a_reg <= a;
         b_reg <= b;
         c_reg <= c;
         d_reg <= d;
         pdt_reg <= a_reg * b_reg;
         pdt2_reg <= c_reg * d_reg;
         result_reg <= pdt_reg + pdt2_reg;
      END IF;
      END PROCESS;
   result <= result_reg;
END rtl;
```

Inferring Memory Functions from HDL Code

The following sections describe how to infer memory functions from generic HDL code and, if applicable, to target the dedicated memory architecture in Altera devices:

- "RAM Functions—Inferring ALTSYNCRAM and ALTDPRAM Megafunctions from HDL Code" on page 6–13
- "ROM Functions—Inferring ALTSYNCRAM and LPM_ROM Megafunctions from HDL Code" on page 6–29
- "Shift Registers—Inferring the ALTSHIFT_TAPS Megafunction from HDL Code" on page 6–32

For synthesis tool features and options, refer to your synthesis tool documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Altera's dedicated memory architecture offers a number of advanced features that can be easily targeted using the MegaWizard Plug-In Manager, as described in "Instantiating Altera Megafunctions in HDL Code" on page 6–3. The coding recommendations in the following sections provide portable examples of generic HDL code that infer the appropriate megafunction. However, if you want to use some of the advanced memory features in Altera devices, consider using the megafunction directly so that you can control the ports and parameters more easily.

RAM Functions—Inferring ALTSYNCRAM and ALTDPRAM Megafunctions from HDL Code

To infer RAM functions, synthesis tools detect sets of registers and logic that can be replaced with the ALTSYNCRAM or ALTDPRAM megafunctions for device families that have dedicated RAM blocks, or may map them directly to device memory atoms. Tools typically consider all signals and variables that have a two-dimensional array type and then create a RAM block, if applicable, based on the way the signals, variables, or both are assigned, referenced, or both in the HDL source description. This section provides examples demonstrating the coding styles that are inferred to create a memory block.

Standard synthesis tools recognize single-port and simple dual-port (one read port and one write port) RAM blocks. Some tools (such as the Quartus II software) also recognize true dual-port RAM blocks that map to the memory blocks in certain Altera devices. Tools usually do not infer small RAM blocks because small RAM blocks typically can be implemented more efficiently using the registers in regular logic.

If you are using Quartus II integrated synthesis, you can direct the software to infer ROM blocks for all sizes with the **Allow Any RAM Size for Recognition** option under **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box.

If your design contains a RAM block that your synthesis tool does not recognize and infer, the design might require a large amount of system memory that can potentially cause compilation problems.

Some synthesis tools provide options to control the implementation of inferred RAM blocks for Altera devices with TriMatrix memory blocks. For example, Quartus II integrated synthesis provides the `ramstyle` synthesis attribute to specify the type of memory block or to specify the use of regular logic instead of a dedicated memory block. Quartus II integrated synthesis does not map inferred memory into Stratix® III MLABs unless the HDL code specifies the appropriate `ramstyle` attribute, although the Fitter may map some memories to MLABs.

For details about using the `ramstyle` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about synthesis attributes in other synthesis tools, refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

When you use a formal verification flow, Altera recommends that you create RAM blocks in separate entities or modules that contain only the RAM logic. In certain formal verification flows, for example, when using Quartus II integrated synthesis, the entity or module containing the inferred RAM is put into a black box automatically because formal verification tools do not support RAM blocks. The Quartus II software issues a warning message when this occurs. If the entity or module contains any additional logic outside the RAM block, this logic also must be treated as a black box for formal verification and therefore cannot be verified.

The following subsections present several guidelines for inferring RAM functions that match the dedicated memory architecture in Altera devices, and then provide recommended HDL code for different types of memory logic.

Use Synchronous Memory Blocks

Altera recommends using synchronous memory blocks for Altera designs. The TriMatrix memory blocks in Altera's newest devices are synchronous, so RAM designs that are targeted towards architectures that contain these dedicated memory blocks must be synchronous to be mapped directly into the device architecture. For these devices, asynchronous memory logic is implemented in regular logic cells.

Synchronous memories are supported in all Altera device families. A memory block is considered synchronous if it uses one of the following read behaviors:

- Memory read occurs in a Verilog always block with a clock signal or a VHDL clocked process.
- Memory read occurs outside a clocked block, but there is a synchronous read address (that is, the address used in the read statement is registered). This type of logic is not always inferred as a memory block, depending on the target device architecture.

The synchronous memory structures in Altera devices differ from the structures in other vendors' devices. Match your design to the target device architecture to achieve the best results.

Later subsections provide coding recommendations for various memory types. All of these examples are synchronous to ensure that they can be directly mapped into the dedicated memory architecture available in Altera FPGAs.

For additional information about the dedicated memory blocks in your specific device, refer to the appropriate Altera device family data sheet on the Altera website at www.altera.com.

Avoid Unsupported Reset and Control Conditions

To ensure that your HDL code can be implemented in the target device architecture, avoid unsupported reset conditions or other control logic that does not exist in the device architecture.

The RAM contents of Altera memory blocks cannot be cleared with a reset signal during device operation. If your HDL code describes a RAM with a reset signal for the RAM contents, the logic is implemented in regular logic cells instead of a memory block. As a general rule, avoid putting RAM read or write operations in an always block or process block with a reset signal. If you want to specify memory contents, initialize the memory as described in "Specifying Initial Memory Contents at Power-Up" on page 6–27 or write the data to the RAM during device operation.

Example 6–9 shows an example of undesirable code where there is a reset signal that clears part of the RAM contents. Avoid this coding style because it is not supported in Altera memories.

Example 6–9. Verilog RAM with Reset Signal that Clears RAM Contents: Not Supported in Device Architecture

```
module clear_ram
(
    input clock,
    input reset,
    input we,
    input [7:0] data_in,
    input [4:0] address,
    output reg [7:0] data_out
);

    reg [7:0] mem [0:31];
    integer i;

    always @ (posedge clock or posedge reset)
    begin
        if (reset == 1'b1)
            mem[address] <= 0;
        else if (we == 1'b1)
            mem[address] <= data_in;

        data_out <= mem[address];
    end
endmodule
```

Example 6–10 shows an example of undesirable code where the reset signal affects the RAM, although the effect may not be intended. Avoid this coding style because it is not supported in Altera memories.

Example 6–10. Verilog RAM with Reset Signal that Affects RAM: Not Supported in Device Architecture

```
module bad_reset
(
    input clock,
    input reset,
    input we,
    input [7:0] data_in,
    input [4:0] address,
    output reg [7:0] data_out,
    input d,
    output reg q
);

    reg [7:0] mem [0:31];
    integer i;

    always @ (posedge clock or posedge reset)
    begin
        if (reset == 1'b1)
            q <= 0;
        else
        begin
            if (we == 1'b1)
                mem[address] <= data_in;

            data_out <= mem[address];
            q <= d;
        end
    end
endmodule
```

In addition to reset signals, other control logic can prevent memory logic from being inferred as a memory block. For example, you cannot use a clock enable on the read address registers in Stratix devices, because doing so affects the output latch of the RAM, and therefore the synthesized result in the device RAM architecture would not match the HDL description. In Stratix II, Cyclone® II, Arria® GX, and other newer devices, however, you can use the address stall feature as a read address clock enable, so there is no such limitation. Check the documentation on your device architecture to ensure that your code matches the hardware available in the device.

Check Read-During-Write Behavior

It is important to check the read-during-write behavior of the memory block described in your HDL design as compared to the behavior in your target device architecture. Your HDL source code specifies the memory behavior when you read and write from the same memory address in the same clock cycle. The code specifies that the read returns either the old data at the address, or the new data being written to the address. This is referred to as the read-during-write behavior of the memory block. Altera memory blocks have different read-during-write behavior depending on the target device family, memory mode, and block type.

Synthesis tools map an HDL design into the target device architecture, with the goal of maintaining the functionality described in your source code. Therefore, if your source code specifies unsupported read-during-write behavior for the device RAM blocks, the software must implement the logic outside the RAM hardware in regular logic cells.

One common problem occurs when there is a continuous read in the HDL code, as shown in the following samples. You should avoid using these coding styles:

```
//Verilog HDL concurrent signal assignment
assign q = ram[raddr_reg];

-- VHDL concurrent signal assignment
q <= ram(raddr_reg);
```

When a write operation occurs, this type of HDL implies that the read should immediately reflect the new data at the address, independent of the read clock. However, that is not the behavior of TriMatrix memory blocks. In the device architecture, the new data is not available until the next edge of the read clock. Therefore, if the synthesis tool mapped the logic directly to a TriMatrix memory block, the device functionality and gate-level simulation results would not match the HDL description or function simulation results. If the write clock and read clock are the same, the synthesis tool can infer memory blocks and add extra bypass logic so that the device behavior does match the HDL behavior. If the write and read clocks are different, the synthesis tool cannot reliably add bypass logic, so the logic is implemented in regular logic cells instead of dedicated RAM blocks. The examples in the following sections discuss some of these differences for read-during-write conditions.

In addition, the MLAB feature in Stratix III logic array blocks (LABs) does not easily support old data or new data behavior for a read during write in the dedicated device architecture. Implementing the extra logic to support this behavior significantly reduces timing performance through the memory.

For best performance in MLAB memories, your design should not depend on the read data during a write operation.

In many synthesis tools, you can specify that the read-during-write behavior is not important to your design; for example, if you never read from the same address to which you write in the same clock cycle. For Quartus II integrated synthesis, add the synthesis attribute `ramstyle="no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM, rather than use the behavior specified by your HDL code. Using this type of attribute prevents the synthesis tool from using extra logic to implement the memory block, and in some cases, can allow memory inference when it would otherwise be impossible.

For more information about attribute syntax, the `no_rw_check` attribute value, or specific options for your synthesis tool, refer to your synthesis tool documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

The following subsections provide coding recommendations for various memory types. Each example describes the read-during-write behavior and addresses the support for the memory type in Altera devices.

Single-Clock Synchronous RAM with Old Data Read-During-Write Behavior

The code examples in this section show Verilog HDL and VHDL code that infers simple dual-port, single-clock synchronous RAM. Single-port RAM blocks use a similar coding style.

The read-during-write behavior in these examples is to read the old data at the memory address. Refer to "Check Read-During-Write Behavior" on page 6–16 for details. Altera recommends that you use this coding style for most RAM blocks as long as your design does not require that a simultaneous read and write to the same RAM location read the new value that is currently being written to that RAM location. For best performance in MLAB memories, use the appropriate attribute so that your design does not depend on the read data during a write operation.

If you require that the read-during-write results in new data, refer to "Single-Clock Synchronous RAM with New Data Read-During-Write Behavior" on page 6–19.

The simple dual-port RAM code samples shown in Example 6–11 and Example 6–12 map directly into Altera TriMatrix memory.

Single-port versions of memory blocks (that is, using the same read address and write address signals) can allow better RAM utilization than dual-port memory blocks, depending on the device family.

Example 6–11. Verilog HDL Single-Clock Simple Dual-Port Synchronous RAM with Old Data Read-During-Write Behavior

```
module single_clk_ram(
    output reg [7:0] q,
    input [7:0] d,
    input [6:0] write_address, read_address,
    input we, clk
);
    reg [7:0] mem [127:0];

    always @ (posedge clk) begin
        if (we)
            mem[write_address] <= d;
        q <= mem[read_address]; // q doesn't get d in this clock cycle
    end
endmodule
```

Example 6–12. VHDL Single-Clock Simple Dual-Port Synchronous RAM with Old Data Read-During-Write Behavior

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY single_clock_ram IS
    PORT (
        clock: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        write_address: IN INTEGER RANGE 0 to 31;
        read_address: IN INTEGER RANGE 0 to 31;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END single_clock_ram;

ARCHITECTURE rtl OF single_clock_ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL ram_block: MEM;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
            q <= ram_block(read_address);
            -- VHDL semantics imply that q doesn't get data
            -- in this clock cycle
        END IF;
    END PROCESS;
END rtl;
```

Single-Clock Synchronous RAM with New Data Read-During-Write Behavior

The examples in this section describe RAM blocks in which a simultaneous read and write to the same location reads the new value that is currently being written to that RAM location.

To implement this behavior in the target device, synthesis software adds bypass logic around the RAM block. This bypass logic increases the area utilization of the design and decreases the performance if the RAM block is part of the design's critical path. Refer to "Check Read-During-Write Behavior" on page 6–16 for details. If this behavior is not required for your design, use the examples from "Single-Clock Synchronous RAM with Old Data Read-During-Write Behavior" on page 6–17.

The simple dual-port RAM shown in Example 6–13 and Example 6–14 require the software to create bypass logic around the RAM block.

Single-port versions of the Verilog memory block (that is, using the same read address and write address signals) do not require any logic cells to create bypass logic in the Arria GX, Stratix, and Cyclone series of devices, because the device memory supports new data read-during-write behavior when in single-port mode (same clock, same read, and write address).

Example 6–13. Verilog HDL Single-Clock Simple Dual-Port Synchronous RAM with New Data Read-During-Write Behavior

```
module single_clock_wr_ram(
    output reg [7:0] q,
    input [7:0] d,
    input [6:0] write_address, read_address,
    input we, clk
);
    reg [7:0] mem [127:0];

    always @ (posedge clk) begin
        if (we)
            mem[write_address] = d;
        q = mem[read_address]; // q does get d in this clock cycle if
we is high
    end
endmodule
```

Example 6–13 is similar to Example 6–11, but Example 6–13 uses a blocking assignment for the write so that the data is assigned immediately.

An alternative way to create a single-clock RAM is to use an assign statement to read the address of mem to create the output q, as shown in the following coding style. By itself, the code describes new data read-during-write behavior. However, if the RAM output feeds a register in another hierarchy, then a read-during-write would result in the old data. Synthesis tools may not infer a RAM block if the tool cannot determine which behavior is described, such as when the memory feeds a hard hierarchical partition boundary. For this reason, avoid using this alternate type of coding style:

```
reg [7:0] mem [127:0];
reg [6:0] read_address_reg;

always @ (posedge clk) begin
    if (we)
        mem[write_address] <= d;

    read_address_reg <= read_address;
end

assign q = mem[read_address_reg];
```

The VHDL sample in Example 6–14 uses a concurrent signal assignment to read from the RAM. By itself, this example describes new data read-during-write behavior. However, if the RAM output feeds a register in another hierarchy, a read-during-write results in the old data. Synthesis tools may not infer a RAM block if the tool cannot determine which behavior is described, such as when the memory feeds a hard hierarchical partition boundary.

Example 6–14. VHDL Single-Clock Simple Dual-Port Synchronous RAM with New Data Read-During-Write Behavior

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY single_clock_rw_ram IS
    PORT (
        clock: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        write_address: IN INTEGER RANGE 0 to 31;
        read_address: IN INTEGER RANGE 0 to 31;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END single_clock_rw_ram;

ARCHITECTURE rtl OF single_clock_rw_ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL ram_block: MEM;
    SIGNAL read_address_reg: INTEGER RANGE 0 to 31;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
            read_address_reg <= read_address;
        END IF;
    END PROCESS;
    q <= ram_block(read_address_reg);
END rtl;
```

Example 6–14 does not infer a RAM block for the APEX series of devices, ACEX®, or the FLEX® series of devices by default because the read-during-write behavior depends on surrounding logic. For Quartus II integrated synthesis, if you do not require the read-through-write capability, add the synthesis attribute `ramstyle="no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM, rather than use the behavior specified by your HDL code.

Simple Dual-Port, Dual-Clock Synchronous RAM

In dual clock designs, synthesis tools cannot accurately infer the read-during-write behavior because it depends on the timing of the two clocks within the target device. Therefore, the read-during-write behavior of the synthesized design is undefined and may differ from your original HDL code. Refer to "Check Read-During-Write Behavior" on page 6–16 for details.

When Quartus II integrated synthesis infers this type of RAM, it issues a warning because of the undefined read-during-write behavior. If this functionality is acceptable in your design, you can avoid the warning by adding the synthesis attribute `ramstyle="no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM.

The code samples shown in Example 6–15 and Example 6–16 show Verilog HDL and VHDL code that infers dual-clock synchronous RAM. The exact behavior depends on the relationship between the clocks.

Example 6–15. Verilog HDL Simple Dual-Port, Dual-Clock Synchronous RAM

```
module dual_clock_ram(
   output reg [7:0] q,
   input [7:0] d,
   input [6:0] write_address, read_address,
   input we, clk1, clk2
);
   reg [6:0] read_address_reg;
   reg [7:0] mem [127:0];

   always @ (posedge clk1)
   begin
      if (we)
         mem[write_address] <= d;
   end

   always @ (posedge clk2) begin
      q <= mem[read_address_reg];
      read_address_reg <= read_address;
   end
endmodule
```

Example 6–16. VHDL Simple Dual-Port, Dual-Clock Synchronous RAM

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY dual_clock_ram IS
   PORT (
      clock1, clock2: IN STD_LOGIC;
      data: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
      write_address: IN INTEGER RANGE 0 to 31;
      read_address: IN INTEGER RANGE 0 to 31;
      we: IN STD_LOGIC;
      q: OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
   );
END dual_clock_ram;
ARCHITECTURE rtl OF dual_clock_ram IS
   TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL ram_block: MEM;
   SIGNAL read_address_reg : INTEGER RANGE 0 to 31;
BEGIN
   PROCESS (clock1)
   BEGIN
      IF (clock1'event AND clock1 = '1') THEN
         IF (we = '1') THEN
            ram_block(write_address) <= data;
         END IF;
      END IF;
   END PROCESS;
   PROCESS (clock2)
   BEGIN
      IF (clock2'event AND clock2 = '1') THEN
         q <= ram_block(read_address_reg);
         read_address_reg <= read_address;
      END IF;
   END PROCESS;
END rtl;
```

True Dual-Port Synchronous RAM

The code examples in this section show Verilog HDL and VHDL code that infers true dual-port synchronous RAM. Different synthesis tools may differ in their support for these types of memories. This section describes the inference rules for Quartus II integrated synthesis. This type of RAM inference is supported only for the Arria GX, Stratix, and Cyclone series of devices.

Altera TriMatrix memory blocks have two independent address ports, allowing for operations on two unique addresses simultaneously. A read operation and a write operation can share the same port if they share the same address. The Quartus II software infers true dual-port RAMs in Verilog HDL and VHDL with any combination of independent read or write operations in the same clock cycle, with at most two unique port addresses, performing two reads and one write, two writes and one read, or two writes and two reads in one clock cycle with one or two unique addresses.

In the TriMatrix RAM block architecture, there is no priority between the two ports. Therefore, if you write to the same location on both ports at the same time, the result is indeterminate in the device architecture. You must ensure your HDL code does not imply priority for writes to the memory block, if you want the design to be implemented in a dedicated hardware memory block. For example, if both ports are defined in the same process block, the code is synthesized and simulated sequentially so there would be a priority between the two ports. If your code does imply a priority, the logic cannot be implemented in the device RAM blocks and is implemented in regular logic cells.

You must also consider the read-during-write behavior of the RAM block, to ensure that it can be mapped directly to the device RAM architecture. Refer to "Check Read-During-Write Behavior" on page 6–16 for details.

When a read and write operation occur on the same port for the same address, the read operation may behave as follows:

- **Read new data**—This mode matches the behavior of TriMatrix memory blocks.
- **Read old data**—This mode is supported only by Stratix IV, Arria II GX, Stratix III, and Cyclone III TriMatrix memory blocks. This behavior is not possible in TriMatrix memory blocks of other families.

When a read and write operation occur on different ports for the same address (also known as mixed port), the read operation may behave as follows:

- **Read new data**—Quartus II integrated synthesis supports this mode by creating bypass logic around the TriMatrix memory block.
- **Read old data**—This behavior is supported by TriMatrix memory blocks.

The Verilog HDL single-clock code sample shown in Example 6–17 maps directly into Altera TriMatrix memory. When a read and write operation occur on the same port for the same address, the new data being written to the memory is read. When a read and write operation occur on different ports for the same address, the old data in the memory is read. Simultaneous writes to the same location on both ports results in indeterminate behavior.

A dual-clock version of this design describes the same behavior, but the memory in the target device will have undefined mixed port read-during-write behavior because it depends on the relationship between the clocks.

Example 6–17. Verilog HDL True Dual-Port RAM with Single Clock

```
module true_dual_port_ram_single_clock
(
    input [(DATA_WIDTH-1):0] data_a, data_b,
    input [(ADDR_WIDTH-1):0] addr_a, addr_b,
    input we_a, we_b, clk,
    output reg [(DATA_WIDTH-1):0] q_a, q_b
);

    parameter DATA_WIDTH = 8;
    parameter ADDR_WIDTH = 6;

    // Declare the RAM variable
    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

    always @ (posedge clk)
    begin // Port A
        if (we_a)
        begin
            ram[addr_a] <= data_a;
            q_a <= data_a;
        end
        else
            q_a <= ram[addr_a];
    always @ (posedge clk)
    begin // Port b
        if (we_b)
        begin
            ram[addr_b] <= data_b;
            q_b <= data_b;
        end
        else
            q_b <= ram[addr_b];
    end

endmodule
```

If you use the following Verilog HDL read statements instead of the `if-else` statements in Example 6–17, the HDL code specifies that the read results in old data when a read and write operation occur at the same time for the same address on the same port or mixed ports. (This behavior is supported only in the TriMatrix memories of Stratix IV, Arria II GX, Stratix III and Cyclone III devices, and is not inferred as memory for other device families):

```
    always @ (posedge clk)
    begin // Port A
      if (we_a)
        ram[addr_a] <= data_a;

      q_a <= ram[addr_a];
    end

    always @ (posedge clk)
    begin // Port B
      if (we_b)
        ram[addr_b] <= data_b;

      q_b <= ram[addr_b];
    end
```

The VHDL single-clock code sample shown in Example 6–18 maps directly into Altera TriMatrix memory. When a read and write operation occurs on the same port for the same address, the new data being written to the memory is read. When a read and write operation occurs on different ports for the same address, the old data in the memory is read. Because simultaneous writes to the same location on both ports results in indeterminate behavior, Altera recommends that you avoid this condition.

A dual-clock version of this design describes the same behavior, but the memory in the target device will have undefined mixed port read-during-write behavior because it depends on the relationship between the clocks.

Example 6–18. VHDL True Dual-Port RAM with Single Clock (Part 1 of 2)

```
library ieee;
use ieee.std_logic_1164.all;

entity true_dual_port_ram_single_clock is

    generic
    (
        DATA_WIDTH : natural := 8;
        ADDR_WIDTH : natural := 6
    );

    port
    (
        clk : in std_logic;
        addr_a: in natural range 0 to 2**ADDR_WIDTH - 1;
        addr_b: in natural range 0 to 2**ADDR_WIDTH - 1;
        data_a: in std_logic_vector((DATA_WIDTH-1) downto 0);
        data_b: in std_logic_vector((DATA_WIDTH-1) downto 0);
        we_a: in std_logic := '1';
        we_b: in std_logic := '1';
        q_a : out std_logic_vector((DATA_WIDTH -1) downto 0);
        q_b : out std_logic_vector((DATA_WIDTH -1) downto 0)
    );

end true_dual_port_ram_single_clock;

architecture rtl of true_dual_port_ram_single_clock is

    -- Build a 2-D array type for the RAM
    subtype word_t is std_logic_vector((DATA_WIDTH-1) downto 0);
    type memory_t is addr_a(raddr'high downto 0) of word_t;

    -- Declare the RAM signal.
    signal ram : memory_t;
```

Example 6-18. VHDL True Dual-Port RAM with Single Clock (Part 2of 2)

```
begin

    process(clk)
    begin
    if(rising_edge(clk)) then -- Port A
        if(we_a = '1') then
            ram(addr_a) <= data_a;

            -- Read-during-write on the same port returns NEW data
            q_a <= data_a;
        else
            -- Read-during-write on the mixed port returns OLD data
            q_a <= ram(addr_a);
        end if;
end if;

    end process;

    process(clk)
    begin
    if(rising_edge(clk)) then -- Port B
        if(we_b = '1') then
            ram(addr_b) <= data_b;

            -- Read-during-write on the same port returns NEW data
            q_b <= data_b;
        else
            -- Read-during-write on the mixed port returns OLD data
            q_b <= ram(addr_b);
        end if;
    end if;
    end process;

end rtl;
```

Specifying Initial Memory Contents at Power-Up

Your synthesis tool may offer various ways to specify the initial contents of an inferred memory.

Certain device memory types do not support initialized memory, such as the M-RAM blocks in Stratix and Stratix II devices.

There are slight power-up and initialization differences between dedicated RAM blocks and the Stratix III MLAB memory due to the continuous read of the MLAB. Altera dedicated RAM block outputs always power-up to zero and are set to the initial value on the first read. For example, if address `0` is pre-initialized to FF, the RAM block powers up with the output at `0`. A subsequent read after power up from address `0` outputs the pre-initialized value of FF. Therefore, if a RAM is powered up and an enable (read enable or clock enable) is held low, then the power-up output of `0` is maintained until the first valid read cycle. The Stratix III MLAB is implemented using registers that power-up to `0`, but are initialized to their initial value immediately at power-up or reset. Therefore, the initial value regardless of the enable status is seen. Quartus II integrated synthesis does not map inferred memory to MLABs unless the HDL code specifies the appropriate `ramstyle` attribute.

Quartus II integrated synthesis supports the `ram_init_file` synthesis attribute that allows you to specify a Memory Initialization File (**.mif**) for an inferred RAM block.

For information about the `ram_init_file` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about synthesis attributes in other synthesis tools, refer to the tool vendor's documentation.

In Verilog HDL, you can use an initial block to initialize the contents of an inferred memory. Quartus II integrated synthesis automatically converts the initial block into a **.mif** for the inferred RAM. Example 6–19 shows Verilog HDL code that infers a simple dual-port RAM block and corresponding **.mif** file.

Example 6–19. Verilog HDL RAM with Initialized Contents

```
module ram_with_init(
   output reg [7:0] q,
   input [7:0] d,
   input [4:0] write_address, read_address,
   input we, clk
);
   reg [7:0] mem [0:31];
   integer i;

   initial begin
      for (i = 0; i < 32; i = i + 1)
         mem[i] = i[7:0];
   end

   always @ (posedge clk) begin
      if (we)
         mem[write_address] <= d;
      q <= mem[read_address];
   end
endmodule
```

Quartus II integrated synthesis and other synthesis tools also support the `$readmemb` and `$readmemh` commands so that RAM and ROM initialization work identically in synthesis and simulation. Example 6–20 shows an initial block that initializes an inferred RAM block using the `$readmemb` command.

Refer to the Verilog Language Reference Manual (LRM) 1364-2001 Section 17.2.8 for details about the format of the **ram.txt** file.

Example 6–20. Verilog HDL RAM Initialized with the readmemb Command

```
reg [7:0] ram[0:15];
initial
begin
$readmemb("ram.txt", ram);
end
```

In VHDL, you can initialize the contents of an inferred memory by specifying a default value for the corresponding signal. Quartus II integrated synthesis automatically converts the default value into a **.mif** file for the inferred RAM. Example 6–21 shows VHDL code that infers a simple dual-port RAM block and corresponding **.mif** file.

Example 6–21. VHDL RAM with Initialized Contents

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY ram_with_init IS
    PORT(
            clock: IN STD_LOGIC;
            data: IN UNSIGNED (7 DOWNTO 0);
            write_address: IN integer RANGE 0 to 31;
            read_address: IN integer RANGE 0 to 31;
            we: IN std_logic;
            q: OUT UNSIGNED (7 DOWNTO 0));
END;

ARCHITECTURE rtl OF ram_with_init IS

    TYPE MEM IS ARRAY(31 DOWNTO 0) OF unsigned(7 DOWNTO 0);
    FUNCTION initialize_ram
        return MEM is
        variable result : MEM;
    BEGIN
        FOR i IN 31 DOWNTO 0 LOOP
            result(i) := to_unsigned(natural(i), natural'(8));
        END LOOP;
        RETURN result;
    END initialize_ram;

    SIGNAL ram_block : MEM := initialize_ram;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
            ram_block(write_address) <= data;
            END IF;
            q <= ram_block(read_address);
        END IF;
    END PROCESS;
END rtl;
```

ROM Functions—Inferring ALTSYNCRAM and LPM_ROM Megafunctions from HDL Code

To infer ROM functions, synthesis tools detect sets of registers and logic that can be replaced with the ALTSYNCRAM or LPM_ROM megafunctions, depending on the target device family, only for device families that have dedicated memory blocks.

ROMs are inferred when a `CASE` statement exists in which a value is set to a constant for every choice in the `case` statement. Because small ROMs typically achieve the best performance when they are implemented using the registers in regular logic, each ROM function must meet a minimum size requirement to be inferred and placed into memory.

If you use Quartus II integrated synthesis, you can direct the software to infer ROM blocks for all sizes with the **Allow Any ROM Size for Recognition** option under **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box.

Some synthesis tools provide options to control the implementation of inferred ROM blocks for Altera devices with TriMatrix memory blocks. For example, Quartus II integrated synthesis provides the `romstyle` synthesis attribute to specify the type of memory block or to specify the use of regular logic instead of a dedicated memory block.

For details about using the `romstyle` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about synthesis attributes in other synthesis tools, refer to the appropriate chapter in the Synthesis section in volume 1 of the *Quartus II Handbook*.

When you are using a formal verification flow, Altera recommends that you create ROM blocks in separate entities or modules that contain only the ROM logic because you may need to treat the entity and module as a black box during formal verification.

Because formal verification tools do not support ROM megafunctions, Quartus II integrated synthesis does not infer ROM megafunctions when a formal verification tool is selected.

The Verilog HDL and VHDL code samples shown in Example 6–22 through Example 6–25 infer synchronous ROM blocks. Depending on the device family's dedicated RAM architecture, the ROM logic may have to be synchronous; consult the device family handbook for details.

For device architectures with synchronous RAM blocks, such as the Stratix series devices and newer device families, either the address or the output has to be registered for ROM code to be inferred. When output registers are used, the registers are implemented using the input registers of the RAM block, but the functionality of the ROM is not changed. If you register the address, the power-up state of the inferred ROM can be different from the HDL design. In this scenario, the synthesis software issues a warning. The Quartus II Help explains the condition under which the functionality changes when you use Quartus II integrated synthesis.

These ROM code samples map directly to the Altera TriMatrix memory architecture.

Example 6–22. Verilog HDL Synchronous ROM

```
module sync_rom (clock, address, data_out);
    input clock;
    input [7:0] address;
    output [5:0] data_out;

    reg [5:0] data_out;

    always @ (posedge clock)
    begin
        case (address)
            8'b00000000: data_out = 6'b101111;
            8'b00000001: data_out = 6'b110110;
            ...
            8'b11111110: data_out = 6'b000001;
            8'b11111111: data_out = 6'b101010;
        endcase
    end
endmodule
```

Example 6–23. VHDL Synchronous ROM

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY sync_rom IS
    PORT (
        clock: IN STD_LOGIC;
        address: IN STD_LOGIC_VECTOR(7 downto 0);
        data_out: OUT STD_LOGIC_VECTOR(5 downto 0)
    );
END sync_rom;

ARCHITECTURE rtl OF sync_rom IS
BEGIN
PROCESS (clock)
    BEGIN
    IF rising_edge (clock) THEN
        CASE address IS
            WHEN "00000000" => data_out <= "101111";
            WHEN "00000001" => data_out <= "110110";
            ...
            WHEN "11111110" => data_out <= "000001";
            WHEN "11111111" => data_out <= "101010";
            WHEN OTHERS => data_out <= "101111";
        END CASE;
    END IF;
    END PROCESS;
END rtl;
```

Example 6–24. Verilog HDL Dual-Port Synchronous ROM Using readmemb

```
module dual_port_rom (
    input [(addr_width-1):0] addr_a, addr_b,
    input clk,
    output reg [(data_width-1):0] q_a, q_b
);
    parameter data_width = 8;
    parameter addr_width = 8;

    reg [data_width-1:0] rom[2**addr_width-1:0];

    initial  // Read the memory contents in the file
dual_port_rom_init.txt.
    begin
        $readmemb("dual_port_rom_init.txt", rom);
    end

    always @ (posedge clk)
    begin
        q_a <= rom[addr_a];
        q_b <= rom[addr_b];
    end
endmodule
```

Example 6–25. VHDL Dual-Port Synchronous ROM Using Initialization Function

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dual_port_rom is
    generic (
        DATA_WIDTH : natural := 8;
        ADDR_WIDTH : natural := 8
    );
    port (
        clk    : in std_logic;
        addr_a: in natural range 0 to 2**ADDR_WIDTH - 1;
        addr_b: in natural range 0 to 2**ADDR_WIDTH - 1;
        q_a    : out std_logic_vector((DATA_WIDTH -1) downto 0);
        q_b    : out std_logic_vector((DATA_WIDTH -1) downto 0)
    );
end entity;

architecture rtl of dual_port_rom is
    -- Build a 2-D array type for the ROM
    subtype word_t is std_logic_vector((DATA_WIDTH-1) downto 0);
    type memory_t is array(addr_a'high downto 0) of word_t;

    function init_rom
        return memory_t is
        variable tmp : memory_t := (others => (others => '0'));
    begin
        for addr_pos in 0 to 2**ADDR_WIDTH - 1 loop
            -- Initialize each address with the address itself
            tmp(addr_pos) := std_logic_vector(to_unsigned(addr_pos,
DATA_WIDTH));
        end loop;
        return tmp;
    end init_rom;

    -- Declare the ROM signal and specify a default initialization value.
    signal rom : memory_t := init_rom;
begin
    process(clk)
    begin
    if (rising_edge(clk)) then
        q_a <= rom(addr_a);
        q_b <= rom(addr_b);
    end if;
    end process;
end rtl;
```

Shift Registers—Inferring the ALTSHIFT_TAPS Megafunction from HDL Code

To infer shift registers, synthesis tools detect a group of shift registers of the same length and convert them to an ALTSHIFT_TAPS megafunction. To be detected, all the shift registers must have the following characteristics:

- Use the same clock and clock enable
- Do not have any other secondary signals
- Have equally spaced taps that are at least three registers apart

When you use a formal verification flow, Altera recommends that you create shift register blocks in separate entities or modules containing only the shift register logic, because yoight have to treat the entity or module as a black box during formal verification.

Because formal verification tools do not support shift register megafunctions, Quartus II integrated synthesis does not infer the ALTSHIFT_TAPS megafunction when a formal verification tool is selected. You can select EDA tools for use with your Quartus II project on the **EDA Tool Settings** page of the **Settings** dialog box.

For more information about the ALTSHIFT_TAPS megafunction, refer to the *ALTSHIFT_TAPS Megafunction User Guide*.

Synthesis software recognizes shift registers only for device families that have dedicated RAM blocks and the software uses certain guidelines to determine the best implementation. The following guidelines are followed in Quartus II integrated synthesis and also are generally followed by other EDA tools:

- For FLEX 10K® and ACEX 1K devices, the software does not infer ALTSHIFT_TAPS megafunctions because FLEX 10K and ACEX 1K devices have a relatively small amount of dedicated memory.
- For APEX 20K and APEX II devices, the software infers the ALTSHIFT_TAPS megafunction only if the shift register has more than a total of 128 bits. Smaller shift registers typically do not benefit from implementation in dedicated memory.
- For the Arria GX, Stratix, and Cyclone series devices, the software determines whether to infer the ALTSHIFT_TAPS megafunction based on the width of the registered bus (W), the length between each tap (L), and the number of taps (N).
  - If the registered bus width is one ($W = 1$), the software infers ALTSHIFT_TAPS if the number of taps times the length between each tap is greater than or equal to 64 ($N \times L \geq 64$).
  - If the registered bus width is greater than one ($W > 1$), the software infers ALTSHIFT_TAPS if the registered bus width times the number of taps times the length between each tap is greater than or equal to 32 ($W \times N \times L \geq 32$).

If the length between each tap (L) is not a power of two, the software uses more logic to decode the read and write counters. This situation occurs because for different sizes of shift registers, external decode logic that uses logic elements (LEs) or Adaptive Logic Modules (ALMs) is required to implement the function. This decode logic eliminates the performance and utilization advantages of implementing shift registers in memory.

The registers that the software maps to the ALTSHIFT_TAPS megafunction and places in RAM are not available in a Verilog HDL or VHDL output file for simulation tools because their node names do not exist after synthesis.

Simple Shift Register

The code samples shown in Example 6–26 and Example 6–27 show a simple, single-bit wide, 64-bit long shift register. The synthesis software implements the register (`W` = 1 and `M = 64`) in an ALTSHIFT_TAPS megafunction for supported devices. If the length of the register is less than 64 bits, the software implements the shift register in logic.

Example 6–26. Verilog HDL Single-Bit Wide, 64-Bit Long Shift Register

```
module shift_1x64 (clk, shift, sr_in, sr_out);
    input clk, shift;
    input sr_in;
    output sr_out;

    reg [63:0] sr;

    always @ (posedge clk)
    begin
        if (shift == 1'b1)
        begin
            sr[63:1] <= sr[62:0];
            sr[0] <= sr_in;
        end
    end
    assign sr_out = sr[63];
endmodule
```

Example 6–27. VHDL Single-Bit Wide, 64-Bit Long Shift Register

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
ENTITY shift_1x64 IS
PORT (
clk: IN STD_LOGIC;
shift: IN STD_LOGIC;
sr_in: IN STD_LOGIC;
sr_out: OUT STD_LOGIC
);
END shift_1x64;

ARCHITECTURE arch OF shift_1x64 IS
TYPE sr_length IS ARRAY (63 DOWNTO 0) OF STD_LOGIC;
SIGNAL sr: sr_length;
BEGIN
PROCESS (clk)
BEGIN
IF (clk'EVENT and clk = '1') THEN
IF (shift = '1') THEN
sr(63 DOWNTO 1) <= sr(62 DOWNTO 0);
sr(0) <= sr_in;
END IF;
END IF;
END PROCESS;
sr_out <= sr(63);
END arch;
```

Shift Register with Evenly Spaced Taps

The code samples shown in Example 6–28 and Example 6–29 show a Verilog HDL and VHDL 8-bit wide, 64-bit long shift register (`W > 1` and `M = 64`) with evenly spaced taps at 15, 31, and 47. The synthesis software implements this function in a single ALTSHIFT_TAPS megafunction and maps it to RAM in supported devices.

Example 6–28. Verilog HDL 8-Bit Wide, 64-Bit Long Shift Register with Evenly Spaced Taps

```
module shift_8x64_taps (clk, shift, sr_in, sr_out, sr_tap_one,
sr_tap_two, sr_tap_three );
    input clk, shift;
    input [7:0] sr_in;
    output [7:0] sr_tap_one, sr_tap_two, sr_tap_three, sr_out;

    reg [7:0] sr [63:0];
    integer n;

    always @ (posedge clk)
    begin
        if (shift == 1'b1)
        begin
            for (n = 63; n>0; n = n-1)
            begin
                sr[n] <= sr[n-1];
            end
            sr[0] <= sr_in;
        end

    end
    assign sr_tap_one = sr[15];
    assign sr_tap_two = sr[31];
    assign sr_tap_three = sr[47];
    assign sr_out = sr[63];
endmodule
```

Example 6–29. VHDL 8-Bit Wide, 64-Bit Long Shift Register with Evenly Spaced Taps

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
ENTITY shift_8x64_taps IS
    PORT (
        clk: IN STD_LOGIC;
        shift: IN STD_LOGIC;
        sr_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_tap_one: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_tap_two : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_tap_three: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_out: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
    END shift_8x64_taps;

ARCHITECTURE arch OF shift_8x64_taps IS
    SUBTYPE sr_width IS STD_LOGIC_VECTOR(7 DOWNTO 0);
    TYPE sr_length IS ARRAY (63 DOWNTO 0) OF sr_width;
    SIGNAL sr: sr_length;
BEGIN
    PROCESS (clk)
    BEGIN
        IF (clk'EVENT and clk = '1') THEN
            IF (shift = '1') THEN
                sr(63 DOWNTO 1) <= sr(62 DOWNTO 0);
                sr(0) <= sr_in;
            END IF;
        END IF;
    END PROCESS;
    sr_tap_one <= sr(15);
    sr_tap_two <= sr(31);
    sr_tap_three <= sr(47);
    sr_out <= sr(63);
END arch;
```

Coding Guidelines for Registers and Latches

This section provides device-specific coding recommendations for Altera registers and latches. Understanding the architecture of the target Altera device helps ensure that your code produces the expected results and achieves the optimal quality of results.

This section provides guidelines in the following areas:

- "Register Power-Up Values in Altera Devices"
- "Secondary Register Control Signals Such as Clear and Clock Enable" on page 6–38
- "Latches" on page 6–42

Register Power-Up Values in Altera Devices

Registers in the device core always power up to a low (0) logic level on all Altera devices. However, there are ways to implement logic such that registers behave as if they were powering up to a high (1) logic level.

If you use a preset signal on a device that does not support presets in the register architecture, then your synthesis tool may convert the preset signal to a clear signal, which requires synthesis to perform an optimization referred to as NOT gate push-back. NOT gate push-back adds an inverter to the input and the output of the register so that the reset and power-up conditions will appear to be high but the device operates as expected. In this case, your synthesis tool may issue a message informing you about the power-up condition. The register itself powers up low, but the register output is inverted, so the signal that arrives at all destinations is high.

Due to these effects, if you specify a non-zero reset value, you may cause your synthesis tool to use the asynchronous clear (`aclr`) signals available on the registers to implement the high bits with NOT gate push-back. In that case, the registers look as though they power up to the specified reset value. You see this behavior, for example, if your design targets FLEX 10KE or ACEX devices.

When a load signal is available in the device, your synthesis tools can implement a reset of 1 or 0 value by using an asynchronous load of 1 or 0. When the synthesis tool uses an asynchronous load signal, it is not performing NOT gate push-back, so the registers power up to a `0` logic level.

For additional details, refer to the appropriate device family handbook or the appropriate handbook of the Altera website.

Designers typically use an explicit reset signal for the design, which forces all registers into their appropriate values after reset but not necessarily at power-up. You can create your design such that the asynchronous reset allows the board to operate in a safe condition and then you can bring up the design with the reset active. This is a good practice so you do not depend on the power-up conditions of the device.

You can make the your design more stable and avoid potential glitches by synchronizing external or combinational logic of the device architecture before you drive the asynchronous control ports of registers.

For additional information about good synchronous design practices, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

If you want to force a particular power-up condition for your design, use the synthesis options available in your synthesis tool. With Quartus II integrated synthesis, you can apply the **Power-Up Level** logic option. You can also apply the option with an `altera_attribute` assignment in your source code. Using this option forces synthesis to perform NOT gate push-back because synthesis tools cannot actually change the power-up states of core registers.

You can apply the Quartus II integrated synthesis **Power-Up Level** logic option to a specific register or to a design entity, module or subdesign. If you do so, every register in that block receives the value. Registers power up to 0 by default; therefore you can use this assignment to force all registers to power up to 1 using NOT gate push-back.

Be aware that using NOT gate push-back as a global assignment could slightly degrade the quality of results due to the number of inverters that are required. In some situations, issues are caused by enable or secondary control logic inference. It may also be more difficult to migrate such a design to an ASIC or a HardCopy® device. You can simulate the power-up behavior in a functional simulation if you use initialization.

The **Power-Up Level** option and the `altera_attribute` assignment are described in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Some synthesis tools can also read the default or initial values for registered signals and implement this behavior in the device. For example, Quartus II integrated synthesis converts default values for registered signals into **Power-Up Level** settings. That way, the synthesized behavior matches the power-up state of the HDL code during a functional simulation.

For example, the code samples in Example 6–30 and Example 6–31 both infer a register for `q` and set its power-up level to high (while the reset value is `0`).

Example 6–30. Verilog Register with Reset and High Power-Up Value

```
reg q = 1'b1;

always @ (posedge clk or posedge aclr)
begin
if (aclr)
q <= 1'b0;
else
q <= d;
end
```

Example 6–31. VHDL Register with Reset and High Power-Up Level

```
SIGNAL q : STD_LOGIC := '1'; -- q has a default value of '1'

PROCESS (clk, reset)
BEGIN
IF (reset = '1') THEN
q <= '0';
ELSIF (rising_edge(clk)) THEN
q <= d;
END IF;
END PROCESS;
```

Secondary Register Control Signals Such as Clear and Clock Enable

FPGA device architectures contain registers, also known as “flipflops”. The registers in Altera FPGAs provide a number of secondary control signals (such as clear and enable signals) that you can use to implement control logic for each register without using extra logic cells. Device families vary in their support for secondary signals, so consult the device family data sheet to verify which signals are available in your target device.

To make the most efficient use of the signals in the device, your HDL code should match the device architecture as closely as possible. The control signals have a certain priority due to the nature of the architecture, so your HDL code should follow that priority where possible.

Your synthesis tool can emulate any control signals using regular logic, so getting functionally correct results is always possible. However, if your design requirements are flexible in terms of which control signals are used and in what priority, match your design to the target device architecture to achieve the most efficient results. If the priority of the signals in your design is not the same as that of the target architecture, then extra logic may be required to implement the control signals. This extra logic uses additional device resources and can cause additional delays for the control signals.

In addition, there are certain cases where using logic other than the dedicated control logic in the device architecture can have a larger impact. For example, the clock enable signal has priority over the synchronous reset or clear signal in the device architecture. The clock enable turns off the clock line in the LAB, and the clear signal is synchronous. So in the device architecture, the synchronous clear takes effect only when a clock edge occurs.

If you code a register with a synchronous clear signal that has priority over the clock enable signal, the software must emulate the clock enable functionality using data inputs to the registers. Because the signal does not use the clock enable port of a register, you cannot apply a Clock Enable Multicycle constraint. In this case, following the priority of signals available in the device is clearly the best choice for the priority of these control signals, and using a different priority causes unexpected results with an assignment to the clock enable signal.

The priority order for secondary control signals in Altera devices differs from the order for other vendors' devices. If your design requirements are flexible regarding priority, verify that the secondary control signals meet design performance requirements when migrating designs between FPGA vendors and try to match your target device architecture to achieve the best results.

The signal order is the same for all Altera device families, although as noted previously, not all device families provide every signal. The following priority order is observed:

1. Asynchronous Clear, `aclr`—highest priority
2. Preset, `pre`
3. Asynchronous Load, `aload`
4. Enable, `ena`
5. Synchronous Clear, `sclr`
6. Synchronous Load, `sload`
7. Data In, `data`—lowest priority

The following examples provide Verilog HDL and VHDL code that creates a register with the `aclr`, `aload`, and `ena` control signals.

The Verilog HDL example (Example 6–32) does not have adata on the sensitivity list, but the VHDL example (Example 6–33) does. This is a limitation of the Verilog HDL language—there is no way to describe an asynchronous load signal (in which q toggles if adata toggles while aload is high). All synthesis tools should infer an aload signal from this construct despite this limitation. When they perform such inference, you may see information or warning messages from the synthesis tool.

Example 6–32. Verilog HDL D-Type Flipflop (Register) with ena, aclr, and aload Control Signals

```
module dff_control(clk, aclr, aload, ena, data, adata, q);
    input clk, aclr, aload, ena, data, adata;
    output q;

    reg q;

    always @ (posedge clk or posedge aclr or posedge aload)
    begin
        if (aclr)
            q <= 1'b0;
        else if (aload)
            q <= adata;
        else if (ena)
            q <= data;
    end
endmodule
```

Example 6–33. VHDL D-Type Flipflop (Register) with ena, aclr, and aload Control Signals

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dff_control IS
   PORT (
clk: IN STD_LOGIC;
aclr: IN STD_LOGIC;
aload: IN STD_LOGIC;
adata: IN STD_LOGIC;
ena: IN STD_LOGIC;
      data: IN STD_LOGIC;
q: OUT STD_LOGIC
    );
END dff_control;

ARCHITECTURE rtl OF dff_control IS
BEGIN
   PROCESS (clk, aclr, aload, adata)
   BEGIN
IF (aclr = '1') THEN
q <= '0';
ELSIF (aload = '1') THEN
q <= adata;
ELSE
           IF (clk = '1' AND clk'event) THEN
               IF (ena ='1') THEN
q <= data;
               END IF;
           END IF;
       END IF;
   END PROCESS;
END rtl;
```

The preset signal is not available in many device families; therefore, it is not included in the examples.

Creating many registers with different `sload` and `sclr` signals can make packing the registers into LABs difficult for the Quartus II Fitter because the `sclr` and `sload` signals are LAB-wide signals. In addition, using the LAB-wide `sload` signal prevents the Fitter from packing registers using the quick feedback path in the device architecture, which means that some registers cannot be packed with other logic.

Synthesis tools typically restrict use of `sload` and `sclr` signals to cases in which there are enough registers with common signals to allow good LAB packing. Using the look-up table (LUT) to implement the signals is always more flexible if it is available. Because different device families offer different numbers of control signals, inference of these signals is also device-specific. For example, Stratix II devices have more flexibility than Stratix devices with respect to secondary control signals, so synthesis tools might infer more `sload` and `sclr` signals for Stratix II devices.

If you use these additional control signals, use them in the priority order that matches the device architecture. To achieve the most efficient results, ensure the `sclr` signal has a higher priority than the `sload` signal in the same way that `aclr` has higher priority than `aload` in the previous examples. Remember that the register signals are not inferred unless the design meets the conditions described previously. However, if your HDL described the desired behavior, the software always implements logic with the correct functionality.

In Verilog HDL, the following code for `sload` and `sclr` could replace the `if (ena) q <= data;` statements in the Verilog HDL example shown in Example 6–32 (after adding the control signals to the module declaration).

Example 6–34. Verilog HDL sload and sclr Control Signals

```
if  (ena) begin
   if  (sclr)
     q <= 1'b0;
   else if  (sload)
     q <= sdata;
   else
     q <= data;
end
```

In VHDL, the following code for `sload` and `sclr` could replace the `IF (ena ='1') THEN q <= data; END IF;` statements in the VHDL example shown in Example 6–33 on page 6–40 (after adding the control signals to the entity declaration).

Example 6–35. VHDL sload and sclr Control Signals

```
IF (ena ='1') THEN
   IF (sclr = '1') THEN
     q <= '0';
   ELSIF (sload = '1') THEN
     q <= sdata;
   ELSE
     q <= data;
   END IF;
END IF;
```

Latches

A latch is a small combinational loop that holds the value of a signal until a new value is assigned.

Altera recommends that you design without the use of latches whenever possible.

For additional information about the issues involved in designing with latches and combinational loops, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

Latches can be inferred from HDL code when you did not intend to use a latch, as described in "Unintentional Latch Generation". If you do intend to infer a latch, it is important to infer it correctly to guarantee correct device operation as detailed in "Inferring Latches Correctly" on page 6–43.

Unintentional Latch Generation

When you are designing combinational logic, certain coding styles can create an unintentional latch. For example, when `CASE` or `IF` statements do not cover all possible input conditions, latches may be required to hold the output if a new output value is not assigned. Check your synthesis tool messages for references to inferred latches. If your code unintentionally creates a latch, make code changes to remove the latch.

Latches have limited support in formal verification tools. Therefore, ensure that you do not infer latches unintentionally. For example, an incomplete `CASE` statement may create a latch when you are using formal verification in your design flow.

The `full_case` attribute can be used in Verilog HDL designs to treat unspecified cases as don't care values (`X`). However, using the `full_case` attribute can cause simulation mismatches because this attribute is a synthesis-only attribute, so simulation tools still treat the unspecified cases as latches.

Refer to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook* for more information about using attributes in your synthesis tool. The *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook* provides an example explaining possible simulation mismatches.

Omitting the final `else` or `when others` clause in an `if` or `case` statement can also generate a latch. Don't care (`X`) assignments on the default conditions are useful in preventing latch generation. For the best logic optimization, assign the default `case` or final `else` value to don't care (`X`) instead of a logic value.

The VHDL sample code shown in Example 6–36 prevents unintentional latches. Without the final `else` clause, this code creates unintentional latches to cover the remaining combinations of the `sel` inputs. When you are targeting a Stratix device with this code, omitting the final `else` condition can cause the synthesis software to use up to six LEs, instead of the three it uses with the `else` statement. Additionally, assigning the final `else` clause to `1` instead of `X` can result in slightly more LEs because the synthesis software cannot perform as much optimization when you specify a constant value compared to a don't care value.

Example 6–36. VHDL Code Preventing Unintentional Latch Creation

```
LIBRARY ieee;
USE IEEE.std_logic_1164.all;

ENTITY nolatch IS
    PORT (a,b,c: IN STD_LOGIC;
        sel: IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        oput: OUT STD_LOGIC);
END nolatch;

ARCHITECTURE rtl OF nolatch IS
BEGIN
    PROCESS (a,b,c,sel) BEGIN
        if sel = "00000" THEN
            oput <= a;
        ELSIF sel = "00001" THEN
            oput <= b;
        ELSIF sel = "00010" THEN
            oput <= c;
        ELSE            --- Prevents latch inference
            oput <= ''X'; --/
        END if;
    END PROCESS;
END rtl;
```

Inferring Latches Correctly

Synthesis tools can infer a latch that does not exhibit the glitch and timing hazard problems typically associated with combinational loops.

Any use of latches generates warnings and is flagged if the design is migrated to a HardCopy ASIC. In addition, timing analysis does not completely model latch timing in some cases. Do not use latches unless you are very certain that your design requires it, and you fully understand the impact of using the latches.

When using Quartus II integrated synthesis, latches that are inferred by the software are reported in the **User-Specified and Inferred Latches** section of the Compilation Report. This report indicates whether the latch is considered safe and free of timing hazards.

If a latch or combinational loop in your design is not listed in the **User-Specified and Inferred Latches** section, it means that it was not inferred as a safe latch by the software and is not considered glitch-free.

All combinational loops listed in the **Analysis & Synthesis Logic Cells Representing Combinational Loops** table in the Compilation Report are at risk of timing hazards. These entries indicate possible problems with your design that you should investigate. However, it is possible to have a correct design that includes combinational loops. For example, it is possible that the combinational loop cannot be sensitized. This can occur in cases where there is an electrical path in the hardware, but either the designer knows that the circuit never encounters data that causes that path to be activated, or the surrounding logic is set up in a mutually exclusive manner that prevents that path from ever being sensitized, independent of the data input.

For macrocell-based devices such as MAX® 7000AE and MAX 3000A, all data (D-type) latches and set-reset (S-R) latches listed in the **Analysis & Synthesis User-Specified and Inferred Latches** table have an implementation free of timing hazards such as glitches. The implementation includes a cover term to ensure there is no glitching, and includes a single macrocell in the feedback loop.

For 4-input LUT-based devices such as Stratix devices, the Cyclone series, and MAX II devices, all latches in the **User-Specified and Inferred Latches** table with a single LUT in the feedback loop are free of timing hazards when a single input changes. Because of the hardware behavior of the LUT, the output does not glitch when a single input toggles between two values that are supposed to produce the same output value. For example, a D-type input toggling when the enable input is inactive, or a set input toggling when a reset input with higher priority is active. This hardware behavior of the LUT means that no cover term is required for a loop around a single LUT. The Quartus II software uses a single LUT in the feedback loop whenever possible. A latch that has data, enable, set, and reset inputs in addition to the output fed back to the input cannot be implemented in a single 4-input LUT. If the Quartus II software cannot implement the latch with a single-LUT loop because there are too many inputs, the **User-Specified and Inferred Latches** table indicates that the latch is not free of timing hazards.

For 6-input LUT-based devices, the software can implement all latch inputs with a single adaptive look-up table (ALUT) in the combinational loop. Therefore, all latches in the **User-Specified and Inferred Latches** table are free of timing hazards when a single input changes.

If a latch is listed as a safe latch, other Quartus II optimizations, such as physical synthesis netlist optimizations in the Fitter, maintain the hazard-free performance.

To ensure hazard-free behavior, only one control input may change at a time. Changing two inputs simultaneously, such as deasserting set and reset at the same time, or changing data and enable at the same time, can produce incorrect behavior in any latch.

Quartus II integrated synthesis infers latches from always blocks in Verilog HDL and process statements in VHDL, but not from continuous assignments in Verilog HDL or concurrent signal assignments in VHDL. These rules are the same as for register inference. The software infers registers or flipflops only from `always` blocks and `process` statements.

The Verilog HDL code sample shown in Example 6–37 infers a S-R latch correctly in the Quartus II software.

Example 6–37. Verilog HDL Set-Reset Latch

```
module simple_latch (
   input SetTerm,
   input ResetTerm,
   output reg LatchOut
   );

   always @ (SetTerm or ResetTerm) begin
      if (SetTerm)
         LatchOut = 1'b1
      else if (ResetTerm)
         LatchOut = 1'b0
   end
endmodule
```

The VHDL code sample shown in Example 6–38 infers a D-type latch correctly in the Quartus II software.

Example 6–38. VHDL Data Type Latch

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY simple_latch IS
PORT (
enable, data     : IN STD_LOGIC;
q                : OUT STD_LOGIC
);
END simple_latch;

ARCHITECTURE rtl OF simple_latch IS
BEGIN

latch : PROCESS (enable, data)
BEGIN
IF (enable = '1') THEN
q <= data;
END IF;
END PROCESS latch;
END rtl;
```

The following example shows a Verilog HDL continuous assignment that does not infer a latch in the Quartus II software. (The behavior is similar to a latch, but it may not function correctly as a latch and its timing is not analyzed as a latch):

```
assign latch_out = (~en & latch_out) | (en & data);
```

Quartus II integrated synthesis also creates safe latches when possible for instantiations of the LPM_LATCH megafunction. You can use this megafunction to create a latch with any combination of data, enable, set, and reset inputs. The same limitations apply for creating safe latches as for inferring latches from HDL code.

Inferring the Altera LPM_LATCH function in another synthesis tool ensures that the implementation is also recognized as a latch in the Quartus II software. If a third-party synthesis tool implements a latch using the LPM_LATCH megafunction, then the Quartus II integrated synthesis lists the latch in the **User-Specified and Inferred Latches** table in the same way as it lists latches created in HDL source code. The coding style necessary to produce an LPM_LATCH implementation may depend on your synthesis tool. Some third-party synthesis tools list the number of LPM_LATCH functions that are inferred.

For LUT-based families, the Fitter uses global routing for control signals including signals that Analysis and Synthesis identifies as latch enables. In some cases the global insertion delay may decrease the timing performance. If necessary, you can turn off the **Quartus II Global Signal** logic option to manually prevent the use of global signals. Global latch enables are listed in the **Global & Other Fast Signals** table in the Compilation Report.

General Coding Guidelines

This section helps you understand how synthesis tools map various types of HDL code into the target Altera device. Following Altera recommended coding styles, and in some cases designing logic structures to match the appropriate device architecture, can provide significant improvements in the design's quality of results.

This section provides coding guidelines for the following logic structures:

- "Tri-State Signals". This section explains how to create tri-state signals for bidirectional I/O pins.
- "Clock Multiplexing" on page 6–47. This section provides recommendations for multiplexing clock signals.
- "Adder Trees" on page 6–51. This section explains the different coding styles that lead to optimal results for devices with 4-input look-up tables and 6-input ALUTs.
- "State Machines" on page 6–53. This section helps ensure the best results when you use state machines.
- "Multiplexers" on page 6–60. This section explains how multiplexers can be synthesized for 4-input LUT devices, addresses common problems, and provides guidelines to achieve optimal resource utilization.
- "Cyclic Redundancy Check Functions" on page 6–68. This section provides guidelines for getting good results when designing Cyclic Redundancy Check (CRC) functions.
- "Comparators" on page 6–69. This section explains different comparator implementations and provides suggestions for controlling the implementation.
- "Counters" on page 6–71. This section provides guidelines to ensure your counter design targets the device architecture optimally.

Tri-State Signals

When you target Altera devices, you should use tri-state signals only when they are attached to top-level bidirectional or output pins. Avoid lower level bidirectional pins, and avoid using the `Z` logic value unless it is driving an output or bidirectional pin.

Synthesis tools implement designs with internal tri-state signals correctly in Altera devices using multiplexer logic, but Altera does not recommend this coding practice.

In hierarchical block-based or incremental design flows, a hierarchical boundary cannot contain any bidirectional ports, unless the lower level bidirectional port is connected directly through the hierarchy to a top-level output pin without connecting to any other design logic. If you use boundary tri-states in a lower level block, synthesis software must push the tri-states through the hierarchy to the top-level to make use of the tri-state drivers on output pins of Altera devices. Because pushing tri-states requires optimizing through hierarchies, lower level tri-states are restricted with block-based design methodologies.

The code examples shown in Example 6–39 and Example 6–40 show Verilog HDL and VHDL code that creates tri-state bidirectional signals.

Example 6–39. Verilog HDL Tri-State Signal

```
module tristate (myinput, myenable, mybidir);
   input myinput, myenable;
   inout mybidir;
   assign mybidir = (myenable ? myinput : 1'bZ);
endmodule
```

Example 6–40. VHDL Tri-State Signal

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY tristate IS
PORT (
   mybidir : INOUT STD_LOGIC;
   myinput : IN STD_LOGIC;
   myenable : IN STD_LOGIC
    );
END tristate;

ARCHITECTURE rtl OF tristate IS
BEGIN
   mybidir <= 'Z' WHEN (myenable = '0') ELSE myinput;
END rtl;
```

Clock Multiplexing

Clock multiplexing is sometimes used to operate the same logic function with different clock sources. This type of logic can introduce glitches that create functional problems, and the delay inherent in the combinational logic can lead to timing problems. Clock multiplexers trigger warnings from a wide range of design rule check and timing analysis tools.

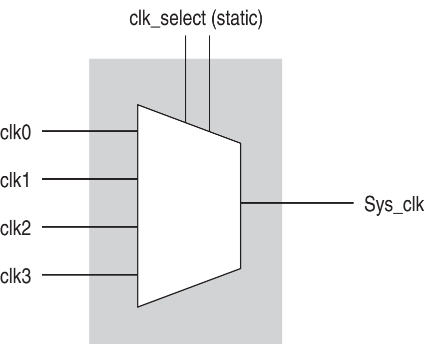
Altera recommends using dedicated hardware to perform clock multiplexing when it is available, instead of using multiplexing logic. For example, you can use the Clock Switchover feature or the Clock Control Block available in certain Altera devices. These dedicated hardware blocks avoid glitches, ensure that you use global low-skew routing lines, and avoid any possible hold time problems on the device due to logic delay on the clock line. Many Altera devices also support dynamic PLL reconfiguration, which is the safest and most robust method of changing clock rates during device operation.

Refer to the appropriate device data sheet or handbook for device-specific information about clocking structures. Also refer to the *ALTCLKCTRL Megafunction User Guide*, the *ALTPLL Megafunction User Guide*, and the *Phase-Locked Loops Reconfiguration (ALTPLL_RECONFIG) Megafunction User Guide*.

If you implement a clock multiplexer in logic cells because the design has too many clocks to use the clock control block, or if dynamic reconfiguration is too complex for your design, it is important to consider simultaneous toggling inputs and ensure glitch-free transitions.

Figure 6–2 shows a simple representation of a clock multiplexer (mux) in a device with 6-input LUTs.

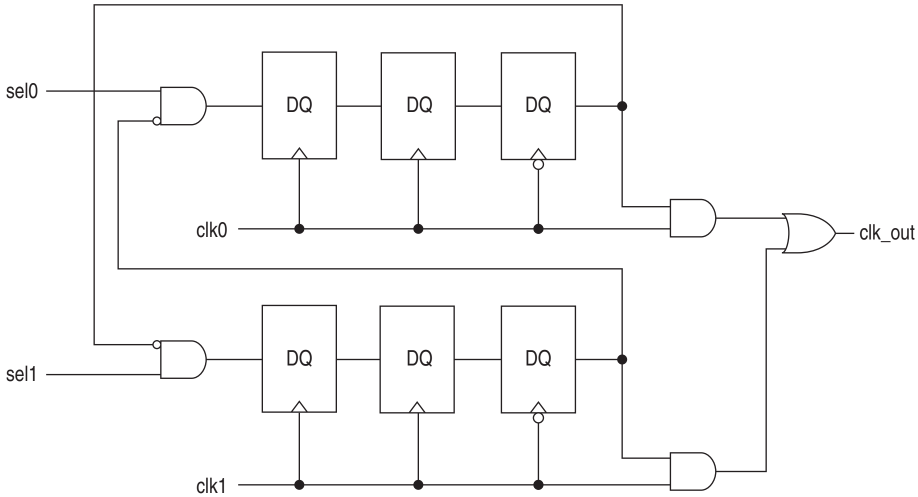
Figure 6–2. Simple Clock Multiplexer in a 6-Input LUT



The data sheet for your target device describes how LUT outputs may glitch during a simultaneous toggle of input signals, independent of the LUT function. Although in practice the 4:1 MUX function does not generate detectable glitches during simultaneous data input toggles, it is possible to construct cell implementations that do exhibit significant glitches, so this simple clock mux structure is not recommended. An additional problem with this implementation is that the output behaves erratically during a change in the `clk_select` signals. This behavior could create timing violations on all registers fed by the system clock and result in possible metastability.

A more sophisticated clock select structure can eliminate the simultaneous toggle and switching problems, as shown in Figure 6–3.

Figure 6–3. Glitch-Free Clock Multiplexer Structure



This structure can be generalized for any number of clock channels. Example 6–41 contains a parameterized version in Verilog HDL. The design enforces that no clock activates until all others have been inactive for at least a few cycles, and that activation occurs while the clock is low. The design applies a `synthesis_keep` directive to the AND gates on the right side of the figure, which ensures there are no simultaneous toggles on the input of the `clk_out` OR gate.

It is important to note that switching from clock `A` to clock `B` requires that clock `A` continue to operate for at least a few cycles. If the old clock stops immediately, the design sticks. The select signals are implemented as a “one-hot” control in this example, but you can use other encoding if you prefer. The input side logic is asynchronous and is not critical. This design can tolerate extreme glitching during the switch process.

Example 6–41. Verilog HDL Clock Multiplexing Design to Avoid Glitches

```
module clock_mux (clk,clk_select,clk_out);

parameter num_clocks = 4;

input [num_clocks-1:0] clk;
input [num_clocks-1:0] clk_select; // one hot
output clk_out;

genvar i;

reg [num_clocks-1:0] ena_r0;
reg [num_clocks-1:0] ena_r1;
reg [num_clocks-1:0] ena_r2;
wire [num_clocks-1:0] qualified_sel;

// A look-up-table (LUT) can glitch when multiple inputs
// change simultaneously. Use the keep attribute to
// insert a hard logic cell buffer and prevent
// the unrelated clocks from appearing on the same LUT.

wire [num_clocks-1:0] gated_clks /* synthesis keep */;

initial begin
ena_r0 = 0;
ena_r1 = 0;
ena_r2 = 0;
end

generate
for (i=0; i<num_clocks; i=i+1)
begin : lp0
wire [num_clocks-1:0] tmp_mask;
assign tmp_mask = {num_clocks{1'b1}} ^ (1 << i);

assign qualified_sel[i] = clk_select[i] &
          (~|(ena_r2 & tmp_mask));

always @(posedge clk[i]) begin
ena_r0[i] <= qualified_sel[i];
ena_r1[i] <= ena_r0[i];
end

always @(negedge clk[i]) begin
ena_r2[i] <= ena_r1[i];
end

assign gated_clks[i] = clk[i] & ena_r2[i];
end
endgenerate

// These will not exhibit simultaneous toggle by construction
assign clk_out = |gated_clks;

endmodule
```

Adder Trees

Structuring adder trees appropriately to match your targeted Altera device architecture can result in significant performance and density improvements. A good example of an application using a large adder tree is a finite impulse response (FIR) correlator. Using a pipelined binary or ternary adder tree appropriately can greatly improve the quality of your results.

This section explains why coding recommendations are different for Altera 4-input LUT devices and 6-input LUT devices.

Architectures with 4-Input LUTs in Logic Elements

Architectures such as Stratix devices and the Cyclone series, APEX series, and FLEX series of devices contain 4-input LUTs as the standard combinational structure in the LE.

If your design can tolerate pipelining, the fastest way to add three numbers `A`, `B`, and `C` in devices that use 4-input lookup tables is to add `A + B`, register the output, and then add the registered output to `C`. Adding `A + B` takes one level of logic (one bit is added in one LE), so this runs at full clock speed. This can be extended to as many numbers as desired.

Example 6–42 shows five numbers `A`, `B`, `C`, `D`, and `E` are added. Adding five numbers in devices that use 4-input lookup tables requires four adders and three levels of registers for a total of 64 LEs (for 16-bit numbers).

Example 6–42. Verilog HDL Pipelined Binary Tree

```
module binary_adder_tree (a, b, c, d, e, clk, out);
   parameter width = 16;
   input [width-1:0] a, b, c, d, e;
   input clk;
   output [width-1:0] out;

   wire [width-1:0] sum1, sum2, sum3, sum4;
   reg [width-1:0] sumreg1, sumreg2, sumreg3, sumreg4;
   // Registers

   always @ (posedge CLK)
      begin
         sumreg1 <= sum1;
         sumreg2 <= sum2;
         sumreg3 <= sum3;
         sumreg4 <= sum4;
      end

   // 2-bit additions
   assign sum1 = A + B;
   assign sum2 = C + D;
   assign sum3 = sumreg1 + sumreg2;
   assign sum4 = sumreg3 + E;
   assign out = sumreg4;
endmodule
```

Architectures with 6-Input LUTs in Adaptive Logic Modules

High-performance Altera device families use a 6-input LUT in their basic logic structure, so these devices benefit from a different coding style from the previous example presented for 4-input LUTs. Specifically, in these devices, ALMs can simultaneously add three bits. Therefore, the tree in Example 6–42 must be two levels deep and contain just two add-by-three inputs instead of four add-by-two inputs.

Although the code in the previous example compiles successfully for 6-input LUT devices, the code is inefficient and does not take advantage of the 6-input adaptive ALUT. By restructuring the tree as a ternary tree, the design becomes much more efficient, significantly improving density utilization. Therefore, when you are targeting with ALUTs and ALMs, large pipelined binary adder trees designed for 4-input LUT architectures should be rewritten to take advantage of the advanced device architecture.

Example 6–43 uses just 32 ALUTs in a Stratix II device—more than a 4:1 advantage over the number of LUTs in the prior example implemented in a Stratix device.

You cannot pack a LAB full when using this type of coding style because of the number of LAB inputs. However, in a typical design, the Quartus II Fitter can pack other logic into each LAB to take advantage of the unused ALMs.

Example 6–43. Verilog HDL Pipelined Ternary Tree

```
module ternary_adder_tree (a, b, c, d, e, clk, out);
   parameter width = 16;
   input [width-1:0] a, b, c, d, e;
   input clk;
   output [width-1:0] out;

   wire [width-1:0] sum1, sum2;
   reg [width-1:0] sumreg1, sumreg2;
   // registers

   always @ (posedge clk)
      begin
         sumreg1 <= sum1;
         sumreg2 <= sum2;
      end

   // 3-bit additions
   assign sum1 = a + b + c;
   assign sum2 = sumreg1 + d + e;
   assign out = sumreg2;
endmodule
```

These examples show pipelined adders, but partitioning your addition operations can help you achieve better results in nonpipelined adders as well. If your design is not pipelined, a ternary tree provides much better performance than a binary tree. For example, depending on your synthesis tool, the HDL code `sum = (A + B + C) + (D + E)` is more likely to create the optimal implementation of a 3-input adder for `A + B + C` followed by a 3-input adder for `sum1 + D + E` than the code without the parentheses. If you do not add the parentheses, the synthesis tool may partition the addition in a way that is not optimal for the architecture.

State Machines

Synthesis tools can recognize and encode Verilog HDL and VHDL state machines during synthesis. This section presents guidelines to ensure the best results when you use state machines. Ensuring that your synthesis tool recognizes a piece of code as a state machine allows the tool to recode the state variables to improve the quality of results, and allows the tool to use the known properties of state machines to optimize other parts of the design. When synthesis recognizes a state machine, it is often able to improve the design area and performance.

To achieve the best results on average, synthesis tools often use one-hot encoding for FPGA devices and minimal-bit encoding for CPLD devices, although the choice of implementation can vary for different state machines and different devices. Refer to your synthesis tool documentation for specific ways to control the manner in which state machines are encoded.

For information about state machine encoding in Quartus II integrated synthesis, refer to the *State Machine Processing* section in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

To ensure proper recognition and inference of state machines and to improve the quality of results, Altera recommends that you observe the following guidelines, which apply to both Verilog HDL and VHDL:

- Assign default values to outputs derived from the state machine so that synthesis does not generate unwanted latches.
- Separate the state machine logic from all arithmetic functions and data paths, including assigning output values.
- If your design contains an operation that is used by more than one state, define the operation outside the state machine and cause the output logic of the state machine to use this value.
- Use a simple asynchronous or synchronous reset to ensure a defined power-up state. If your state machine design contains more elaborate reset logic, such as both an asynchronous reset and an asynchronous load, the Quartus II software generates regular logic rather than inferring a state machine.

If a state machine enters an illegal state due to a problem with the device, the design likely ceases to function correctly until the next reset of the state machine. Synthesis tools do not provide for this situation by default. The same issue applies to any other registers if there is some kind of fault in the system. A `default` or `when others` clause does not affect this operation, assuming that your design never deliberately enters this state. Synthesis tools remove any logic generated by a default state if it is not reachable by normal state machine operation.

Many synthesis tools (including Quartus II integrated synthesis) have an option to implement a safe state machine. The software inserts extra logic to detect an illegal state and force the state machine's transition to the reset state. It is commonly used when the state machine can enter an illegal state. The most common cause of this situation is a state machine that has control inputs that come from another clock domain, such as the control logic for a dual-clock FIFO.

This option protects only state machines. All other registers in the design are not protected this way.

For additional information about tool-specific options for implementing state machines, refer to the tool vendor's documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

The following two sections, "Verilog HDL State Machines" and "VHDL State Machines" on page 6–58, describe additional language-specific guidelines and coding examples.

Verilog HDL State Machines

To ensure proper recognition and inference of Verilog HDL state machines, observe the following additional Verilog HDL guidelines. Some of these guidelines may be specific to Quartus II integrated synthesis. Refer to your synthesis tool documentation for specific coding recommendations.

If the state machine is not recognized and inferred by the synthesis software (such as Quartus II integrated synthesis), the state machine is implemented as regular logic gates and registers and the state machine is not listed as a state machine in the **Analysis & Synthesis** section of the Quartus II Compilation Report. In this case, the software does not perform any of the optimizations that are specific to state machines.

- If you are using the SystemVerilog standard, use enumerated types to describe state machines (as shown in the "SystemVerilog State Machine Coding Example" on page 6–57).
- Represent the states in a state machine with the parameter data types in Verilog-1995 and -2001 and use the parameters to make state assignments (as shown below in the "Verilog-2001 State Machine Coding Example"). This implementation makes the state machine easier to read and reduces the risk of errors during coding.

  Altera recommends against the direct use of integer values for state variables such as `next_state <= 0`. However, using an integer does not prevent inference in the Quartus II software.

- No state machine is inferred in the Quartus II software if the state transition logic uses arithmetic similar to that shown in the following example:

```
case (state)
    0: begin
        if (ena) next_state <= state + 2;
        else next_state <= state + 1;
    end
    1: begin
    ...
endcase
```

- No state machine is inferred in the Quartus II software if the state variable is an output.
- No state machine is inferred in the Quartus II software for signed variables.

Verilog-2001 State Machine Coding Example

The following module `verilog_fsm` is an example of a typical Verilog HDL state machine implementation (Example 6–44).

This state machine has five states. The asynchronous reset sets the variable `state` to `state_0`. The sum of `in_1` and `in_2` is an output of the state machine in `state_1` and `state_2`. The difference (`in_1` – `in_2`) is also used in `state_1` and `state_2`. The temporary variables `tmp_out_0` and `tmp_out_1` store the sum and the difference of `in_1` and `in_2`. Using these temporary variables in the various states of the state machine ensures proper resource sharing between the mutually exclusive states.

Example 6–44. Verilog-2001 State Machine

```
module verilog_fsm (clk, reset, in_1, in_2, out);
   input clk, reset;
   input [3:0] in_1, in_2;
   output [4:0] out;
   parameter state_0 = 3'b000;
   parameter state_1 = 3'b001;
   parameter state_2 = 3'b010;
   parameter state_3 = 3'b011;
   parameter state_4 = 3'b100;

   reg [4:0] tmp_out_0, tmp_out_1, tmp_out_2;
   reg [2:0] state, next_state;

   always @ (posedge clk or posedge reset)
   begin
      if (reset)
         state <= state_0;
      else
         state <= next_state;
   end
   always @ (state or in_1 or in_2)
   begin
      tmp_out_0 = in_1 + in_2;
      tmp_out_1 = in_1 - in_2;
      case (state)
         state_0: begin
            tmp_out_2 <= in_1 + 5'b00001;
            next_state <= state_1;
         end
         state_1: begin
            if (in_1 < in_2) begin
               next_state <= state_2;
               tmp_out_2 <= tmp_out_0;
            end
            else begin
               next_state <= state_3;
               tmp_out_2 <= tmp_out_1;
            end
         end
         state_2: begin
            tmp_out_2 <= tmp_out_0 - 5'b00001;
            next_state <= state_3;
         end
         state_3: begin
            tmp_out_2 <= tmp_out_1 + 5'b00001;
            next_state <= state_0;
         end
         state_4:begin
            tmp_out_2 <= in_2 + 5'b00001;
            next_state <= state_0;
         end
         default:begin
            tmp_out_2 <= 5'b00000;
            next_state <= state_0;
         end
      endcase
   end
   assign out = tmp_out_2;
endmodule
```

An equivalent implementation of this state machine can be achieved by using `` `define `` instead of the `parameter` data type, as follows:

```
`define state_0 3'b000
`define state_1 3'b001
`define state_2 3'b010
`define state_3 3'b011
`define state_4 3'b100
```

In this case, the `state` and `next_state` assignments are assigned a `` `state_x `` instead of a `state_x`, as shown in the following example:

```
next_state <= `state_3;
```

Although the `` `define `` construct is supported, Altera strongly recommends the use of the `parameter` data type because doing so preserves the state names throughout synthesis.

SystemVerilog State Machine Coding Example

The module `enum_fsm` shown in Example 6–45 is an example of a SystemVerilog state machine implementation that uses enumerated types. Altera recommends using this coding style to describe state machines in SystemVerilog.

In Quartus II integrated synthesis, the enumerated type that defines the states for the state machine must be of an unsigned integer type as shown in Example 6–45. If you do not specify the enumerated type as `int unsigned`, a signed `int` type is used by default. In this case, the Quartus II integrated synthesis synthesizes the design, but does not infer or optimize the logic as a state machine.

Example 6–45. SystemVerilog State Machine Using Enumerated Types

```
module enum_fsm (input clk, reset, input int data[3:0], output int o);

enum int unsigned { S0 = 0, S1 = 2, S2 = 4, S3 = 8 } state, next_state;

always_comb begin : next_state_logic
      next_state = S0;
      case(state)
        S0: next_state = S1;
        S1: next_state = S2;
        S2: next_state = S3;
        S3: next_state = S3;
      endcase
end

always_comb begin
      case(state)
         S0: o = data[3];
         S1: o = data[2];
         S2: o = data[1];
         S3: o = data[0];
      endcase
end

always_ff@(posedge clk or negedge reset) begin
      if(~reset)
         state <= S0;
      else
         state <= next_state;
end
endmodule
```

VHDL State Machines

To ensure proper recognition and inference of VHDL state machines, represent the states in a state machine with enumerated types and use the corresponding types to make state assignments. This implementation makes the state machine easier to read and reduces the risk of errors during coding. If the state is not represented by an enumerated type, synthesis software (such as Quartus II integrated synthesis) does not recognize the state machine. Instead, the state machine is implemented as regular logic gates and registers and the state machine is not listed as a state machine in the **Analysis & Synthesis** section of the Quartus II Compilation Report. In this case, the software does not perform any of the optimizations that are specific to state machines.

VHDL State Machine Coding Example

The following entity, `vhd1_fsm`, is an example of a typical VHDL state machine implementation (Example 6–46).

This state machine has five states. The asynchronous reset sets the variable `state` to `state_0`. The sum of `in1` and `in2` is an output of the state machine in `state_1` and `state_2`. The difference (`in1` - `in2`) is also used in `state_1` and `state_2`. The temporary variables `tmp_out_0` and `tmp_out_1` store the sum and the difference of `in1` and `in2`. Using these temporary variables in the various states of the state machine ensures proper resource sharing between the mutually exclusive states.

Example 6–46. VHDL State Machine (Part 1 of 2)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY vhdl_fsm IS
   PORT(
      clk: IN STD_LOGIC;
      reset: IN STD_LOGIC;
      in1: IN UNSIGNED(4 downto 0);
      in2: IN UNSIGNED(4 downto 0);
      out_1: OUT UNSIGNED(4 downto 0)
      );
END vhdl_fsm;

ARCHITECTURE rtl OF vhdl_fsm IS
   TYPE Tstate IS (state_0, state_1, state_2, state_3, state_4);
   SIGNAL state: Tstate;
   SIGNAL next_state: Tstate;

BEGIN
   PROCESS(clk, reset)
   BEGIN
      IF reset = '1' THEN
            state <=state_0;
      ELSIF rising_edge(clk) THEN
            state <= next_state;
      END IF;
   END PROCESS;

PROCESS (state, in1, in2)
      VARIABLE tmp_out_0: UNSIGNED (4 downto 0);
      VARIABLE tmp_out_1: UNSIGNED (4 downto 0);
```

Example 6-46. VHDL State Machine (Part 2 of 2))

```
   BEGIN
      tmp_out_0 := in1 + in2;
      tmp_out_1 := in1 - in2;
      CASE state IS
         WHEN state_0 =>
            out_1 <= in1;
            next_state <= state_1;
         WHEN state_1 =>
            IF (in1 < in2) then
               next_state <= state_2;
               out_1 <= tmp_out_0;
            ELSE
               next_state <= state_3;
               out_1 <= tmp_out_1;
            END IF;
         WHEN state_2 =>
            IF (in1 < "0100") then
               out_1 <= tmp_out_0;
            ELSE
               out_1 <= tmp_out_1;
            END IF;
               next_state <= state_3;

         WHEN state_3 =>
               out_1 <= "11111";
               next_state <= state_4;
         WHEN state_4 =>
               out_1 <= in2;
               next_state <= state_0;
         WHEN OTHERS =>
               out_1 <= "00000";
               next_state <= state_0;
      END CASE;
   END PROCESS;
END rtl;
```

Multiplexers

Multiplexers form a large portion of the logic utilization in many FPGA designs. By optimizing your multiplexer logic, you ensure the most efficient implementation in your Altera device. This section addresses common problems and provides design guidelines to achieve optimal resource utilization for multiplexer designs. The section also describes various types of multiplexers, and how they are implemented in the 4-input LUT found in many FPGA architectures, such as Altera's Stratix devices.

Stratix II and other high-performance devices have 6-input ALUTs and are not specifically addressed here. Although many of the principles and techniques for optimization are similar, device utilization differs in the 6-input LUT devices. For example, these devices can implement wider multiplexers in one ALM than can be implemented in the 4-input LUT of an LE.

Quartus II Software Option for Multiplexer Restructuring

Quartus II integrated synthesis provides the **Restructure Multiplexers** logic option that extracts and optimizes buses of multiplexers during synthesis. In certain situations, this option automatically performs some of the recoding functions described in this section without changing the HDL code in your design. This option is on by default, when the **Optimization technique** is set to **Balanced** (the default for most device families) or set to **Area**.

For details, refer to the *Restructure Multiplexers* subsection in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Even with this Quartus II-specific option turned on, it is beneficial to understand how your coding style can be interpreted by your synthesis tool, and avoid the situations that can cause problems in your design.

Multiplexer Types

This subsection addresses how multiplexers are created from various types of HDL code. `CASE` statements, `IF` statements, and state machines are all common sources of multiplexer logic in designs. These HDL structures create different types of multiplexers including binary multiplexers, selector multiplexers, and priority multiplexers. Understanding how multiplexers are created from HDL code and how they might be implemented during synthesis is the first step towards optimizing multiplexer structures for best results.

Binary Multiplexers

Binary multiplexers select inputs based on binary-encoded selection bits. Example 6–47 shows Verilog HDL code for two ways to describe a simple 4:1 binary multiplexer.

Example 6–47. Verilog HDL Binary-Encoded Multiplexers

```
case (sel)
   2'b00: z = a;
   2'b01: z = b;
   2'b10: z = c;
   2'b11: z = d;
endcase
```

A 4:1 binary multiplexer is efficiently implemented by using two 4-input LUTs. Larger binary multiplexers can be constructed that use the 4:1 multiplexer; constructing an N-input multiplexer (N:1 multiplexer) from a tree of 4:1 multiplexers can result in a structure using as few as 0.66*(N - 1) LUTs.

Selector Multiplexers

Selector multiplexers have a separate select line for each data input. The select lines for the multiplexer are one-hot encoded. Example 6–48 shows a simple Verilog HDL code example describing a one-hot selector multiplexer.

Example 6–48. Verilog HDL One-Hot-Encoded Case Statement

```
case (sel)
   4'b0001: z = a;
   4'b0010: z = b;
   4'b0100: z = c;
   4'b1000: z = d;
   default: z = 1'bx;
endcase
```

Selector multiplexers are commonly built as a tree of AND and OR gates. Using this scheme, two inputs can be selected using two select lines in a single 4-input LUT that uses two AND gates and an OR gate. The outputs of these LUTs can be combined with a wide OR gate. An N-input selector multiplexer of this structure requires at least 0.66*(N-0.5) LUTs, which is just slightly worse than the best binary multiplexer.

Priority Multiplexers

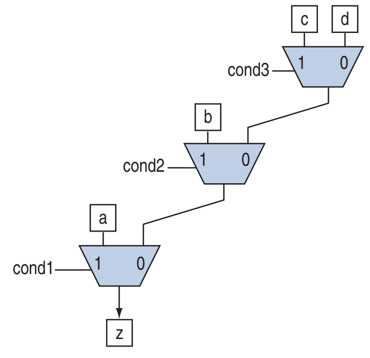
In priority multiplexers, the select logic implies a priority. The options to select the correct item must be checked in a specific order based on signal priority. These structures commonly are created from `IF`, `ELSE`, `WHEN`, `SELECT`, and `?:` statements in VHDL or Verilog HDL. The example VHDL code in Example 6–49 probably result in the schematic implementation illustrated in Figure 6–4.

Example 6–49. VHDL IF Statement Implying Priority

```
IF cond1 THEN z <= a;
ELSIF cond2 THEN z <= b;
ELSIF cond3 THEN z <= c;
ELSE z <= d;
END IF;
```

The multiplexers shown in Figure 6–4 form a chain, evaluating each condition or select bit, one at a time.

Figure 6–4. Priority Multiplexer Implementation of an IF Statement



An N-input priority multiplexer uses a LUT for every 2:1 multiplexer in the chain, requiring N-1 LUTs. This chain of multiplexers generally increases delay because the critical path through the logic traverses every multiplexer in the chain.

To improve the timing delay through the multiplexer, avoid priority multiplexers if priority is not required. If the order of the choices is not important to the design, use a `CASE` statement to implement a binary or selector multiplexer instead of a priority multiplexer. If delay through the structure is important in a multiplexed design requiring priority, consider recoding the design to reduce the number of logic levels to minimize delay, especially along your critical paths.

Default or Others Case Assignment

To fully specify the cases in a `CASE` statement, include a `default` (Verilog HDL) or `OTHERS` (VHDL) assignment. This assignment is especially important in one-hot encoding schemes where many combinations of the select lines are unused. Specifying a case for the unused select line combinations gives the synthesis tool information about how to synthesize these cases, and is required by the Verilog HDL and VHDL language specifications.

Some designs do not require that the outcome in the unused cases be considered, often because designers assume these cases will not occur. For these types of designs, you can choose any value for the `default` or `OTHERS` assignment. However, be aware that the assignment value you choose can have a large effect on the logic utilization required to implement the design due to the different ways synthesis tools treat different values for the assignment, and how the synthesis tools use different speed and area optimizations.

In general, to obtain best results, explicitly define invalid `CASE` selections with a separate `default` or `OTHERS` statement instead of combining the invalid cases with one of the defined cases.

If the value in the invalid cases is not important, specify those cases explicitly by assigning the `X` (don’t care) logic value instead of choosing another value. This assignment allows your synthesis tool to perform the best area optimizations.

You can experiment with different `default` or `OTHERS` assignments for your HDL design and your synthesis tool to test the effect they have on logic utilization in your design.

Implicit Defaults

The `IF` statements in Verilog HDL and VHDL can be a convenient way to specify conditions that do not easily lend themselves to a `CASE`-type approach. However, using `IF` statements can result in complicated multiplexer trees that are not easy for synthesis tools to optimize.

In particular, every `IF` statement has an implicit `ELSE` condition, even when it is not specified. These implicit defaults can cause additional complexity in a multiplexed design.

The code in Example 6–50 represents a multiplexer with four inputs (`a`, `b`, `c`, `d`) and one output (`z`). Altera does not recommend using this coding style.

Example 6–50. VHDL IF Statement with Implicit Defaults

```
IF cond1 THEN
   IF cond2 THEN
      z <= a;
   END IF;
ELSIF cond3 THEN
   IF cond4 THEN
      z <= b;
   ELSIF cond5 THEN
      z <= c;
   END IF;
ELSIF cond6 THEN
   z <= d;
END IF;
```

Although the code appears to implement a 4:1 multiplexer, each of the three separate IF statements in the code has an implicit ELSE condition that is not specified. Because the output values for the ELSE cases are not specified, the synthesis tool assumes the intent is to maintain the same output value for these cases and infers a combinational loop, such as a latch. Latches add to the design's logic utilization and can also make timing analysis difficult and lead to other problems.

The code sample shown in Example 6–51 shows code with the same functionality as the code shown in Example 6–50, but specifies the ELSE cases explicitly. (This is not a recommended coding style improvement, but it explicitly shows the default conditions from the previous example.)

Example 6–51. VHDL IF Statement with Default Conditions Explicitly Specified

```
IF cond1 THEN
   IF cond2 THEN
      z <= a;
   ELSE
      z <= z;
   END IF;
ELSIF cond3 THEN
   IF cond4 THEN
      z <= b;
   ELSIF cond5 THEN
      z <= c;
   ELSE
      z <= z;
   END IF;
ELSIF cond6 THEN
   z <= d;
ELSE
   z <= z;
END IF;
```

Figure 6–5 is a schematic representing the code in Example 6–51, which illustrates that the multiplexer logic is significantly more complicated than a basic 4:1 multiplexer, although there are only four inputs.

Figure 6–5. Multiplexer Implementation of an IF Statement with Implicit Defaults

There are several ways you can simplify the multiplexed logic and remove the unrequired defaults. The optimal method may be to recode the design so the logic takes the structure of a 4:1 `CASE` statement. Alternatively, if priority is important, you can restructure the code to deduce default cases and flatten the multiplexer. In this example, instead of `IF cond1 THEN IF cond2`, use `IF (cond1 AND cond2)`, which performs the same function. In addition, examine whether the defaults are don't care cases. In this example, you can promote the last `ELSIF cond6` statement to an `ELSE` statement if no other valid cases can occur.

Avoid unnecessary default conditions in your multiplexer logic to reduce the complexity and logic utilization required to implement your design.

Degenerate Multiplexers

A degenerate multiplexer is a multiplexer in which not all of the possible cases are used for unique data inputs. The cases that are not required tend to contribute to inefficiency in the logic utilization for these multiplexers. You can recode degenerate multiplexers so they take advantage of the efficient logic utilization possible with full binary multiplexers.

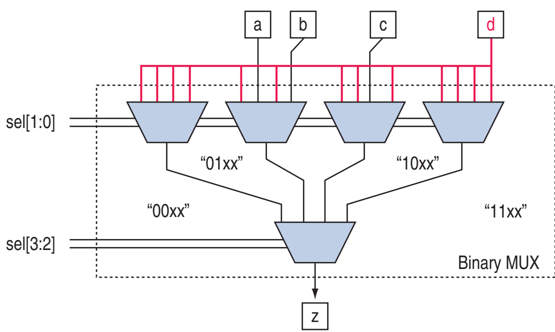
Example 6–52 shows a `VHDL CASE` statement describing a degenerate multiplexer.

Example 6–52. VHDL CASE Statement Describing a Degenerate Multiplexer

```
CASE sel[3:0] IS
   WHEN "0101" => z <= a;
   WHEN "0111" => z <= b;
   WHEN "1010" => z <= c;
   WHEN OTHERS => z <= d;
END CASE;
```

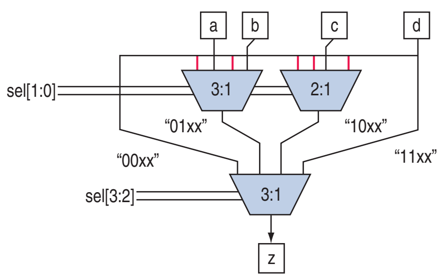
The number of select lines in a binary multiplexer normally dictates the size of a multiplexer required to implement the desired function. For example, the multiplexer structure represented in Figure 6–6 has four select lines capable of implementing a binary multiplexer with 16 inputs. However, the design does not use all 16 inputs, which makes this multiplexer a degenerate 16:1 multiplexer.

Figure 6–6. Binary Degenerate Multiplexer



In Figure 6–6, the first and fourth multiplexers in the top level can easily be eliminated because all four inputs to each multiplexer are the same value, and the number of inputs to the other multiplexers can be reduced, as shown in Figure 6–7.

Figure 6–7. Optimized Version of the Degenerate Binary Multiplexer



Implementing this version of the multiplexer still requires at least five 4-input LUTs, two for each of the remaining 3:1 multiplexers and one for the 2:1 multiplexer. This design selects an output from only four inputs, a 4:1 binary multiplexer can be implemented optimally in two LUTs, so this degenerate multiplexer tree reduces the efficiency of the logic.

You can improve logic utilization of this structure by recoding the select lines to implement a full 4:1 binary multiplexer. The code sample shown in Example 6–53 shows a recoder design that translates the original select lines into the `z_sel` signal with binary encoding.

Example 6–53. VHDL Recoder Design for Degenerate Binary Multiplexer

```
CASE sel[3:0] IS
   WHEN "0101" => z_sel <= "00";
   WHEN "0111" => z_sel <= "01";
   WHEN "1010" => z_sel <= "10";
   WHEN OTHERS => z_sel <= "11";
END CASE;
```

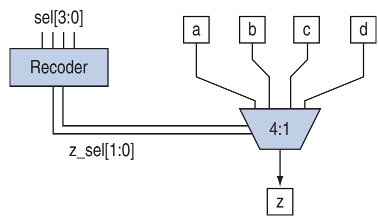
The code sample shown in Example 6–54 shows you how to implement the full binary multiplexer.

Example 6–54. VHDL 4:1 Binary Multiplexer Design

```
CASE z_sel[1:0] IS
   WHEN "00" => z <= a;
   WHEN "01" => z <= b;
   WHEN "10" => z <= c;
   WHEN "11" => z <= d;
END CASE;
```

Use the new `z_sel` control signal from the recoder design to control the 4:1 binary multiplexer that chooses between the four inputs `a`, `b`, `c`, and `d`, as illustrated in Figure 6–8. The complexity of the select lines is handled in the recoder design, and the data multiplexing is performed with simple binary select lines enabling the most efficient implementation.

Figure 6–8. Binary Multiplexer with Recorder



The design for the recoder can be implemented in two LUTs and the efficient 4:1 binary multiplexer uses two LUTs, for a total of four LUTs. The original degenerate multiplexer required five LUTs, so the recoded version uses 20% less logic than the original.

You can often improve the logic utilization of multiplexers by recoding the select lines into full binary cases. Although logic is required to perform the encoding, the overall logic utilization is often improved.

Buses of Multiplexers

The inputs to multiplexers are often data input buses in which the same multiplexer function is performed on a set of data input buses. In these cases, any inefficiency in the multiplexer is multiplied by the number of bits in the bus. The issues described in the previous sections become even more important for wide multiplexer buses.

For example, the recoding of select lines into full binary cases detailed in the previous section can often be used in multiplexed buses. Recoding the select lines may have to be completed only once for all the multiplexers in the bus. By sharing the recoder logic among all the bits in the bus, you can greatly improve the logic efficiency of a bus of multiplexers.

The degenerate multiplexer in the previous section requires five LUTs to implement. If the inputs and output are 32 bits wide, the function could require `32 ¥ 5` or 160 LUTs for the whole bus. The recoder design uses only two LUTs, and the select lines only have to be recoded once for the entire bus. The binary 4:1 multiplexer requires two LEs per bit of the bus. The total logic utilization for the recoded version could be 2 + (2 × 32) or 66 LUTs for the whole bus, compared to 160 LUTs for the original version. The logic savings become more important with wide multiplexer buses.

Using techniques to optimize degenerate multiplexers, removing implicit defaults are not required, and choosing the optimal `DEFAULT` or `OTHERS` case can play an important role when optimizing buses of multiplexers.

Cyclic Redundancy Check Functions

CRC computations are used heavily by communications protocols and storage devices to detect any corruption of the data. These functions are highly effective; there is a very low probability that corrupted data can pass a 32-bit CRC check.

CRC functions typically use wide XOR gates to compare the data. The way that synthesis tools flatten and factor these XOR gates to implement the logic in FPGA LUTs can greatly impact the area and performance results for the design. XOR gates have a cancellation property which creates an exceptionally large number of reasonable factoring combinations, so synthesis tools cannot always choose the best result by default.

The 6-input ALUT has a significant advantage over 4-input LUTs for these designs. When properly synthesized, CRC processing designs can run at high speeds in devices with 6-input ALUTs.

The following guidelines help you improve the quality of results for CRC designs in Altera devices.

If Performance is Important, Optimize for Speed

Synthesis tools flatten XOR gates to minimize area and depth of levels of logic. Synthesis tools such as Quartus II integrated synthesis target area optimization by default for these logic structures. Therefore, for more focus on depth reduction, set the synthesis optimization technique to speed.

Flattening for depth sometimes causes a significant increase in area.

Use Separate CRC Blocks Instead of Cascaded Stages

Some designers optimize their CRC designs to use cascaded stages, for example, four stages of 8 bits. In such designs, intermediate calculations are used as required (such as the calculations after 8, 24, or 32 bits) depending on the data width. This design is not optimal in FPGA devices. The XOR cancellations that can be performed in CRC designs mean that the function does not require all the intermediate calculations to determine the final result. Therefore, forcing the use of intermediate calculations increases the area required to implement the function, as well as increasing the logic depth because of the cascading. It is typically better to create full separate CRC blocks for each data width that you require in the design, then multiplex them together to choose the appropriate mode at a given time.

Use Separate CRC Blocks Instead of Allowing Blocks to Merge

Synthesis tools often attempt to optimize CRC designs by sharing resources and extracting duplicates in two different CRC blocks because of the factoring options in the XOR logic. As addressed previously, the CRC logic allows significant reductions but this works best when each CRC function is optimized separately. Check for duplicate extraction behavior if you have different CRC functions that are driven by common data signals or that feed the same destination signals.

If you are having problems with the quality of results and you see that two CRC functions are sharing logic, ensure that the blocks are synthesized independently using one of the following methods:

- Define each CRC block as a separate design partition in an incremental compilation design flow

 For details, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

- Synthesize each CRC block as a separate project in your third-party synthesis tool and then write a separate Verilog Quartus Mapping (**.vqm**) or EDIF netlist file for each

Take Advantage of Latency if Available

If your design can use more than one cycle to implement the CRC functionality, adding registers and retiming the design can help reduce area, improve performance, and reduce power utilization. If your synthesis tool offers a retiming feature (such as the Quartus II software **Perform gate-level register retiming** option), you can insert an extra bank of registers at the input and allow the retiming feature to move the registers for better results. You can also build the CRC unit half as wide and alternate between halves of the data in each clock cycle.

Save Power by Disabling CRC Blocks When Not in Use

CRC designs are heavy consumers of dynamic power because the logic toggles whenever there is a change in the design. To save power, use clock enables to disable the CRC function for every clock cycle that the logic is not required. Some designs don't check the CRC results for a few clock cycles while other logic is performed. It is valuable to disable the CRC function even for this short amount of time.

Use the Device Synchronous Load (sload) Signal to Initialize

The data in many CRC designs must be initialized to 1's before operation. If your target device supports the use of the `sload` signal, you should use it to set all the registers in your design to 1's before operation. To enable use of the `sload` signal, follow the coding guidelines presented in "Secondary Register Control Signals Such as Clear and Clock Enable" on page 6–38. You can check the register equations in the Timing Closure Floorplan or the Chip Planner to ensure that the signal was used as expected.

If you must force a register implementation using an `sload` signal, you can use low-level device primitives as described in the *Designing with Low-Level Primitives User Guide*.

Comparators

Synthesis software, including Quartus II integrated synthesis, uses device and context-specific implementation rules for comparators (<, >, or ==) and selects the best one for your design. This section provides some information about the different types of implementations available and provides suggestions on how you can code your design to encourage a specific implementation.

The == comparator is implemented in general logic cells. The < comparison can be implemented using the carry chain or general logic cells. In devices with 6-input ALUTs, the carry chain is capable of comparing up to three bits per cell. In devices with 4-input LUTs, the capacity is one bit of comparison per cell, similar to an add/subtract chain. The carry chain implementation tends to be faster than the general logic on standalone benchmark test cases, but can result in lower performance when it is part of a larger design due to the increased restriction on the Fitter. The area requirement is similar for most input patterns. The synthesis software selects an appropriate implementation based on the input pattern.

If you are using Quartus II integrated synthesis, you can guide the synthesis by using specific coding styles. To select a carry chain implementation explicitly, rephrase your comparison in terms of addition. As a simple example, the following coding style allows the synthesis tool to select the implementation, which is most likely using general logic cells in modern device families:

```
wire [6:0] a,b;
wire alb = a<b;
```

In the following coding style, the synthesis tool uses a carry chain (except for a few cases, such as when the chain is very short or the signals `a` and `b` minimize to the same signal):

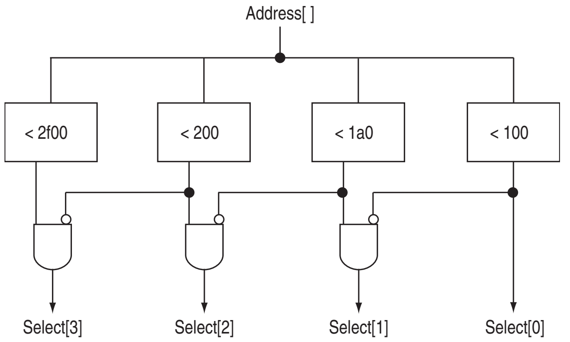
```
wire [6:0] a,b;
wire [7:0] tmp = a - b;
wire alb = tmp[7]
```

This second coding style uses the top bit of the `tmp` signal, which is 1 in twos complement logic if *a* is less than *b*, because the subtraction *a* – *b* results in a negative number.

If you have any information about the range of the input, you have "don't care" values that you can use to optimize the design. Because this information is not available to the synthesis tool, you can often reduce the device area required to implement the comparator with specific hand implementation of the logic.

You can also check whether a bus value is within a constant range with a small amount of logic area by using the logic structure shown in Figure 6–9. This type of logic occurs frequently in address decoders.

Figure 6–9. Example Logic Structure for Using Comparators to Check a Bus Value Range



Counters

Implementing counters in HDL code is easy; they are implemented with an adder followed by registers. Remember that the register control signals, such as enable (`ena`), synchronous clear (`sclr`), and synchronous load (`sload`), are available. For the best area utilization, ensure that the up/down control or controls are expressed in terms of one addition instead of two separate addition operators.

If you use the following coding style, your synthesis tool may implement two separate carry chains for addition (if it doesn't detect the issue and optimize the logic):

```
out <= count_up ? out + 1 : out - 1;
```

The following coding style requires only one adder along with some other logic:

```
out <= out + (count_up ? 1 : -1);
```

In this case, the coding style better matches the device hardware because there is only one carry chain adder, and the –1 constant logic is implemented in the LUT in front of the adder without adding extra area utilization.

Designing with Low-Level Primitives

Low-level HDL design is the practice of using low-level primitives and assignments to dictate a particular hardware implementation for a piece of logic. Low-level primitives are small architectural building blocks that assist you in creating your design. With the Quartus II software, you can use low-level HDL design techniques to force a specific hardware implementation that can help you achieve better resource utilization or faster timing results.

Using low-level primitives is an advanced technique to help with specific design challenges, and is optional in the Altera design flow. For many designs, synthesizing generic HDL source code and Altera megafunctions gives you the best results.

Low-level primitives allow you to use the following types of coding techniques:

- Instantiate the logic cell or `LCELL` primitive to prevent Quartus II integrated synthesis from performing optimizations across a logic cell
- Create carry and cascade chains using `CARRY`, `CARRY_SUM`, and `CASCADE` primitives
- Instantiate registers with specific control signals using `DFF` primitives
- Specify the creation of LUT functions by identifying the LUT boundaries
- Use I/O buffers to specify I/O standards, current strengths, and other I/O assignments
- Use I/O buffers to specify differential pin names in your HDL code, instead of using the automatically-generated negative pin name for each pair

For details about and examples of using these types of assignments, refer to the *Designing with Low-Level Primitives User Guide*.

Conclusion

Because coding style and megafunction implementation can have such a large effect on your design performance, it is important to match the coding style to the device architecture from the very beginning of the design process. To improve design performance and area utilization, take advantage of advanced device features, such as memory and DSP blocks, as well as the logic architecture of the targeted Altera device by following the coding recommendations presented in this chapter.

For additional optimization recommendations, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Referenced Documents

This chapter references the following documents:

- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Advanced Synthesis Cookbook: A Design Guide for Stratix II, Stratix III, and Stratix IV Devices*
- *ALTSHIFT_TAPS Megafunction User Guide*
- *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*
- *Designing with Low-Level Primitives User Guide*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Synthesis* section in volume 1 of the *Quartus II Handbook*

Document Revision History

Table 6–2 shows the revision history for this chapter.

Table 6–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Corrected and updated several examples ■ Added support for Arria II GX devices ■ Other minor changes to chapter	Updated for the Quartus II 9.0 software release.
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	Updated for the Quartus II 8.1 software release.
May 2008 v8.0.0	Updates for the Quartus II software version 8.0 release, including: ■ Added information to “RAM Functions—Inferring ALTSYNCRAM and ALTDPRAM Megafunctions from HDL Code” on page 6–13 ■ Added information to “Avoid Unsupported Reset and Control Conditions” on page 6–14 ■ Added information to “Check Read-During-Write Behavior” on page 6–16 ■ Added two new examples to “ROM Functions—Inferring ALTSYNCRAM and LPM_ROM Megafunctions from HDL Code” on page 6–28: Example 6–24 and Example 6–25 ■ Added new section: “Clock Multiplexing” on page 6–46 ■ Added hyperlinks to references within the chapter ■ Minor editorial updates	Updates and enhancements to subject coverage for the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

7. Managing Metastability with the Quartus II Software

QII51018-9.0.0

Introduction

This chapter describes the industry-leading analysis, reporting, and optimization features that can help you manage metastability in Altera® devices. You can use the Quartus® II software to analyze the average mean time between failures (MTBF) due to metastability when a design synchronizes asynchronous signals, and optimize the design to improve the metastability MTBF. This chapter explains how to take advantage of these features in the Quartus II software, and provides guidelines to help you reduce the chance of metastability effects due to signal synchronization.

What is metastability? As is well known in digital design, all registers have defined signal timing requirements that allow the register to correctly capture data at the inputs and produce the output signal. To ensure reliable operation, the input to the register must be stable for a minimum time before the clock edge (register setup time or t_{SU}) and a minimum time after the clock edge (register hold time or t_H). The register output will then be available after a specified clock-to-output delay (t_{CO}). If the data violates a register's setup and/or hold time requirements, the output of the register may go into a metastable state. In this state, the register output hovers at a value between the high and low states, which means the output transition to a defined high or low state is delayed beyond the specified t_{CO}. Different destination registers might capture different values for the metastable signal, which can cause the system to fail.

In synchronous systems, the input signals must always meet the register timing requirements, so metastability does not occur. Metastability problems commonly occur when a signal is transferred between circuitry in unrelated or asynchronous clock domains, because the designer cannot guarantee that the signal will meet setup and hold time requirements in this case.

The MTBF due to metastability is an estimate of the average time between instances when metastability could cause a design failure. A high MTBF (such as hundreds or thousands of years between metastability failures) indicates a more robust design. You should determine an acceptable target MTBF given the context of your entire system and the fact that MTBF calculations are statistical estimates.

The metastability MTBF for a specific signal transfer, or all the transfers in a design, can be calculated using information about the design and the device characteristics. Improving the metastability MTBF for your design reduces the chance that signal transfers could cause metastability problems in your device.

For more information about metastability due to signal synchronization, its effects in FPGAs, and how MTBF is calculated, refer to the *Understanding Metastability in FPGAs* white paper. Your overall device MTBF is also affected by other FPGA failure mechanisms that you cannot control with your design. For information about Altera device reliability, refer to the *Reliability Report*.

The Quartus II software provides analysis, optimization, and reporting features to help manage metastability in Altera designs. These metastability features are supported only for designs constrained with the Quartus II TimeQuest Timing Analyzer, and for select device families. In Quartus II software version 9.0, typical MTBF values are generated for designs using Arria® II GX, Stratix® IV, Stratix III, and Cyclone® III devices. Additional worst-case MTBF values are reported for the fully characterized Stratix III devices.

This chapter contains the following topics:

- "Metastability Analysis in the Quartus II Software" on page 7–2, including the important first step "Identifying Synchronizers for Metastability Analysis"
- "Metastability and MTBF Reporting" on page 7–6
- "MTBF Optimization" on page 7–9
- "Reducing Metastability Effects" on page 7–11, including guidelines to ensure complete and accurate metastability analysis and some suggestions to follow if the Quartus II metastability reports calculate an unacceptable MTBF value

For details about device and version support for the metastability features in the Quartus II software, refer to Quartus II Help.

Metastability Analysis in the Quartus II Software

When a signal transfers between circuitry in unrelated or asynchronous clock domains, the first register in the new clock domain acts as a synchronization register. To minimize the failures due to metastability in asynchronous signal transfers, circuit designers typically use a sequence of registers (a synchronization register chain or synchronizer) in the destination clock domain to resynchronize the signal to the new clock domain and allow additional time for a potentially metastable signal to resolve to a known value. Designers commonly use two registers to synchronize a new signal, but Altera recommends using a standard of three registers for better metastability protection.

The TimeQuest Timing Analyzer can analyze and report the MTBF for each identified synchronizer that meets its timing requirements, and can generate an estimate of the overall design MTBF. The software uses this information to optimize the design MTBF, and you can use this information to determine whether you require longer synchronizer chains in your design.

The first step to enable metastability MTBF analysis and optimization in the Quartus II software is to identify which registers are part of a synchronization register chain.

This section contains the following subsections:

- "What is a Synchronization Register Chain?"
- "Identifying Synchronizers for Metastability Analysis" on page 7–4
- "How Timing Constraints Affect Synchronizer Chain Identification and Metastability Analysis" on page 7–6

For information about the reports generated by the TimeQuest Timing Analyzer, refer to "Metastability and MTBF Reporting" on page 7–6. For more information about optimizing the MTBF, refer to "MTBF Optimization" on page 7–9.

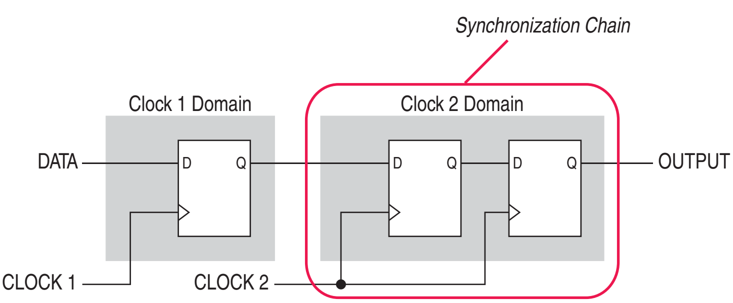
What is a Synchronization Register Chain?

A synchronization register chain, or synchronizer, is defined as a sequence of registers that meets the following requirements:

- The registers in the chain are all clocked by the same or phase-related clocks
- The first register in the chain is driven from an unrelated clock domain, or asynchronously
- Each register fans out to only one register, except the last register in the chain

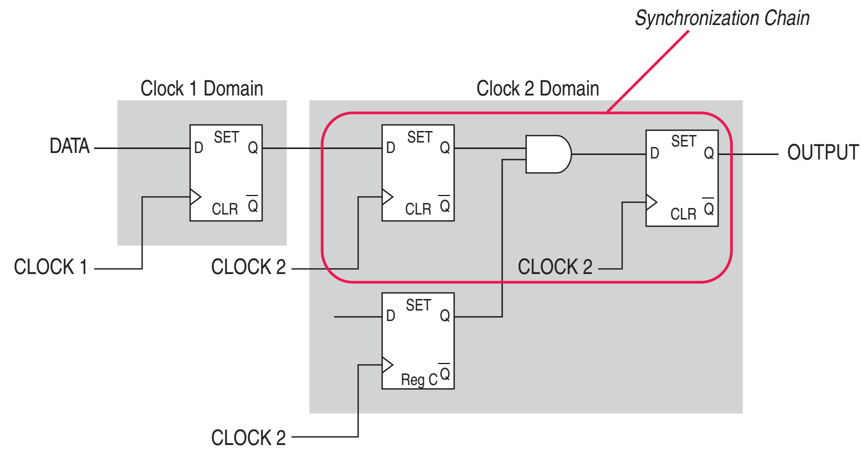
The length of the synchronization register chain is the number of registers in the synchronizing clock domain that meet the above requirements. Figure 7–1 shows a sample synchronization chain of length two, assuming the `OUTPUT` signal fans out to more than one register destination.

Figure 7–1. Sample Synchronization Register Chain



The path between synchronization registers can contain combinational logic, as long as all registers of the synchronization chain are in the same clock domain. Figure 7–2 shows an example of a synchronization register chain that includes logic between the registers.

Figure 7–2. Sample Synchronization Register Chain Containing Logic



The Quartus II software uses the design timing constraints to determine which connections are asynchronous signal transfers, as described in "How Timing Constraints Affect Synchronizer Chain Identification and Metastability Analysis" on page 7–6.

Synchronization registers allow time for a potentially metastable signal to resolve to a known value before the signal is used in the rest of the design. The timing slack available in the synchronizer register-to-register paths provides time for a metastable signal to settle, and is known as the available settling time. The available settling time for a synchronization chain is the sum of the output timing slacks for each register in the chain. Adding available settling time with additional synchronization registers improves the metastability MTBF.

Identifying Synchronizers for Metastability Analysis

The first step to enable metastability MTBF analysis and optimization in the Quartus II software is to identify which registers are part of a synchronization register chain with the **Synchronizer Identification** option.

You can apply synchronizer identification settings globally, to a design entity, or to specific registers of a synchronization chain. You can use the global options in "Using the Global Synchronizer Identification Setting" on page 7–4 to automatically list or analyze possible synchronizers. You should review this list of possible synchronizers, and identify the confirmed synchronization chains with specific registers assignments as described in "Refining Synchronizer Identification Using the Instance-Specific Assignment" on page 7–5.

Synchronization chains are already identified within most Altera intellectual property (IP) cores.

Using the Global Synchronizer Identification Setting

To set the global **Synchronizer Identification** option, on the Assignments menu, click **Settings**. Under **Timing Analysis Settings**, click on the **TimeQuest Timing Analyzer** page and select the appropriate **Synchronizer Identification** setting under **Metastability Analysis**: **Off**, **Auto**, or **Forced If Asynchronous**. To apply the global assignment with Tcl, use the following command:

```
set_global_assignment -name SYNCHRONIZER_IDENTIFICATION
<OFF|AUTO|"FORCED IF ASYNCHRONOUS">
```

Use the following guidelines to choose the global setting:

- The default global **Off** setting means that no synchronization registers are automatically analyzed.
- Use the global **Auto** setting to generate a list of likely synchronization chains in your design, based on the software's automatic synchronizer detection criteria.
  - With this setting, any chain synchronizing an asynchronous signal with more than one register is listed as a likely synchronizer if the chain does not contain logic.
  - MTBF is not reported or optimized for automatically detected register chains.

- You can use the global **Forced If Asynchronous** setting to report and optimize MTBF for all asynchronous signal transfers in the design.
  - This setting forces synchronization register identification and MTBF analysis if the software detects any asynchronous signal transfer, even if there is combinational logic or only one register in the synchronization register chain.
  - This setting is likely to identify some registers that were not designed as synchronizers, and thus might report an MTBF that is too conservative. For example, asynchronous reset signals to registers and SignalTap® II Embedded Logic Analyzer signals might generate metastability reports.
  - You can turn off synchronizer identification for specific registers using the instance-specific option, as described below.

Refining Synchronizer Identification Using the Instance-Specific Assignment

Altera recommends applying the **Synchronizer Identification** option **Forced If Asynchronous** to each register in a synchronization chain. Instance-specific assignments ensure that the software analyzes an accurate set of synchronizers to report an accurate MTBF. If you use the global **Auto Synchronizer Identification** setting to detect likely synchronizers, use specific assignments to identify the true synchronizers for MTBF analysis.

Use the Assignment Editor on the Assignments menu to set the **Synchronizer Identification** option on a register or design entity to **Auto**, **Forced If Asynchronous**, **Forced**, or **Off**. To apply the assignment with Tcl, use the following command:

```
set_instance_assignment -name SYNCHRONIZER IDENTIFICATION <AUTO|"FORCED
IF ASYNCHRONOUS"|FORCED|OFF> -to <register name>
```

Use the following guidelines to choose the register-specific settings:

- Enable MTBF analysis and optimization by identifying each synchronizer register with the **Forced If Asynchronous** setting
  - Use the register chains listed by the global **Auto Synchronizer Identification** option to help you identify these registers.
  - There may be additional synchronization chains in your design that are not detected by the **Auto** setting because they contain logic or only one register.
- If you have a specific register or register chain in your design that the software detects as synchronous but you want to be analyzed and optimized for metastability like an asynchronous signal, apply the **Forced** setting to the first synchronization register in the chain.
  - Use this setting as a manual "override" for a signal that the TimeQuest Timing Analyzer does not report as asynchronous, such as a virtual pin associated with a virtual clock.
  - This **Forced** setting is not available globally in the **Settings** dialog box because making this setting globally would incorrectly identify every design register as a synchronizer.

- If some register chains are misidentified as synchronizers when you apply a global or entity level **Forced If Asynchronous** setting, you can disable metastability analysis for specific registers. To do so, use the Assignment Editor to set **Synchronizer Identification** to **Off** for the first synchronization register in these register chains.

How Timing Constraints Affect Synchronizer Chain Identification and Metastability Analysis

The TimeQuest Timing Analyzer can analyze metastability MTBF only if a synchronization chain meets its timing requirements. The metastability failure rate depends on the timing slack available in the synchronizer's register-to-register connections, because that slack is the available settling time for a potential metastable signal. Therefore, it is important for your design to be correctly constrained with the real application frequency requirements to get an accurate MTBF report.

In addition, the **Auto** and **Forced If Asynchronous** synchronizer identification options use timing constraints to automatically detect the synchronizer chains in the design. These options check for signal transfers between circuitry in unrelated or asynchronous clock domains, so clock domains must be related correctly with timing constraints.

The TimeQuest Timing Analyzer views input ports as asynchronous signals unless they are associated correctly with a clock domain. If an input port fans out to registers that are not acting as synchronization registers, you should apply a `set_input_delay` constraint to the input port; otherwise, the input register is reported as a synchronization register. If you constrain a synchronous input port with a `set_max_delay` constraint for a setup (t_{SU}) requirement, this does not prevent synchronizer identification because the constraint does not associate the input port with a clock domain.

Instead, use the following syntax to specify an input setup requirement associated with a clock:

```
set_input_delay -max -clock <clock name> <latch – launch – tSU
requirement> <input port name>
```

Registers that are at the end of false paths are also considered synchronization registers because false paths are not timing-analyzed. Because there are no timing requirements for these paths, the signal may change at any point, which may violate the t_{SU} and t_H of the register. Therefore, these registers are identified as synchronization registers. If these registers are not used for synchronization, you can turn off synchronizer identification and analysis as described in the previous section.

Metastability and MTBF Reporting

The Quartus II software reports the metastability analysis results in the Compilation Report and TimeQuest Timing Analyzer reports as described in "Metastability Report". The MTBF calculation uses timing and structural information about the design, silicon characteristics, and operating conditions, along with the data toggle rate described in "Synchronizer Data Toggle Rate in MTBF Calculation" on page 7–9.

For more information about how metastability MTBF is calculated, refer to the *Understanding Metastability in FPGAs* white paper.

 © March 2009 Altera Corporation

If you change the **Synchronizer Identification** settings, you can generate new metastability reports by rerunning the TimeQuest Timing Analyzer. However, you should rerun the Fitter first so that the registers identified with the new setting can be optimized for metastability MTBF. For information about metastability optimization, refer to "MTBF Optimization" on page 7–9.

Metastability Report

The Metastability Report provides a summary of the metastability analysis results. In addition to the MTBF Summary and Synchronizer Summary reports, the TimeQuest Timing Analyzer tool reports additional statistics in a report for each synchronizer chain. This section provides more information about the reports.

To view the MTBF Summary and Synchronizer Summary reports, open the Metastability Report in the **TimeQuest Timing Analyzer** section of the Compilation Report. If the software performs multicorner timing analysis, expand the timing analysis results for one of the timing corners, and then select the Metastability Report for those operating conditions.

To view the additional synchronizer Statistics in the TimeQuest report, open the TimeQuest Timing Analyzer from the Tools menu, and double-click **Report Metastability** in the **Tasks** list (or use the `report_metastability` command). You can generate the reports with Tcl commands in addition to the TimeQuest Timing Analyzer user interface; refer to "Scripting Support" on page 7–13 for details.

If the design uses only the **Auto Synchronizer Identification** setting, the reports list likely synchronizers but do not report MTBF. To obtain an MTBF for each register chain, force identification of synchronization registers as described in "Identifying Synchronizers for Metastability Analysis" on page 7–4.

If the synchronizer chain does not meet its timing requirements, the reports list identified synchronizers but do not report MTBF. To obtain MTBF calculations, ensure that the design is properly constrained and that the synchronizer meets its timing requirements, as described in "How Timing Constraints Affect Synchronizer Chain Identification and Metastability Analysis" on page 7–6.

MTBF Summary Report

The MTBF Summary reports an estimate of the overall robustness of cross-clock domain and asynchronous transfers in the design. This estimate uses the MTBF results of all synchronization chains in the design to calculate an MTBF for the entire design.

The MTBF Summary Report reports the **Typical MTBF of Design** for supported devices and the **Worst-Case MTBF of Design** for supported fully-characterized devices. The typical MTBF result assumes typical conditions, defined as nominal silicon characteristics for the selected device speed grade, as well as nominal operating conditions. The worst case MTBF result uses the worst case silicon characteristics for the selected device speed grade.

When you analyze multiple timing corners in the TimeQuest Timing Analyzer, the MTBF calculation may vary because of changes in the operating conditions, and the timing slack or available metastability settling time. Altera recommends running multi-corner timing analysis to ensure that you analyze the worst MTBF results, because the worst timing corner for MTBF does not necessarily match the worst corner for timing performance. In the **Settings** dialog box, under **Timing Analysis Settings**, click the **TimeQuest Timing Analyzer** page, and turn on **Enable multicorner timing analysis during compilation**.

The MTBF Summary report also lists the **Number of Synchronizer Chains Found** and the length of the **Shortest Synchronizer Chain**, which can help you identify whether the report is based on accurate information. If the number of synchronizer chains found is different from what you expect, or if the length of the shortest synchronizer chain is less than you expect, you might have to add or change **Synchronizer Identification** settings for the design. The report also provides the **Worst Case Available Settling Time**, defined as the available settling time for the synchronizer with the worst MTBF.

You can use the reported **Fraction of Chains for which MTBFs Could Not be Calculated** to determine whether a high proportion of chains are missing in the metastability analysis. A fraction of 1, for example, means that MTBF could not be calculated for any chains in the design. MTBF is not calculated if you have not identified the chain with the appropriate Synchronizer Identification option, or if paths are not timing-analyzed and therefore have no valid slack for metastability analysis. You might have to correct your timing constraints to enable complete analysis of the applicable register chains.

Finally, the MTBF Summary report specifies how an increase of 100ps in available settling time increases the MTBF values. If your MTBF is not satisfactory, this metric can help you determine how much extra slack would be required in your synchronizer chain to allow you to reach the desired design MTBF.

Synchronizer Summary

The **Synchronizer Summary** lists the synchronization register chains detected in the design depending on the Synchronizer Identification setting. The **Source Node** is the register or input port that is the source of the asynchronous transfer. The **Synchronization Node** is the first register of the synchronization chain. The **Source Clock** is the clock domain of the source node, and the **Synchronization Clock** is the clock domain of the synchronizer chain.

This summary reports the calculated **Worst-Case MTBF**, if available, and the **Typical MTBF**, for each appropriately identified synchronization register chain that meets its timing requirement. To see more detail about each synchronizer, refer to the TimeQuest statistics report described in the following section.

Synchronizer Chain Statistics Report in the TimeQuest Timing Analyzer

The TimeQuest Timing Analyzer provides an additional report for each synchronizer chain. The **Chain Summary** tab matches the Synchronizer Summary information described in the previous section, while the **Statistics** tab adds more details. The Statistics list whether the **Method of Synchronizer Identification** was **User Specified** (with the **Forced if Asynchronous** or **Forced Synchronizer Identification** setting) or **Automatic** (with the **Auto** setting). The **Number of Synchronization Registers in Chain** is also reported. This report provides more details about the parameters that

affect the MTBF calculation: the **Available Settling Time** for the chain and the **Data Toggle Rate Used in MTBF Calculation** (for information about the toggle rate, see "Synchronizer Data Toggle Rate in MTBF Calculation" on page 7–9). There is also additional detail to help you identify where this chain is in your design: the **Source Clock** and **Asynchronous Source** node of the signal, the **Synchronization Clock** in the destination clock domain, and the node names of the **Synchronization Registers** in the chain.

Synchronizer Data Toggle Rate in MTBF Calculation

The MTBF calculations assume the data being synchronized is switching at a toggle rate of 12.5% of the source clock frequency. That is, the arriving data is assumed to switch once every eight source clock cycles. If multiple clocks apply, the highest frequency is used. If no source clocks can be determined, the data rate is taken as 12.5% of the synchronization clock frequency.

If you know the approximate rate at which the data changes, and would like to obtain a more accurate MTBF, use the **Synchronizer Toggle Rate** assignment in the Assignment Editor. Set the data toggle rate, in number of transitions per second, on the first register of a synchronization chain. The TimeQuest Timing Analyzer takes the specified rate into account when computing the MTBF of that particular register chain. You can also apply this assignment to an entity or the entire design. Because a **Synchronizer Toggle Rate** assignment of 0 indicates that the data signal never toggles, the affected synchronization chain will not be reported since it does not affect the reliability of the design. To apply the assignment with Tcl, use the following command:

```
set_instance_assignment -name SYNCHRONIZER_TOGGLE_RATE <toggle rate in
transitions/second> -to <register name>
```

There are two other assignments associated with toggle rates, which are not used for metastability MTBF calculations. The I/O Maximum Toggle Rate is only used for pins, and specifies the worst-case toggle rates used for signal integrity purposes. The Power Toggle Rate assignment is used to specify the expected time-averaged toggle rate, and is used by the PowerPlay Power Analyzer to estimate time-averaged power consumption.

MTBF Optimization

In addition to reporting synchronization register chains and MTBF values found in the design, the Quartus II software can also protect these registers from optimizations that might negatively impact MTBF and can optimize the register placement and routing if the MTBF is too low. Synchronization register chains must first be explicitly identified as synchronizers, as described in "Identifying Synchronizers for Metastability Analysis" on page 7–4. Altera recommends that you set **Synchronizer Identification** to **Forced If Asynchronous** for all registers that are part of a synchronizer chain.

Optimization algorithms, such as register duplication and logic retiming in physical synthesis, are not performed on identified synchronization registers. The Fitter protects the number of synchronization registers specified by the **Synchronizer Register Chain Length** option described in the next section.

In addition, the Fitter optimizes identified synchronizers for improved MTBF, by placing and routing the registers to increase the output setup slacks of synchronization registers. Adding slack in the synchronizer chain increases the available settling time for a potentially metastable signal, which improves the chance that the signal will resolve to a known value, and exponentially increases the design MTBF. The Fitter optimizes the number of synchronization registers specified by the **Synchronizer Register Chain Length** option described in the next section.

Metastability optimization is on by default. To view or change the option, on the Assignments menu, click **Settings**. Under **Fitter Settings**, click **More Settings**. From the **More Settings** dialog box, you can turn on or off the **Optimize Design for Metastability** option. To turn the optimization on or off with Tcl, use the following command:

```
set_global_assignment -name OPTIMIZE_FOR_METASTABILITY <ON|OFF>
```

Synchronization Register Chain Length

The **Synchronization Register Chain Length** option specifies how many registers should be protected from optimizations that might reduce MTBF for each register chain, and controls how many registers should be optimized to increase MTBF with the **Optimize Design for Metastability** option. For example, if the **Synchronization Register Chain Length** option is set to **2**, optimizations such as register duplication or logic retiming are prevented from being performed on the first two registers in all identified synchronization chains. The first two registers are also optimized to improve MTBF when the **Optimize Design for Metastability** option is turned on.

The default setting for the **Synchronization Register Chain Length** option is **2**. The first register of a synchronization chain is always protected from operations that might reduce MTBF, but you should set the protection length to protect more of the synchronizer chain. Altera recommends that you set this option to the maximum length of synchronization chains you have in your design so that all synchronization registers are preserved and optimized for MTBF.

To change the global **Synchronization Register Chain Length** option, on the Assignments menu, click **Settings**. Under **Analysis & Synthesis Settings**, click **More Settings**. From the **More Settings** dialog box, you can set the **Synchronization Register Chain Length**.

You can also set the **Synchronization Register Chain Length** on a node or an entity in the Assignment Editor. You can set this value on the first register in a synchronization chain to specify how many registers to protect and optimize in this chain. This individual setting is useful if you want to protect and optimize extra registers that you have created in a specific synchronization chain that has low MTBF, or optimize less registers for MTBF in a specific chain where the maximum frequency or timing performance is not being met.

To make the global setting with Tcl, use the following command:

```
set_global_assignment -name SYNCHRONIZATION_REGISTER_CHAIN_LENGTH
<number of registers>
```

To apply the assignment to a design instance or the first register in a specific chain with Tcl, use the following command:

```
set_instance_assignment -name SYNCHRONIZATION_REGISTER_CHAIN_LENGTH
<number of registers> -to <register or instance name>
```

Reducing Metastability Effects

You can check your design's metastability MTBF in the Metastability Summary report described in "Metastability Report" on page 7–7. As discussed in the "Introduction", you should determine an acceptable target MTBF given the context of your entire system and the fact that MTBF calculations are statistical estimates. A high metastability MTBF (such as hundreds or thousands of years between metastability failures) indicates a more robust design.

This section provides guidelines to ensure complete and accurate metastability analysis, and some suggestions to follow if the Quartus II metastability reports calculate an unacceptable MTBF value. The Timing Optimization Advisor (available from the Tools menu) gives similar suggestions in the Metastability Optimization section.

Apply Complete System-Centric Timing Constraints for the TimeQuest Timing Analyzer

To enable the Quartus II metastability features, make sure that the TimeQuest Timing Analyzer is used for timing analysis.

Ensure that the design is fully timing constrained and that it meets its timing requirements. If the synchronization chain does not meet its timing requirements, the MTBF can not be calculated. If the clock domain constraints are set up incorrectly, the signal transfers between circuitry in unrelated or asynchronous clock domains may not be identified correctly.

Use industry-standard system-centric I/O timing constraints instead of using FPGA-centric timing constraints. As described in "How Timing Constraints Affect Synchronizer Chain Identification and Metastability Analysis" on page 7–6, you should use `set_input_delay` constraints in place of `set_max_delay` constraints to associate each input port with a clock domain to help eliminate false positives during synchronization register identification.

Force the Identification of Synchronization Registers

Use the guidelines in "Identifying Synchronizers for Metastability Analysis" on page 7–4 to ensure the software reports and optimizes the appropriate register chains.

In summary, identify synchronization registers with the **Synchronizer Identification** set to **Forced If Asynchronous** in the Assignment Editor. If there are any registers that the software detects as synchronous but you want to be analyzed for metastability, apply the **Forced** setting to the first synchronizing register. Set **Synchronizer Identification** to **Off** for registers that are not synchronizers for asynchronous signals or unrelated clock domains.

To help you find the synchronizers in your design, you can set the global **Synchronizer Identification** setting on the **TimeQuest Timing Analyzer** page of the **Settings** dialog box to **Auto** to generate a list of all the possible synchronization chains in your design.

Set the Synchronizer Data Toggle Rate

The MTBF calculations assume the data being synchronized is switching at a toggle rate of 12.5% of the source clock frequency. To obtain a more accurate MTBF for a specific chain or all chains in your design, set the **Synchronizer Toggle Rate** as described in "Synchronizer Data Toggle Rate in MTBF Calculation" on page 7–9.

Optimize Metastability During Fitting

Ensure that the **Optimize Design for Metastability** setting described in "MTBF Optimization" on page 7–9 is turned on.

Increase the Length of Synchronizers to Protect and Optimize

Increase the Synchronizer Chain Length parameter to the maximum length of synchronization chains in your design, as described in "Synchronization Register Chain Length" on page 7–10. If you have synchronization chains longer than 2 identified in your design, you can protect the entire synchronization chain from operations that might reduce MTBF and allow metastability optimizations to improve the MTBF.

Set Fitter Effort to Standard Fit instead of Auto Fit

If your design MTBF is too low after following the previous guidelines in this section, you can try increasing the **Fitter effort** to perform more metastability optimization. The default **Auto Fit** setting reduces the Fitter's effort after meeting the design's timing and routing requirements to reduce compilation time. This effort reduction can result in less metastability optimization if the timing requirements are easy to meet. If **Auto Fit** reduces Fitter effort during your design compilation, setting the **Fitter effort** to **Standard Fit** might improve the design's MTBF results. On the Assignments menu, click **Settings**. On the **Fitter Settings** page, set **Fitter effort** to **Standard Fit**.

If Possible, Increase the Number of Stages Used in Synchronizers

Designers commonly use two registers in a synchronization chain to minimize the occurrence of metastable events, and Altera recommends using a standard of three registers for better metastability protection. However, using chains of length two or even three may not be enough to produce a high enough MTBF when the design runs at high clock and data frequencies.

If a synchronization chain is reported to have a low MTBF, consider adding an additional register stage to your synchronization chain. This additional stage increases the settling time of the synchronization chain, allowing more opportunity for the signal to resolve to a known state during a metastable event. Additional settling time increases the MTBF of the chain and improves the robustness of your design. Of course, adding a synchronization stage does introduce an additional stage of latency on the signal.

If you use the Altera FIFO megafunction with separate read and write clocks to cross clock domains, increase the metastability protection (and latency) for better MTBF. In the MegaWizard™ Plug-In Manager for the DCFIFO function, choose the **Best metastability protection, best fmax, unsynchronized clocks** option to add 3 or more synchronization stages. You can increase the number of stages to more than 3 using the **How many sync stages?** setting.

If Possible, Select a Faster Speed Grade Device

The design MTBF depends on process parameters of the device used. Faster devices are less susceptible to metastability issues. If the design MTBF falls significantly below the target MTBF, switching to a faster speed grade can improve the MTBF substantially.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF form.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Identifying Synchronizers for Metastability Analysis

To apply the global Synchronizer Identification assignment described on page "Identifying Synchronizers for Metastability Analysis" on page 7–4, use the following command:

```
set_global_assignment -name SYNCHRONIZER_IDENTIFICATION
<OFF|AUTO|"FORCED IF ASYNCHRONOUS">
```

To apply the **Synchronizer Identification** assignment to a specific register or instance, use the following command:

```
set_instance_assignment -name SYNCHRONIZER_IDENTIFICATION
<AUTO|"FORCED IF ASYNCHRONOUS"|FORCED|OFF> -to <register or instance
name>
```

Synchronizer Data Toggle Rate in MTBF Calculation

To specify a toggle rate for MTBF calculations as described on page "Synchronizer Data Toggle Rate in MTBF Calculation" on page 7–9, use the following command:

```
set_instance_assignment -name SYNCHRONIZER_TOGGLE_RATE <toggle rate in
transitions/second> -to <register name>
```

report_metastability TimeQuest and Tcl Command

If you use a command-line or scripting flow, you can generate the metastability analysis reports described in "Metastability Report" on page 7–7 outside of the Quartus II and TimeQuest user interfaces. Table 7–1 describes the options for the `report_metastability` TimeQuest and Tcl command.

Table 7–1. report_metastabilty Command Options

Option	Description
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-file <name>`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type—either ***.txt** or ***.html**.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.
`-stdout`	Indicates the report be sent to the standard output, via messages. This option is required only if you have selected another output format, such as a file, and would also like to receive messages.

MTBF Optimization

To ensure that metastability optimization described on page "MTBF Optimization" on page 7–9 is turned on (or to turn it off), use the following command:

```
set_global_assignment -name OPTIMIZE_FOR_METASTABILITY <ON|OFF>
```

Synchronization Register Chain Length

To globally set the number of registers in a synchronization chain to be protected and optimized as described on page "Synchronization Register Chain Length" on page 7–10, use the following command:

```
set_global_assignment -name SYNCHRONIZATION_REGISTER_CHAIN_LENGTH
<number of registers>
```

To apply the assignment to a design instance or the first register in a specific chain, use the following command:

```
set_instance_assignment -name SYNCHRONIZATION_REGISTER_CHAIN_LENGTH
<number of registers> -to <register or instance name>
```

Conclusion

Altera's Quartus II software provides industry-leading analysis and optimization features to help you manage metastability in your FPGA designs. Set up your Quartus II project with the appropriate constraints and settings to enable the software to analyze, report, and optimize the design MTBF. Take advantage of these features in the Quartus II software and follow the guidelines in this chapter as required to make your design more robust with respect to metastability.

Referenced Documents

This chapter references the following documents:

- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Reference Manual*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Understanding Metastability in FPGAs* white paper

Document Revision History

Table 7–2 shows the revision history for this chapter.

Table 7–2. Document Revision History

Date and Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Initial release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

8. Best Practices for Incremental Compilation Partitions and Floorplan Assignments

QII51017-9.0.0

Introduction

The Quartus® II incremental compilation feature allows you to partition a design, compile partitions separately, and reuse results for unchanged partitions. It provides the following benefits:

- Reduces compilation times by as much as 70%
- Preserves performance for unchanged design blocks
  - Provides repeatable results and reduces the number of compilations
- Enables true team-based design

This document provides a set of guidelines to help you partition your design to take advantage of Quartus II incremental compilation, and to help you create a design floorplan (using LogicLock™ regions) to support the flow.

This document contains the following sections:

- "Overview: Incremental Compilation"
- "Why Plan Partitions and Floorplan Assignments for Incremental Compilation?" on page 8–4
- "Creating Design Partitions: General Partitioning Guidelines" on page 8–6
- "Creating Design Partitions: Design Guidelines" on page 8–9
- "Creating Design Partitions: Consider Additional Design Suggestions" on page 8–23
- "Checking Partition Quality" on page 8–29
- "Introduction to Design Floorplans" on page 8–35
- "Creating a Design Floorplan: Placement Guidelines" on page 8–38
- "Checking Floorplan Quality" on page 8–43
- "Recommended Design Flows and Application Examples" on page 8–45
- "Potential Issues with Creating Partitions and Floorplan Assignments" on page 8–47

Overview: Incremental Compilation

Quartus II incremental compilation is an optional compilation flow that enhances the default Quartus II compilation. If you do not divide up your design for incremental compilation, your design is compiled using the default "flat" compilation flow. This section provides an overview of the incremental flow, and highlights several best practices.

For details about feature usage and application examples, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

The following procedure outlines the general Quartus II incremental compilation flow:

1. Set up your design hierarchy and source code to support partitioning along logical hierarchy boundaries. If you use a third-party synthesis tool, set up your tool to generate separate netlist files.
2. Create design partition assignments in the Quartus II software to specify which hierarchy blocks are compiled independently as partitions (including empty partitions for any missing or incomplete logic blocks).
3. When the design is compiled, Quartus II Analysis and Synthesis and the Fitter create separate netlists for each partition. These netlists are internal post-synthesis and post-fit database representations of the design.
4. Select which netlist type to preserve for each partition. You can reuse the synthesis or fitting netlist, or instruct the software to resynthesize the source files. You can also import compilation results from another project as part of a bottom-up design flow, as described in "Top-Down versus Bottom-Up Compilation Flows" on page 8–3.
5. After part of the design changes, the software recompiles only the required partitions and merges the new compilation results with existing netlists for other partitions, according to the settings from step 4.

In some cases, as described in "Introduction to Design Floorplans" on page 8–35, you should create a design floorplan with placement assignments to constrain each part of the design to a specific region of the device.

Choosing the Netlist Type and Fitter Preservation Level

You must specify which post-compilation netlist you want to use in subsequent compilations by specifying a Netlist Type setting for each partition. For post-fit netlists, you also specify a Fitter Preservation Level setting to indicate the amount of fitting information you want to preserve. Use the following general guidelines for these standard Netlist Type settings:

- Source File: Use this setting to resynthesize the source code (with any new assignments and replace any previous synthesis or Fitter results)
  - If you modify the design source, the software automatically resynthesizes the appropriate partitions with standard Netlist Type settings, so setting the partition to **Source File** is optional in this case
  - Most assignments do not trigger an automatic recompilation, so setting the partition to **Source File** is optional in this case
- Post-Synthesis (default): Use this setting to re-fit the design (with any new Fitter assignments) but preserve the synthesis results

 © March 2009 Altera Corporation

- Post-Fit: Use this setting to preserve placement and performance results
  - The default setting for post-fit is to preserve placement and reroute the entire design; this usually allows the router to find the best routing for all partitions given their placement on the design, and gives very good performance preservation
- Post-Fit with Fitter Preservation Level set to **Placement and Routing**: Use these settings to reserve routing, only if necessary
  - Use post-fit with routing if necessary to meet the timing requirements for specific partitions

Top-Down versus Bottom-Up Compilation Flows

The Quartus II incremental compilation feature supports both top-down and bottom-up compilation flows, and combinations of the two. The design flow affects how much impact design partitions have on the design optimization.

For more information about the different types of incremental design flows and example applications, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

With top-down compilation, one designer performs placement and routing on the entire design in one Quartus II project, although different designers or IP providers can create and verify HDL code separately, or you can optimize critical design blocks or IP cores before adding the rest of the design. Bottom-up design flows allow individual designers or IP providers to complete the optimization of their design in separate Quartus II projects, and then export the lower-level design partitions for import and integration into the top-level project.

A top-down flow is generally simpler to perform than its bottom-up counterpart. For example, having to export and import lower-level designs is eliminated. In addition, a top-down approach provides the design software with information about the entire design so it can perform some global placement and routing optimizations. Therefore, it is often easier to ensure good quality of results with a top-down flow than with a bottom-up flow.

The Quartus II incremental compilation feature is very flexible and supports numerous design methodologies. You can mix top-down and bottom-up flows within a single project. If the top-level design includes one or more design blocks that are optimized by different designers or IP providers, you can import those blocks (using a bottom-up methodology) into a project that also includes partitions for a top-down incremental methodology. In addition, as you perform timing closure for a design, you can create a subproject for one block of the design to be optimized by another designer in a separate Quartus II project, and pass information about the rest of the design to the subproject to obtain the best results.

By following a mixed design methodology, you can take advantage of the team-based capabilities of a bottom-up flow while maintaining the advantages of a top-down flow for most of the design logic.

Bottom-up incremental compilation is not supported in HardCopy® ASIC migration flows. You cannot use a bottom-up methodology if you want to migrate to a HardCopy ASIC. The Revision Compare feature requires that the HardCopy and FPGA netlists are the same, and all operations performed on one revision must also occur on the other revision. Unfortunately, using the bottom-up flow and importing partitions does not support this requirement.

Generating Bottom-Up Design Partition Scripts for Project Management

If you are using a bottom-up or team-based methodology, you can create design partition scripts to pass top-level constraints (such as floorplan assignments or optimization constraints) to the designers of lower-level blocks.

The bottom-up design partition scripting feature provides a project manager interface for managing resource and timing budgets in the top-level design. This interface makes it easier for designers of lower-level modules to implement the instructions from the project lead, and avoid conflicts between projects when importing and incorporating the projects into the top-level design. Using the scripts also helps reduce the need for further optimization to the designs after integration and improves overall designer productivity and team collaboration.

The feature creates Tcl files that each designer can run to set up a project and makefiles for designers who use a make environment. To use this feature, first set up the top-level project with appropriate constraints and floorplan assignments to be passed to lower levels. Then generate design partition scripts after successful compilation of the top-level design. (You can perform a Fast Synthesis and Early Timing Estimation instead of full compilation to reduce compilation time.) The top-level design can have empty partitions when you generate the scripts. To generate the scripts, on the Project menu, click **Generate Bottom-Up Design Partition Scripts** and set the appropriate options.

For details about using these scripts, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Why Plan Partitions and Floorplan Assignments for Incremental Compilation?

Incremental compilation flows require more up-front planning than flat compilations. For example, you might have to structure your source code or design hierarchy to ensure that logic is grouped correctly for optimization. It is easier to implement the correct logic grouping early in the design cycle than to restructure the code later. Incremental compilation generally requires that you be more rigorous about following good design practices than flat compilations.

Planning involves setting up the design logic for partitioning and may involve planning placement assignments to create a floorplan. Not all design flows require floorplan assignments. If you decide to add floorplan assignments later, when the design is close to completion, well-planned partitions make floorplan creation much easier. Poor partition or floorplan assignments can hurt design area utilization and performance, making timing closure more difficult.

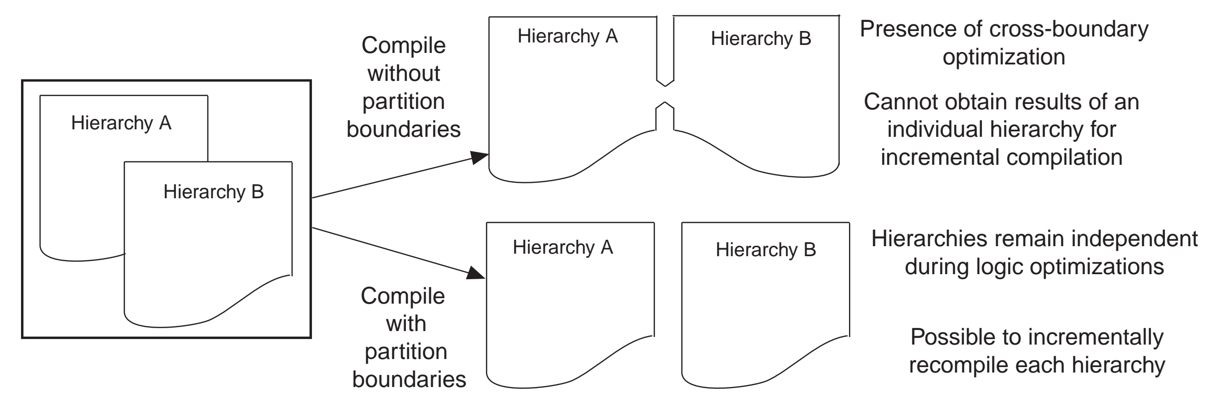
 © March 2009 Altera Corporation

As FPGA devices get larger and more complex, following good design practices becomes more important for all design flows.These planning issues are similar to the requirements for a multiple-chip solution if you were using smaller devices, although planning for one chip is much easier. Adhering to the recommended synchronous design practices makes designs more robust and easier to debug. Using an incremental compilation flow adds additional steps and requirements to your project, but can provide significant benefits in design productivity by preserving the performance of critical blocks and reducing compilation times.

Partition Boundaries and Optimization

If there are any cross-boundary optimizations between partitions, the software cannot obtain separate results for each individual partition. Figure 8–1 describes this effect in more detail. To allow the software to synthesize and place each partition independently, the logical hierarchical boundaries between partitions are treated as hard boundaries for logic optimization. It is important to understand this effect so that you can effectively plan your design partitions.

Figure 8–1. Effects of Partition Boundaries during Logic Optimization



To avoid cross-boundary optimizations, the software synthesizes each partition without using any information about logic contained in other partitions. In a flat compilation, the software uses unconnected signals, constants, inversions, and other design information to perform optimizations. When you partition a design, these types of optimizations do not take place on partition I/O ports. Good design partitions do not rely on these types of logic optimizations.

When all partitions are placed together, the Fitter can perform *placement* optimizations on the design as a whole to optimize the placement of cross-partition paths. (However, the Fitter can never perform any logic optimizations such as physical synthesis across the partition boundary.) When partitions are fit separately in a bottom-up flow or if some partitions use previous post-fitting results, the Fitter does not place and route the entire cross-boundary path at the same time and cannot fully optimize placement across the partition boundaries. Good design partitions can be placed independently because cross-partition paths are not the critical timing paths in the design.

Because cross-boundary logic and placement optimizations cannot occur, the quality of results may decrease as the number of partitions increases. Although more partitions allow for greater reduction in compilation time, you might want to limit the number of partitions to prevent degradation in the quality of results. Creating good design partitions and good floorplan location assignments helps improve the performance results for cross-partition paths. Guidelines for creating these assignments are discussed in the following sections.

Creating Design Partitions: General Partitioning Guidelines

The first stage in planning your design partitions is to organize your source code so that it supports good partition assignments. Although you can assign any hierarchical block of your design as a design partition, following the design guidelines presented in this section ensures better results. This section includes the following topics:

- "Plan Design Hierarchy and Source Design Files" on page 8–6
- "Partition Design by Functionality and Block Size" on page 8–8
- "Partition Design by Clock Domain and Timing Criticality" on page 8–8
- "Consider What Is Changing" on page 8–8

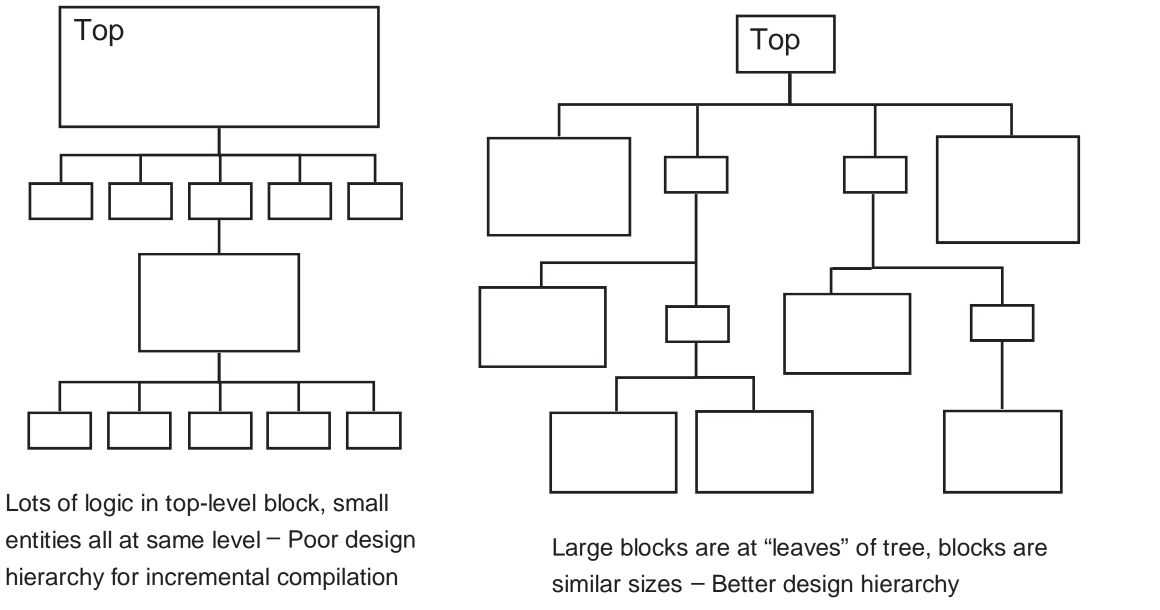
Plan Design Hierarchy and Source Design Files

Start by planning the entities in the design hierarchy. When you assign a hierarchical instance as a design partition, the partition includes the assigned instance and any entities instantiated below it that are not defined as separate partitions. You cannot group separate hierarchical entities into one partition. Take advantage of the design hierarchy to provide flexibility for partitioning and to support different design flows. Keep logic in the "leaves" of the hierarchy tree instead of having a lot of logic at the top level of the design. Doing so ensures that you can isolate partitions if required.

Create entities that can lead to partitions of approximately equal size. For example, do not instantiate a lot of small entities at the same hierarchy level because it will be difficult to group them to form reasonably sized partitions.

Figure 8–2 represents the logic blocks in a design hierarchy. The left side does not follow the recommendations for entity organization, while the right side provides much more flexibility for creating good partitions.

Figure 8–2. Design Hierarchy Recommendations



Create each entity in an independent file. The compiler uses a file checksum to detect changes, and automatically recompiles a partition if its source file changes (for standard Netlist Type settings). If the design entities for two partitions are defined in the same file, changes to the logic in one partition trigger recompilation for both partitions.

Design dependencies also affect which partitions are compiled when a source file changes. If two partitions rely on the same lower-level entity definition, changes in that lower level affect both partitions. Commands such as VHDL `use` and Verilog HDL `` `include `` create dependencies between files, so that changes to one file can trigger recompilations in all dependent files. Avoid these types of file dependencies if they are not required. The **Partition Dependent Files** report for each partition in the **Analysis & Synthesis** folder of the Compilation Report lists which files contribute to each partition.

For more details about what changes trigger an automatic recompilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Using Partitions with Third-Party Synthesis Tools

Incremental compilation works well with third-party synthesis tools in addition to Quartus II integrated synthesis. If you use a third-party synthesis tool, set up your tool to create a separate Verilog Quartus Mapping File (**.vqm**) or EDIF Input File (**.edf**) netlist for each hierarchical partition. In the Quartus II software, assign the top-level entity from each netlist to be a design partition. The **.vqm** or **.edf** netlist file is treated as the source file for the partition in the Quartus II software.

For more details about incremental synthesis in third-party tools, refer to your tool vendor's documentation or the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Partition Design by Functionality and Block Size

Initially, partition your design along functional boundaries. In a top-level system block diagram, each block often is a natural design partition. Typically, each block of a system is relatively independent and has more signal interaction internally than interaction between blocks, which helps reduce optimizations between partition boundaries. Keeping functional blocks together means that synthesis and fitting can optimize related logic as a whole, which can lead to improved optimization.

Consider how many partitions you want to maintain in your design to determine how large each partition should be. How much compilation time reduction you want to achieve is also a factor, because compiling small partitions is typically faster than compiling large partitions.

There is no minimum size for partitions; however, having too many partitions can reduce the quality of results by limiting optimization. Ensure that the design partitions are not too small. As a general guideline, each partition should be more than ~2,000 logic elements (LEs) or adaptive logic modules (ALMs). If your design is not yet complete when you partition the design, use previous designs to help you estimate the size that each block is likely to be.

Partition Design by Clock Domain and Timing Criticality

Consider which clock in your design feeds the logic in each partition. If possible, keep clock domains within one partition. When a clock signal is isolated to one partition, it reduces dependence on other partitions for timing optimization. Isolating a clock domain to one partition also allows better use of regional clock routing networks if the partition logic is going to be constrained to one region of the design. In addition, limiting the number of clocks within each partition simplifies the timing requirements for each partition during optimization. Use an appropriate subsystem to handle any clock domain transfers (such as a synchronization circuit, dual-port RAM, or FIFO). You can include this logic inside the partition at one side of the transfer.

Try to isolate timing-critical logic from logic that you expect to meet its timing requirements easily. Doing so allows you to preserve the satisfactory results for non-critical partitions and focus optimization iterations on just the timing-critical portions of the design to minimize compilation time.

Consider What Is Changing

When assigning partitions, think about what is changing in the design. Is there any intellectual property (IP) or reused logic for which the source code will not change during future design iterations? If so, define the logic in its own partition so that you can compile once and immediately preserve the results, and you will not have to compile that part of the design again. Is some logic being tuned or optimized, or are specifications changing for part of the design? If so, define changing logic in its own partition so that you can recompile only the changing part while the rest of the design stays the same.

As a general rule, create partitions to isolate logic that will change from logic that will not change. Partitioning a design in this way maximizes the preservation of unchanged logic and minimizes compilation time.

Creating Design Partitions: Design Guidelines

Follow the partitioning guidelines presented in this section when creating or modifying the HDL code for each design block that you might want to assign as a design partition. Not all of these recommendations have to be followed exactly to be successful with incremental compilation, but adhering to as many as possible maximizes your chances of success.

This section includes the following topics:

- "Register Partition Inputs and Outputs" on page 8–9
- "Minimize Cross-Partition-Boundary I/O" on page 8–10
- "Avoid the Need for Logic Optimization Across Partitions" on page 8–11

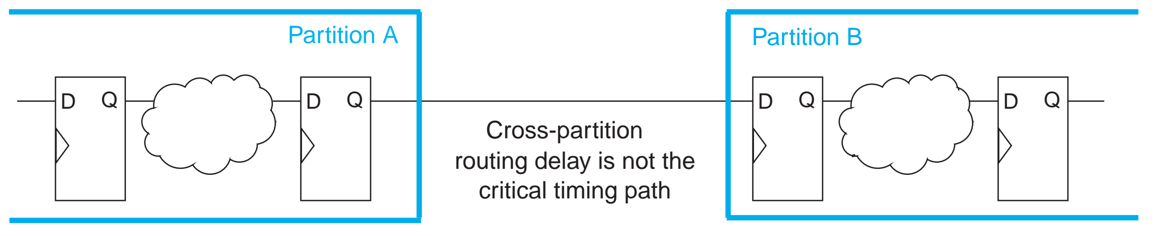
This last subsection includes examples of the types of optimizations that are prevented by partition boundaries, and describes how you can structure or modify your partitions to avoid such optimizations.

Register Partition Inputs and Outputs

Use registers at partition input and output connections that are potentially timing-critical. Registers minimize the delays on inter-partition paths, and prevent the need for cross-boundary logic optimizations.

If every partition boundary has a register as shown in Figure 8–3, every register-to-register timing path between partitions includes only routing delay. Therefore, the timing paths between partitions are likely not timing-critical, and the Fitter can generally place each partition independently from other partitions. This advantage makes it easier to create floorplan location assignments for each separate partition, and is especially important for bottom-up flows in which each partition is placed completely independently. In addition, the partition boundary does not affect combinational logic optimization because each register-to-register logic path is contained within a single partition.

Figure 8–3. Registering Partition I/O



If a design cannot include both input and output registers for each partition due to latency or resource utilization concerns, choose to register one end of each connection. If you register every partition output, for example, the combinational logic that occurs in each cross-partition path is included in one partition so that it can be optimized together.

It is also good synchronous design practice to include registers for every output of a design block. Registered outputs ensure that the input timing performance for each design block is controlled exclusively within the destination logic block.

The statistics described in “Partition Statistics Report” on page 8–33 list how many I/Os are registered or unregistered. The Incremental Compilation Advisor described on page 8–43 lists the unregistered ports for each partition.

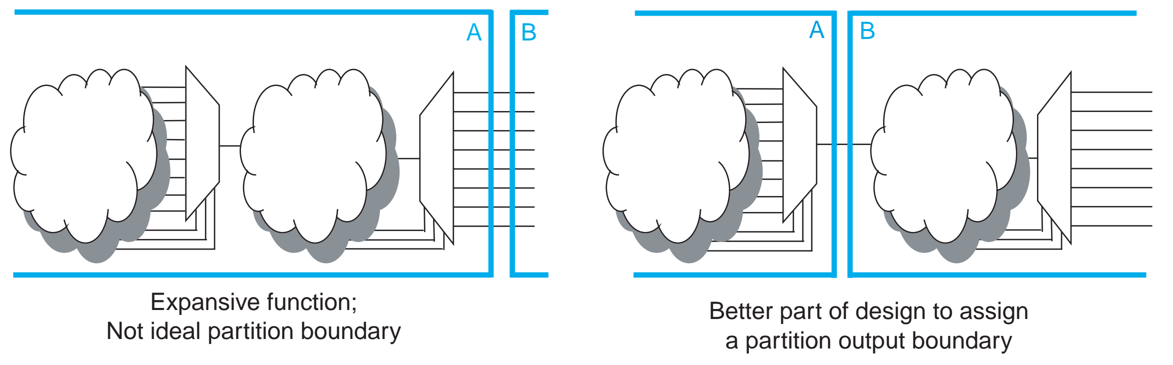
Minimize Cross-Partition-Boundary I/O

Minimize the number of I/O paths that cross between partition boundaries to keep logic paths within a single partition for optimization. Doing so makes partitions more independent for both logic and placement optimization.

This guideline is most important for the timing-critical and high-speed connections between partitions, especially in cases where the input and output of each partition is not registered. Slow connections that are not timing-critical are acceptable because they should not impact the overall timing performance of the design. If there are timing-critical paths between partitions, rework the partitions to avoid these inter-partition paths.

When dividing your design into partitions, consider the types of functions at the partition boundaries. Figure 8–4 shows an expansive function with more outputs than inputs on the left side, which makes a poor partition boundary, and a better place to assign the partition boundary that minimizes cross-partition I/Os on the right side. Adding registers to one or both sides of the cross-partition path in this example would improve the partition quality even more.

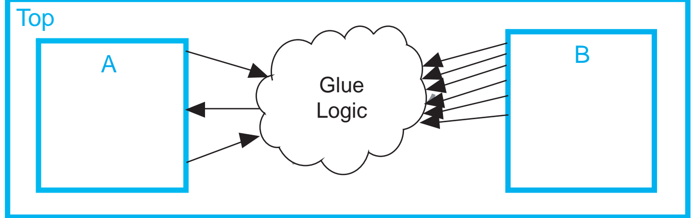
Figure 8–4. Minimizing I/O between Partitions by Moving the Partition Boundary



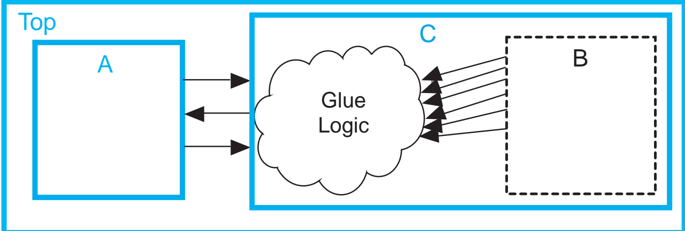
Another way to minimize connections between partitions is to avoid using combinational “glue logic” between partitions. You can often move the logic to the partition at one end of the connection to keep more logic paths within one partition. For example, the bottom diagram in Figure 8–5 includes a new level of hierarchy C that is defined as a partition instead of block B. It is clear that there are fewer I/O connections between partitions A and C than between partitions A and B in the top diagram.

Figure 8–5. Minimizing I/O between Partitions by Modifying Glue Logic

Many cross-partition paths: Poor design partition assignment



Fewer cross-partition paths: Better partitions



The statistics described in "Partition Statistics Report" on page 8–33 list the number of I/O ports as well as the number of inter-partition connections for each partition. The Incremental Compilation Advisor described on "Incremental Compilation Advisor" on page 8–43 lists the number of intra-partition (within a partition) and inter-partition (between partitions) timing edges.

Avoid the Need for Logic Optimization Across Partitions

As discussed in "Partition Boundaries and Optimization" on page 8–5, partition boundaries prevent logic optimizations across partitions. Remember this rule: Logic cannot be optimized or merged across a partition boundary.

To ensure correct and optimal logic optimization, follow the guidelines in this section. In some cases, especially if part of the design is already complete or comes from another designer, these guidelines may not have been followed when the source code was created. These guidelines are not mandatory to implement an incremental compilation flow, but can improve the quality of results. If assigning a partition affects resource utilization or timing performance of a design block as compared to the flat design, it might be due to one of the issues described in this section. Many of the examples provide suggestions for making simple changes to your design or hierarchy to move the partition boundary and improve your results.

These guidelines ensure that your design does not require any logic optimization across partitions:

- "Keep Logic in the Same Partition for Optimization and Merging"
- "Keep Constants in the Same Partition as Logic" on page 8–13
- "Avoid Unconnected Partition I/O" on page 8–14
- "Avoid Signals That Drive Multiple Partition I/O or Connect I/O Together" on page 8–14

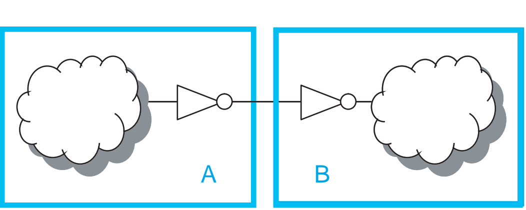
- "Invert Clocks in Destination Partitions" on page 8–16
- "Connect I/O Directly to I/O Register for Packing Across Partition Boundaries" on page 8–16
- "Do Not Use Internal Tri-States" on page 8–20
- "Include All Tri-State and Enable Logic in the Same Partition" on page 8–20
- "Include Bidirectional I/O Registers in the Same Partition" on page 8–21

Keep Logic in the Same Partition for Optimization and Merging

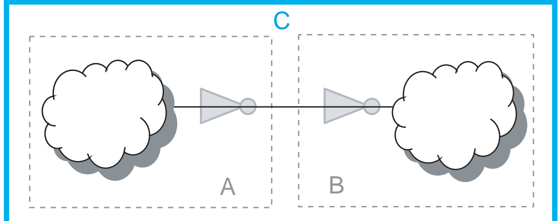
If any design logic requires logic optimization or merging to obtain optimal results, ensure all the logic is part of the same partition.

If a combinational logic path is split across two partitions, the logic cannot be optimized or merged into one logic cell in the device. This effect can result in an extra logic cell in the path, increasing the logic delay. As a very simple example, consider two inverters on the same signal in two different partitions, A and B, as shown in the left side of Figure 8–6. To maintain correct incremental functionality, these two inverters cannot be removed from the design during optimization because they occur in different design partitions. The software cannot use information about other partitions when it compiles each partition, because each partition is allowed to change independently from the other. On the right side of the figure, a new hierarchy block C has been created and defined as a partition to group the logic in blocks A and B instead of having two separate partitions. With the logic contained in one partition, the software can optimize the logic and remove the two inverters (shown in gray color), which reduces the delay for that logic path. Removing two inverters is not a significant reduction in resource utilization because inversion logic is readily available in Altera device architecture; however, it is a good demonstration of the types of logic optimization that are prevented by partition boundaries.

Figure 8–6. Keeping Logic in the Same Partition for Optimization



Inverters in separate partitions A and B cannot be removed from design: Poor design partition assignment



Inverters in same partition C can be removed: Better partition

In a flat design, the Quartus II Fitter can also merge logical instantiations into the same physical device resource. With incremental compilation, logic defined in different partitions cannot be merged to use the same physical device resource.

For example, the Fitter can merge two single-port RAMs from a design into one dedicated RAM block in the device. If the two RAMs are defined in different partitions, the Fitter cannot merge them into one dedicated device RAM block.

This limitation is a concern only if merging is required to fit the design in the target device. Therefore, you are more likely to encounter this issue during troubleshooting than during planning, if your design uses more logic than is available in the device.

Merging PLLs and Transceivers (GXB)

Multiple instances of the ALTPLL megafunction can use the same PLL resource on the device. Similarly, GXB transceiver instances can share high-speed serial interface (HSSI) resources in the same quad as other instances.

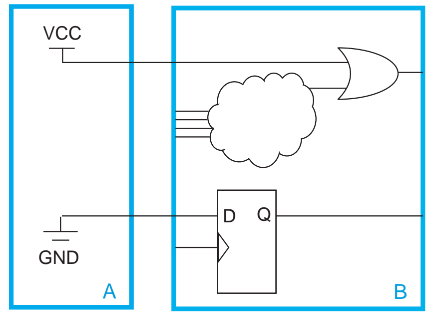
The Fitter can merge multiple instantiations of these blocks into the same device resource, even if it requires optimization across partitions (beginning with the Quartus II software version 7.2). Therefore, there are no restrictions for PLLs and high-speed transceiver blocks when setting up partitions.

Keep Constants in the Same Partition as Logic

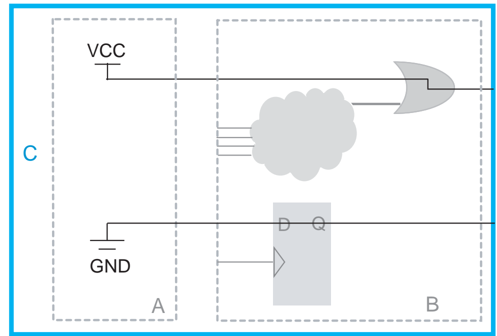
Because the software cannot optimize across a partition boundary, constants are not propagated across partition boundaries. A signal that is constant (1/V_{CC} or 0/GND) in one partition cannot affect another partition.

For example, the left side of Figure 8–7 shows part of a design in which partition A defines some signals as constants (and assumes that the other input connections come from elsewhere in the design and are not shown in the figure). Constants like this could appear due to parameter/generic settings or configurations with parameters, setting a bus to a specific set of values, or could result from optimizations that occur within a group of logic. Because the blocks are independent, the software cannot optimize the logic in block B based on the information from block A. The right side of Figure 8–7 shows a new partition C that groups the logic in blocks A and B, instead of having the two separate partitions. Within the single partition, the software can use the constants to optimize and remove much of the logic in block B (shown in gray color).

Figure 8–7. Keeping Constants in the Same Partition as the Logic They Feed



Connections to constants in another partition:
Poor design partition assignment



Constants in same partition C are used to optimize:
Better partition

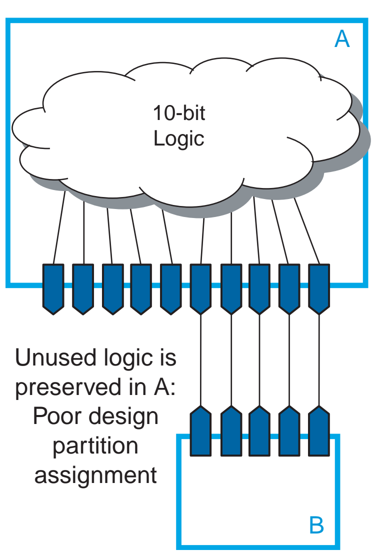
The statistics described in “Partition Statistics Report” on page 8–33 list how many input ports are fed by GND or V_{CC}. The Incremental Compilation Advisor described on page 8–43 lists the ports.

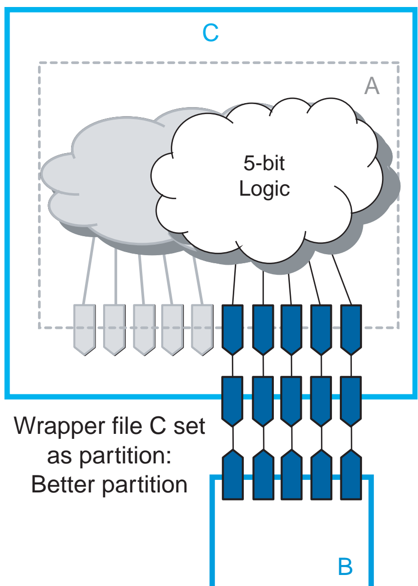
Avoid Unconnected Partition I/O

When a port is left unconnected, optimizations might be able to remove logic driving that port and improve results, similar to a constant connection. However, these optimizations are not allowed across partitions in incremental compilation, because they would create cross-partition dependence. Connect ports to an appropriate node or remove them from the partition. If you know a port will not be used, consider defining a wrapper module with a port interface that reflects this fact.

For example, the left side of Figure 8–8 shows a design that has a 10-bit function defined in partition A, but has only 5 bits connected in partition B. In a flat design, you would expect the logic for the other unused 5 bits to be removed during synthesis. With incremental compilation, synthesis does not remove the unused logic from partition A because partition B is allowed to change independently from partition A. Therefore, you could later connect all 10 bits in partition B and use all 10 bits from partition A. In this case, if you know that you will not use the other 5 bits of partition A, you should remove the unconnected ports and replace them with ground signals inside A. You can create a new wrapper file in the design hierarchy to do this, as shown on the right side of the figure. A new partition C contains the logic from A but includes only the 5 output ports required for connection with partition B. Within this new partition C, the logic for the unused 5 bits can be removed from the design, reducing area utilization.

Figure 8–8. Avoiding Unconnected Partition I/O by Creating a Wrapper File





The statistics described in “Partition Statistics Report” on page 8–33 list how many I/Os are unconnected. The Incremental Compilation Advisor described on page 8–43 lists the unconnected ports.

Avoid Signals That Drive Multiple Partition I/O or Connect I/O Together

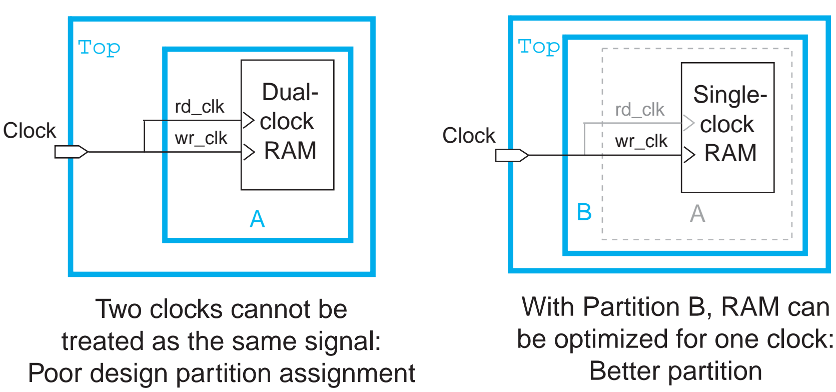
Do not use the same signal to drive multiple ports of a single partition or directly connect two ports of a partition.

If the same signal drives multiple ports of a partition, or if two ports of a partition are directly connected, those ports are logically equivalent. However, because the software has no information about connections made in another partition (including the Top partition), the compilation cannot take advantage of the equivalence. This restriction usually results in sub-optimal results.

If your design has these types of connections, redefine the partition boundaries to remove the affected ports. If one signal from a higher-level partition feeds two input ports of the same partition, feed the one signal into the partition and then make the two connections within the partition. If an output port drives an input port of the same partition, the connection can be made internally without going through any I/O ports. If an input port drives an output port directly, the connection can likely be implemented without the ports in the lower-level partition by connecting the signals in a higher-level design partition.

Figure 8–9 shows an example of one signal driving more than one port. The left diagram shows a design where a single clock signal is used to drive both the read and write clocks of a RAM block. Because the RAM block is compiled as a separate partition A, the RAM block is implemented as though there are two unique clocks. If you know that the port connectivity will not change (that is, the ports will always be driven by the same signal in the Top partition in this case), redefine the port interface so there is only a single port that can drive both connections inside the partition. You can create a wrapper file to define a partition that has fewer ports, as shown in the diagram on the right side. With the single clock fed into the partition, the RAM can be optimized into a single-clock RAM instead of a dual-clock RAM. Single-clock RAM can provide better performance in the device architecture. In addition, partition A might use two global routing lines for the two copies of the clock signal. Partition B can use one global line that fans out to all destinations. Using just the single port connection prevents overuse of global routing resources.

Figure 8–9. Preventing One Signal from Driving Multiple Partition Inputs



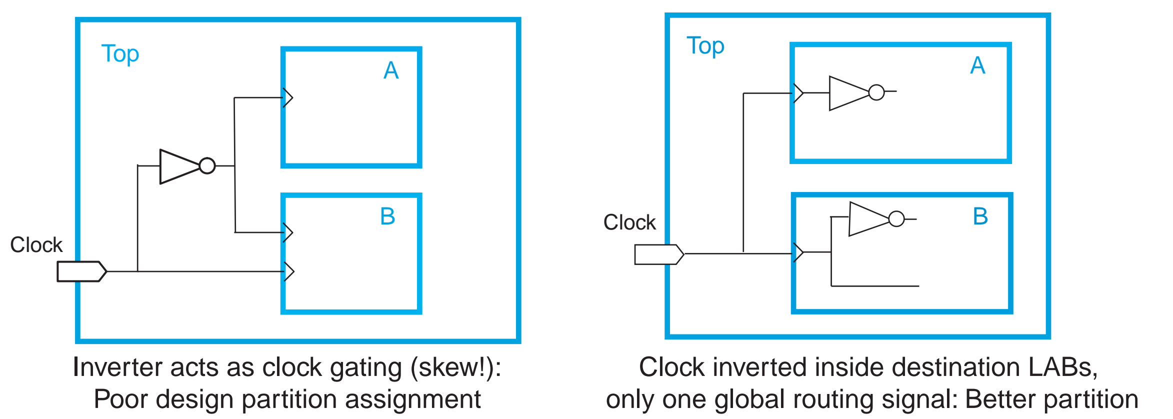
The Incremental Compilation Advisor described on "Incremental Compilation Advisor" on page 8–43 lists partition ports that have the same driving signal, and ports that are directly connected together.

Invert Clocks in Destination Partitions

For best results, clock inversion should be done in the destination logic array block (LAB), because each LAB contains clock inversion circuitry in the device architecture. In a flat compilation, the software can optimize a clock inversion to propagate it to the destination LABs regardless of where the inversion takes place in the design hierarchy. However, clock inversion cannot propagate through a partition boundary to take advantage of the inversion architecture in the destination LABs.

With partition boundaries as shown on the left side of Figure 8–10, the Quartus II software uses logic to invert the signal in the partition that defines the inversion (the Top partition in this example), and then routes the signal on a global clock resource to its destinations (in partitions A and B). The inverted clock acts as a gated clock with high skew. A better solution is to invert the clock signal in the destination partitions as shown on the right side of the figure. In this case the correct logic and routing resources can be used, and the signal is not a gated clock.

Figure 8–10. Inverting Clock Signal in Destination Partitions



Notice that this diagram also shows another example of a single pin feeding two ports of a partition boundary. In the left diagram, partition B does not have the information that the clock and inverted clock come from the same source. In the right diagram, partition B has more information to help optimize the design because the clock is connected as one port of the partition.

Connect I/O Directly to I/O Register for Packing Across Partition Boundaries

Cross-partition register packing of I/O registers is allowed in certain cases where your input and output pins exist in the top-level hierarchy (and the Top partition), but the corresponding I/O registers exist in other partitions.

The following specific circumstances are required for input pin cross-partition register packing:

- The input pin feeds exactly one register.
- The path between the input pin and register includes only input ports of partitions that have one fan-out each.

The following specific circumstances are required for output register cross-partition register packing:

- The register feeds exactly one output pin.

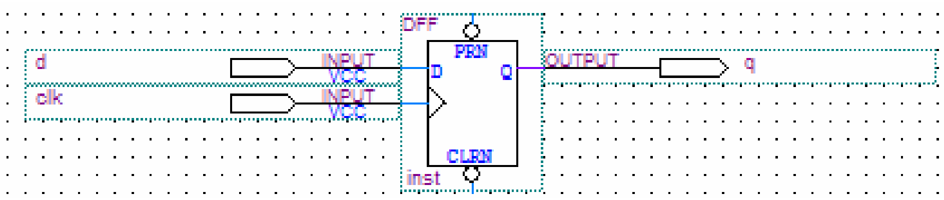
- The output pin is fed by only one signal.
- The path between the register and output pin includes only output ports of partitions that have one fan-out each.

The following examples of I/O register packing illustrate this point using block diagram file (**.bdf**) schematics to describe the design logic.

Example 1—Output Register in Partition Feeding Multiple Output Pins

In this example, a subdesign contains a single register, as shown in Figure 8–11.

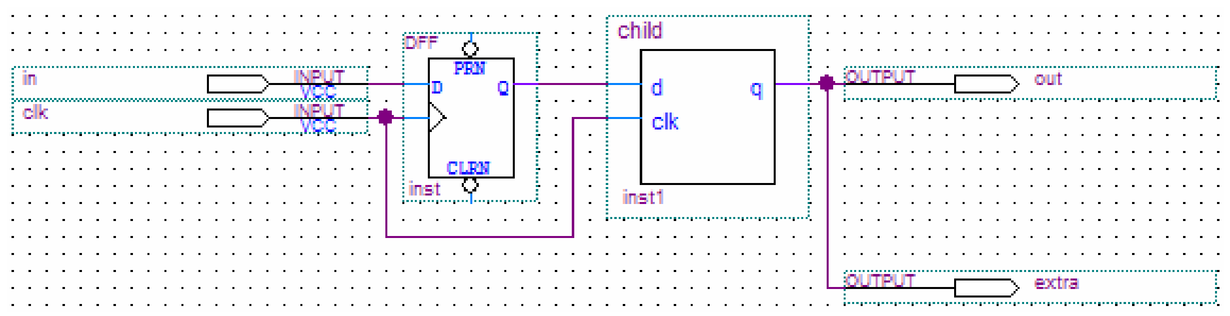
Figure 8–11. Subdesign with One Register, Designated as a Separate Partition



If the top-level design instantiates the subdesign with a single fan-out directly feeding an output pin, and designates the subdesign as a separate design partition, the Quartus II software can perform cross-partition register packing because the single partition port feeds the output pin directly.

In this example, the top-level design instantiates the subdesign as an output register with more than one fan-out signal, as shown in Figure 8–12.

Figure 8–12. Top-Level Design Instantiating the Subdesign in Figure 8–11 with Two Output Pins



In this case, the software does not perform output register packing. If there is a **Fast Output Register** assignment on pin `out`, the software issues a warning that the Fitter cannot pack the node to an I/O pin because the node and the I/O cell are connected across a design partition boundary.

This type of cross-partition register packing is not permitted because it requires modification to the interface of the subdesign partition. To perform incremental compilation, the software must preserve the interface of design partitions.

To allow the software to pack the register in the subdesign from Figure 8–11 with the output pin `out` in Figure 8–12, restructure your HDL code so that output registers directly connects output pins by making one of the following changes:

- Place the register in the same partition as the output pin. The simplest option is to move the register from the subdesign partition into the partition containing the output pin. This guarantees that the Fitter can optimize the two nodes without violating any partition boundaries.
- Duplicate the register in your subdesign HDL as in Figure 8–13 so that each register feeds only one pin, then connect the extra output pin to the new port in the top-level design as shown in Figure 8–14. This converts the cross-partition register packing into the simplest case where each register has a single fan-out.

Figure 8–13. Modified Subdesign from Figure 8–11 with Two Output Registers and Two Output Ports

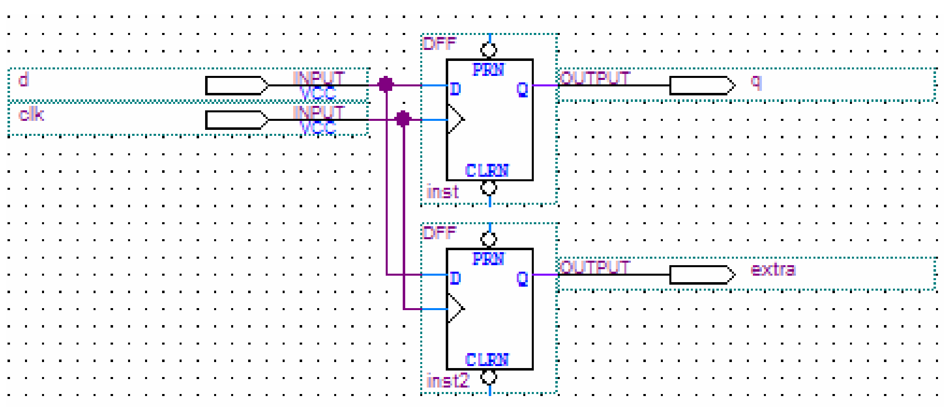
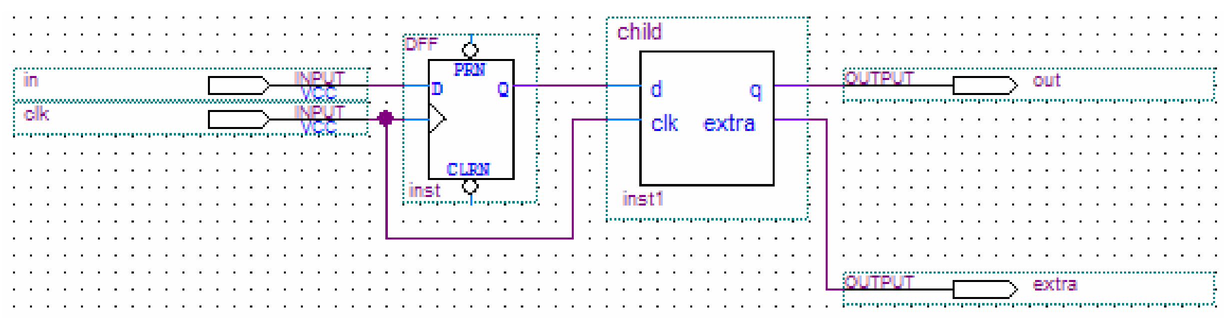


Figure 8–14. Modified Top-Level Design from Figure 8–12 Connecting Two Output Ports to Output Pins



Example 2—Input Register in Partition Fed by an Inverted Input Pin or Output Register in Partition Feeding an Inverted Output Pin

In this example, a subdesign designated as a separate partition contains a register, as shown in Figure 8–11. The top-level design in Figure 8–15 instantiates the subdesign as an input register with the input pin inverted. The top-level design in Figure 8–16 instantiates the subdesign as an output register with the signal inverted before feeding an output pin.

Figure 8–15. Top-Level Design Instantiating the Subdesign in Figure 8–11 as an Input Register with an Inverted Input Pin

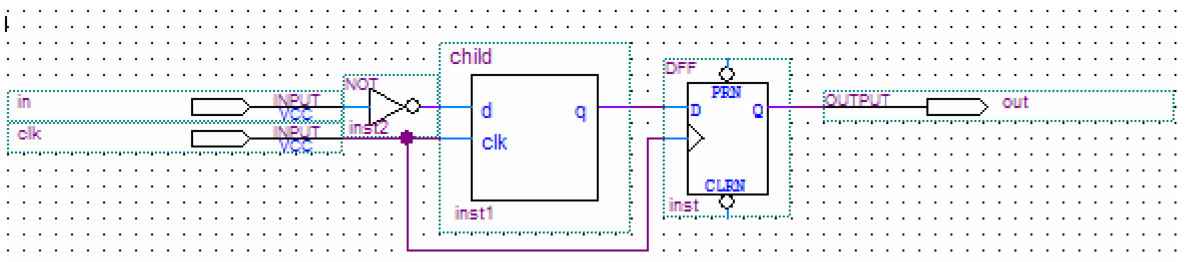
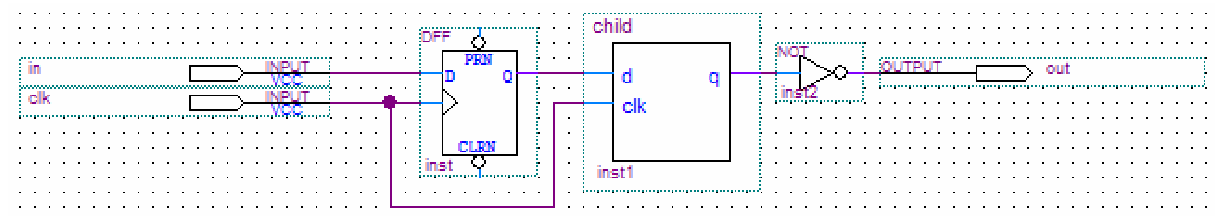


Figure 8–16. Top-Level Design Instantiating the Subdesign in Figure 8–11 as an Output Register Feeding an Inverted Output Pin



In these cases, the software does not perform register packing. If there is a **Fast Input Register** assignment on pin `in` in Figure 8–15 or a **Fast Output Register** assignment on pin `out` in Figure 8–16, the software issues a warning that the Fitter cannot pack the node to an I/O pin because the node and I/O cell are connected across a design partition boundary.

This type of register packing is not permitted because it requires moving logic across a design partition boundary to place into a single I/O device atom. To perform register packing, either the register must be moved out of the subdesign partition or the inverter must be moved into the subdesign partition to be implemented in the register.

To allow the software to pack the register in the subdesign from Figure 8–11 with the input pin `in` in Figure 8–15 or the output pin `out` in Figure 8–16, restructure your HDL code to place the register in the same partition as the inverter by making one of the following changes:

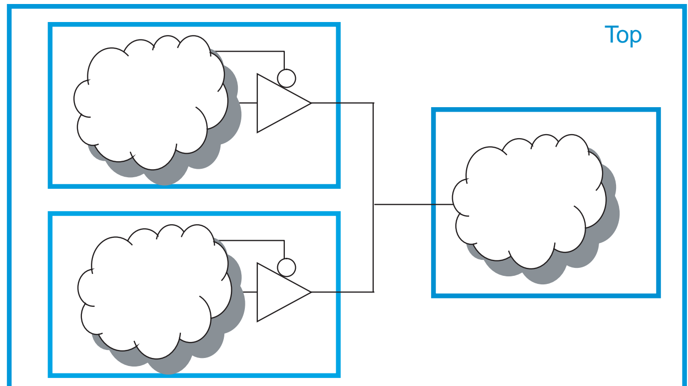
- Move the register from the subdesign partition into the top-level partition containing the pin. This ensures that the Fitter can optimize the I/O register and inverter without violating any partition boundaries.
- Move the inverter from the top-level block into the subdesign, then connect the subdesign directly to a pin in the top-level design. This allows the Fitter to optimize the inverter into the register implementation, so the register is directly connected to a pin, which enables register packing.

Do Not Use Internal Tri-States

Internal tri-state signals are not recommended for FPGAs because the device architecture does not include internal tri-state logic. If designs do use internal tri-states in a flat design (with no partitions), the tri-state logic is typically converted to OR gates or multiplexing logic. But if tri-state logic occurs on a hierarchical partition boundary, the software cannot convert the logic to combinational gates because the partition could be connected to a top-level device I/O through another partition.

Figure 8–17 shows a design with partitions that are not supported for incremental compilation due to the internal tri-state output logic on the partition boundaries. Instead of using internal tri-state logic for partition outputs, implement the correct logic to select between the two signals. Doing so is good practice even when there are no partitions, because such logic explicitly defines the behavior for the internal signals instead of relying on the software to convert the tri-state signals into logic.

Figure 8–17. Unsupported Internal Tri-State Signals



Design results in Quartus II error message:
The software cannot synthesize this
design and maintain incremental functionality

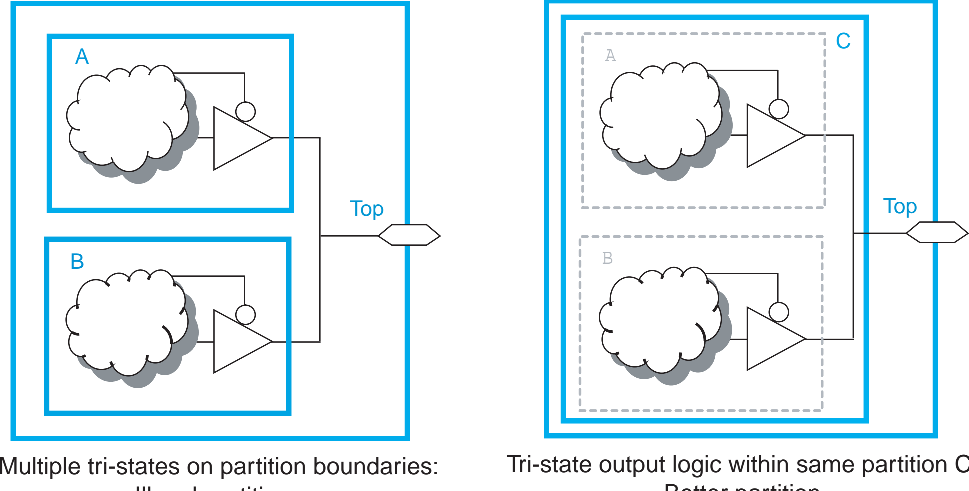
Do not use tri-state signals or bidirectional ports on hierarchical partition boundaries, unless the port is connected directly to a top-level I/O pin on the device. If you must use internal tri-state logic, ensure that all the control and destination logic is contained in the same partition, in which case the software can convert the internal tri-state signals into multiplexing logic like in a flat design. If possible, you should avoid using internal tri-state logic in any Altera FPGA design to ensure that you get the desired implementation when the design is compiled for the target device architecture.

Include All Tri-State and Enable Logic in the Same Partition

When multiple output signals use tri-state logic to drive a device output pin, the Quartus II software merges the logic into one tri-state output pin. The software cannot merge tri-state outputs into one output pin if any of the tri-state logic occurs on a partition boundary. Similarly, output pins with an output enable signal cannot be packed into the device I/O cell if the output enable logic is part of a different partition from the output register. To allow register packing for output pins with an output enable signal, structure your HDL code or design partition assignments so that the register and enable logic are defined in the same partition.

Figure 8–18 shows a design with tri-state output signals that feed a device bidirectional I/O pin (assuming that the input connection feeds elsewhere in the design and is not shown in the figure). On the left side of the figure, the tri-state output signals appear as the outputs of two separate partitions. In this case, the software cannot implement the specified logic and maintain incremental functionality. On the right side, another level of hierarchy C has been created to group the logic from blocks A and B. With this single partition C, the Quartus II software can merge the two tri-state output signals and implement them in the tri-state logic available in the device I/O element.

Figure 8–18. Including All Tri-State Output Logic in the Same Partition



Include Bidirectional I/O Registers in the Same Partition

For a bidirectional partition port that feeds a bidirectional I/O pin at the top level, all the logic that forms the bidirectional I/O cell must reside in the same partition. This guideline applies to the Stratix II family, Cyclone® II family, and all older Altera device families that include I/O registers. In addition, as discussed in the previous two recommendations, the I/O logic must feed the I/O pin without any intervening logic.

In Figure 8–19, for the software to implement all three registers in the I/O element along with the tri-state logic, all the I/O logic must be defined inside the same partition. The logic connected to the registers can occur in the same partition or any other partition; only the I/O registers must be grouped with the tri-state logic definition. The bidirectional I/O port of the partition must be directly connected to the bidirectional device pin at the top level. The signal can go through several partition boundaries if necessary, as long as the connection path contains no logic.

Figure 8–19. Including All Bidirectional I/O Registers in the Same Partition

Bidirectional logic is within one partition, and I/O logic directly feeds I/O pin

Summary of Guidelines Related to Logic Optimization Across Partitions

Follow the guidelines presented in this section to ensure that your design does not require any logic optimization across partitions:

- Keep logic in the same partition for optimization and merging
- Keep constants in the same partition as logic
- Avoid unconnected partition I/O
- Avoid signals that drive multiple partition I/O or connect I/O together
- Invert clocks in destination partitions
- Connect I/O directly to I/O register for packing across partition boundaries
- Do not use internal tri-states
- Include all tri-state and enable logic in the same partition
- Include bidirectional I/O registers in the same partition

Remember that these guidelines are not strict rules to implement an incremental compilation flow, but can improve the quality of results. When creating source design code, keep these guidelines in mind and organize your HDL code to support good partition boundaries. For designs that are already complete, assess whether assigning a partition affects the resource utilization or timing performance of a design block as compared to the flat design, and make the appropriate changes to your design or hierarchy to improve your results.

Creating Design Partitions: Consider Additional Design Suggestions

This section includes several additional design practices that may improve success in incremental compilation flows, if they are applicable to your design:

- "Balance Logic Resources" on page 8–23
- "Balance Global Routing Signals and Clock Networks if Required" on page 8–24
- "Assign Virtual Pins in Bottom-Up Flows" on page 8–25
- "Perform Timing Budgeting if Required" on page 8–26
- "Consider a Cascaded Reset Structure" on page 8–26
- "Drive Clocks Directly in Bottom-Up Flows" on page 8–28
- "Recreate PLLs for Lower-Level Partitions if Required in Bottom-Up Flows" on page 8–28

Balance Logic Resources

When using incremental compilation, the software synthesizes each partition separately with no data about the resources used in other partitions. This means that device resources could be overused in the individual partitions during synthesis, thus, the design may not fit in the target device when the partitions are merged.

In a bottom-up design flow in which designers optimize their lower-level designs and export them to a top-level design, the software places and routes each partition separately. In some cases, partitions can use conflicting resources when combined at the top level.

For example, in the standard synthesis flow, the Quartus II Compiler can perform automated resource balancing for DSP blocks or RAM blocks and convert some of the logic into regular logic cells to prevent overuse. Without data about DSP and RAM blocks used in other partitions, it is possible for the logic in each separate partition to maximize the use of a particular device resource.

To avoid these effects, you may have to perform manual resource balancing across partitions. This is more applicable to bottom-up design flows, because top-down compilation usually handles resource balancing without any user intervention.

To prevent overuse of device resources such as DSP or RAM blocks, you may be able to manually balance the resources. You can use the Quartus II synthesis options to control inference of megafunctions that use the DSP or RAM blocks. You can also use the MegaWizard™ Plug-In Manager to customize your RAM or DSP megafunctions to use regular logic instead of the dedicated hardware blocks.

To assign a number of DSP or RAM resources for each partition, use the following logic options to specify the maximum number of dedicated logic blocks that the software can use in the specified partition: **Maximum DSP Block Usage**, **Maximum Number of M4K/M9K Memory Blocks**, or **Maximum Number of M-RAM/M144K Memory Blocks**. You can set these options globally for all partitions. To set an option for all partitions, on the Assignments menu, click **Settings**. Under **Category**, select **Analysis & Synthesis Settings**. Click **More Settings,** and in the **Existing option settings** list, select the appropriate option. You can also set the option for a specific partition using the Assignment Editor. Select the assignment name, apply it to the root

entity of a partition, and set an integer as the value. The partition-specific assignment overrides the global assignment, if any. However, *each* partition that does not have a partition-specific assignment can use the number of DSP or RAM blocks set by the global assignment. Be aware that this behavior can lead to over-allocation of dedicated logic blocks, eventually resulting in a no-fit error.

For more information about resource balancing when using Quartus II synthesis, refer to the "Megafunction Inference Control" section in the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For more tips about resource balancing and reducing resource utilization, refer to the appropriate "Resource Utilization Optimization Techniques" section in the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

It is often helpful to create a LogicLock region to isolate the placement of each partition, especially in bottom-up flows, to minimize the chance that the logic in more than one partition uses the same logic resource. However, there are situations in which partition placement may still cause conflicts at the top level. For example, you can design a partition one way in a lower-level design (such as using an M-RAM memory block) and then instantiate it in two different ways in the top level (such as one using an M-RAM block and another using an M4K block). In this case, you can export a post-fit netlist with no placement information from the lower-level design and allow the software to refit the logic at the top level.

Balance Global Routing Signals and Clock Networks if Required

If your design is very complex and has many clocks, you may have to allocate global routing resources between the different design partitions. In most cases, you do not have to allocate routing because the software finds the best solution for the global signals.

Global routing signals can cause conflicts when multiple projects are imported into a top-level design. The Quartus II software automatically promotes high fan-out signals to use global routing resources available in the device. Lower-level partitions can use the same global routing resources, thus causing conflicts at the top level. In addition, LAB placement depends on whether the inputs to the logic cells within the LAB are using a global clock signal. Therefore, problems can occur if a design does not use a global signal in the lower-level design, but does use a global signal in the top-level design.

To avoid these problems, the project lead can first determine which partitions use which type of global routing signals. Each designer of a lower-level partition can then assign the appropriate type of global signals manually and prevent other signals from using global routing resources, or set a maximum number of clocks for the partition.

You can use the **Global Signal** assignment to force or prevent the use of a global routing line, making the assignment to a clock source node or signal. You can also assign certain types of global clock resources in some device families, such as regional clocks that cover only part of the device. Alternatively, designers of lower-level partitions can specify the number of clocks allowed in the project using the maximum clocks allowed options. On the Assignments menu, click **Settings**. Under **Category**,

select **Fitter Settings**. Click **More Settings**, and in the **Existing option settings** list, select the appropriate option. Choose **Maximum number of clocks of any type allowed**, or use the **Maximum number of global clocks allowed**, **Maximum number of regional clocks allowed**, and **Maximum number of periphery clocks allowed** options to restrict the number of clock resources of a given type in the project.

You can view the resource coverage of regional clocks in the Chip Planner, and then align LogicLock regions that constrain partition placement with available global clock routing resources. For example, if the LogicLock region for a particular partition is limited to one device quadrant, that partition's clock can use a regional clock routing type that covers only one device quadrant. If all partition logic is available, the project lead can compile the entire design at the top level with floorplan assignments to allow the use of regional clocks that span only a part of the chip. You can use the Fitter's results to make assignments when optimizing the lower-level partitions in separate Quartus II projects.

If you require more control when planning a design with imported partitions, you can assign a specific signal to use a particular clock network in Stratix II devices and newer device families by assigning the clock control block instance called CLKCTRL. Use the **Global Clock CLKCTRL Location** logic option. You can make a point-to-point assignment from a clock source node to a destination node, or a single-point assignment to a clock source node. Set the assignment value to the name of the clock control block: `CLKCTRL_G`<*global network number*> to choose one of the global routing networks or `CLKCTRL_R`<*regional network number*> to choose one of the dedicated regional routing networks in the device.

If you want to disable the automatic global promotion performed in the Fitter to prevent other signals from being placed on global (or regional) routing networks, turn off the **Auto Global Clock** and **Auto Global Register Control Signals** options. On the Assignments menu, click **Settings**. On the **Fitter Settings** page, click **More Settings** and change the settings to **Off**.

If you are performing a bottom-up flow using design partition scripts, the software can automatically write the commands to pass global constraints and turn off the automatic options. For more information, refer to "Generating Bottom-Up Design Partition Scripts for Project Management" on page 8–4.

Alternatively, to avoid problems when importing, direct the Fitter to discard the placement and routing of the imported netlist by setting the Fitter preservation level property of the partition to Netlist Only. With this option, the Fitter reassigns all the global signals for this particular partition when compiling the top-level design.

Assign Virtual Pins in Bottom-Up Flows

Virtual pins map lower-level design I/Os to internal cells. Use them when the number of I/Os on a lower-level design exceeds the device I/O count, and to increase the timing accuracy of cross-partition paths.

Make a **Virtual Pin** assignment in the Assignment Editor for lower-level design I/Os that will become internal nodes in the top level. Leave clock pins mapped to I/O pins to ensure proper routing.

You can specify locations for the virtual pins that correspond to the placement of other partitions. You can also make timing assignments to the virtual pins to define a timing budget, as described in the following section.

Virtual pins are created automatically from the top-level design if you use the **Generate Bottom-Up Design Partition Scripts** command. The scripts place the virtual pins to correspond with other partitions' placement from the top-level design. Refer to "Generating Bottom-Up Design Partition Scripts for Project Management" on page 8–4 for details.

Perform Timing Budgeting if Required

If you optimize lower-level partitions independently and import them to the top level, or compile with empty partitions, any unregistered paths that cross between partitions are not optimized as an entire path. In these cases, the Compiler has no information about the placement of the logic that connects to the I/O ports. If the logic in one partition is placed far away from logic in another partition, the routing delay between the logic can lead to problems in meeting the timing requirements. You can reduce this effect by ensuring that input and output ports of the partitions are registered whenever possible.

To ensure that the Compiler correctly optimizes the input and output logic in each partition, you may be required to perform some manual timing budgeting. For each unregistered timing path that crosses between partitions, make timing assignments on the corresponding I/O path in each partition to constrain both ends of the path to the budgeted timing delay. Assigning a timing budget for each part of the connection ensures that the Compiler optimizes the paths appropriately.

When performing manual timing budgeting in a lower-level partition for I/O ports that become internal partition connections in a top-level design, you can assign location and/or timing constraints to the virtual pin that represents each connection to further improve the quality of the timing budget. Refer to the previous section for a description of virtual pins.

If you are performing a bottom-up flow using the design partition scripts, the software can write I/O timing budget constraints automatically for virtual pins. Refer to "Generating Bottom-Up Design Partition Scripts for Project Management" on page 8–4 for details.

Consider a Cascaded Reset Structure

Designs typically have a global asynchronous reset signal where a top-level signal feeds all partitions. To minimize skew for the high fan-out signal, the global reset signal is typically placed onto a global routing resource.

In some cases, having one global reset signal can lead to recovery and removal time problems. This issue is not specific to incremental flows; it could be applicable in any large high-speed design. For incremental flows, the global reset signal also creates a timing dependency between the Top partition and lower-level partitions.

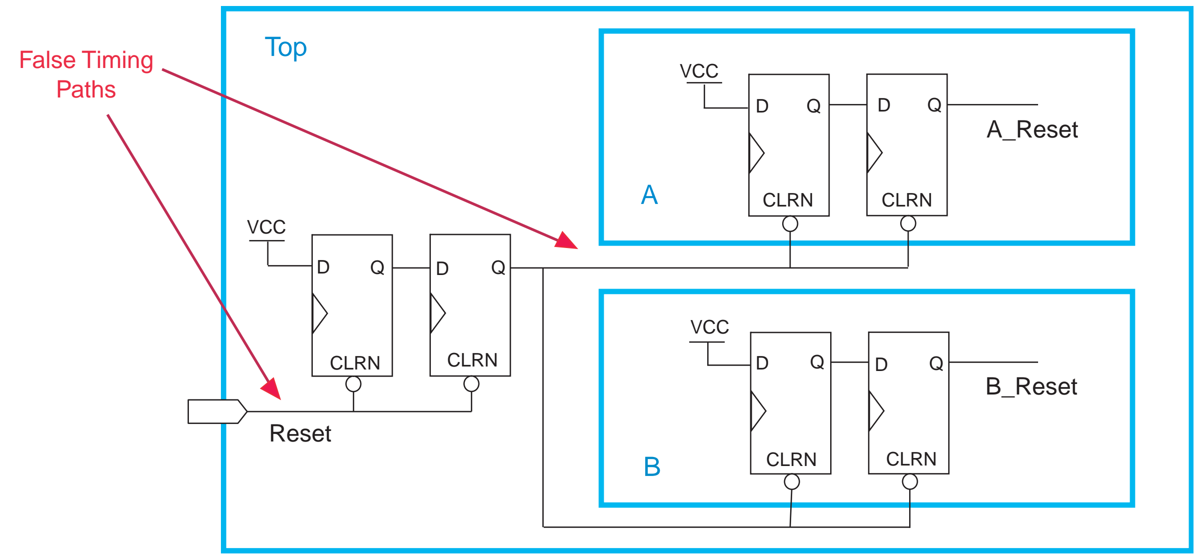
For incremental compilation, minimizing the impact of global structures is helpful. To isolate each partition, consider adding reset synchronizers. By using cascaded reset structures, the design intent is to reduce the inter-partition fan-out of the reset signal, thereby minimizing the effect of the global signal. Reducing the fan-out of the global reset signal also provides more flexibility in routing the cascaded signals, and may help recovery and removal times in some cases.

This suggestion can help in large designs, regardless of whether you are using incremental compilation. However, if one global signal can feed all the logic in its domain and meet recovery and removal times, you probably do not have to follow this recommendation. It is more relevant for high-performance designs where meeting timing on the reset logic can be challenging. Isolating each partition and allowing more flexibility in global routing structures is an additional advantage in incremental flows.

If you add additional reset synchronizers to your design, it adds latency to the reset path, so be sure that this is acceptable in your design. In addition, parts of the design may come out of reset in different clock cycles. You can balance the latency or add hand-shaking logic between partitions, if necessary, to accommodate these differences.

Figure 8–20 shows a cascaded reset structure. The signal is first synchronized as it comes on the chip, following good synchronous design practices. This logic means the design asynchronously resets, but synchronously releases from reset to avoid any race conditions or metastability problems. Then, to minimize the impact of global structures, the circuit employs a divide-and-conquer approach for the reset structure. By implementing a cascaded reset structure, each partition's reset paths are independent. This reduces the effect of inter-partition dependency because the inter-partition reset signals can now be treated as false paths for timing analysis. In some cases, the partition's reset signal can be placed on local lines to reduce the delay added by routing to a global routing line. In other cases, the signal can be routed on a regional or quadrant clock signal.

Figure 8–20. Cascaded Reset Structure



This circuit as one that may help you achieve timing closure and partition independence for your global reset signal. Evaluate the circuit and consider how it works for your design.

Drive Clocks Directly in Bottom-Up Flows

In bottom-up flows, the drive partition clock inputs directly with device clock input pins.

Connecting the clock signal directly avoids any timing analysis difficulties with gated clocks. Clock gating is never recommended for FPGA designs because of potential glitches and clock skew. Clock gating can cause trouble especially in bottom-up flows because the lower-level partitions have no information about any gating that takes place at the top level or in another partition. If a gated clock is required in a partition, perform the gating within that partition, as described for clock inversion in "Invert Clocks in Destination Partitions" on page 8–16.

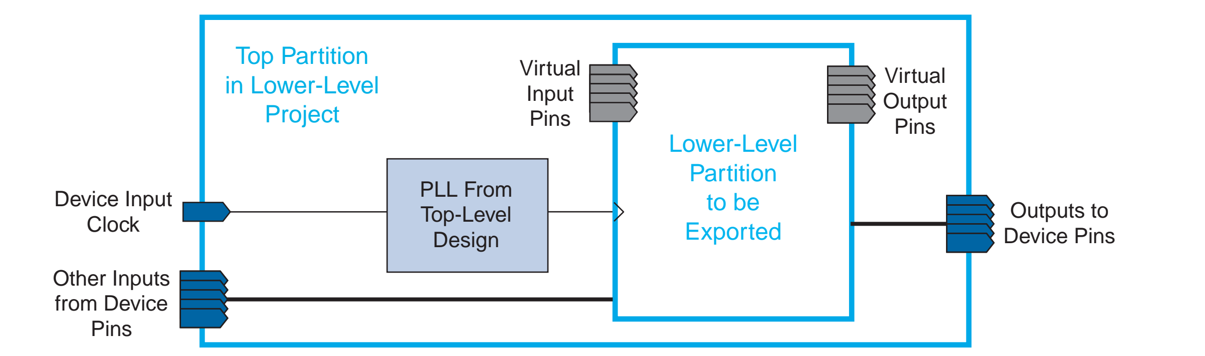
Direct connections to input clock pins also allows design partition scripts to send constraints from the top-level device pin to the lower-level partitions.

Recreate PLLs for Lower-Level Partitions if Required in Bottom-Up Flows

If you use a PLL in your top-level design and connect it to lower-level partitions, the lower-level partitions do not have information about the multiplication, phase shift, or compensation delays for the PLL. To accommodate the PLL timing, you can make appropriate timing assignments in your lower-level Quartus II project to ensure that clocks are not left unconstrained or constrained with an incorrect frequency. Alternatively, you can manually duplicate the top-level PLL (or other derived clock logic) in the lower-level design file to ensure that you have the correct PLL parameters and clock delays for complete, accurate timing analysis.

Include a copy of the top-level PLL in your lower-level project as shown in Figure 8–21, and create a design partition for the rest of the lower-level design logic that will be exported to the top level. When the design is complete, you can export just the lower-level partition, without exporting any auxiliary PLL components to the top-level design. When you use the feature to export a partition within a project, the software exports any hierarchy under the specified partition into the Quartus II Exported Partition (**.qxp**) file but does not include logic defined outside the partition (the PLL in this example).

Figure 8–21. Recreating a Top-Level PLL in a Lower-Level Partition



Checking Partition Quality

This section provides an overview of tools you can use as you make partitions in the Quartus II software. Take advantage of these tools to assess your partition quality, and use the information to improve your design or assignments as required to achieve the best results.

Incremental Compilation Advisor

You can use the Incremental Compilation Advisor to check that your design follows the recommendations for creating design partitions that are presented in this document.

On the Tools menu, point to **Advisors** and click **Incremental Compilation Advisor**. Recommendations are split into General Recommendations that apply to all compilation flows and Bottom-Up Design Recommendations that apply to bottom-up design methodologies. Each recommendation provides an explanation, describes the effect of the recommendation, and provides the action required to make the suggested change.

To check whether the design follows the recommendations, go to the **Timing Independent Recommendations** page or the **Timing Dependent Recommendations** page (for the TimeQuest Timing Analyzer or the Classic Timing Analyzer), and click **Check Recommendations**. For large designs, these operations can take a few minutes.

After you check the design, a symbol appears next to each recommendation that indicates whether or not your design follows that particular recommendation. Refer to the Legend on the **How to use the Incremental Compilation Advisor** page in the Incremental Compilation Advisor for more information.

In some items, there is a link to the appropriate Quartus II settings page where you can make a suggested change to assignments or settings. For many items, if your design does not follow the recommendation, the Check Recommendations operation creates a table that lists any nodes or paths in the design that could be improved.

For example, if not all of the partition I/O ports follow the Register All Ports recommendation, the Incremental Compilation Advisor displays a list of unregistered ports with the partition name and the source and destination nodes for the port. When the Incremental Compilation Advisor provides a list of nodes, you can right-click on a node and click **Locate** to cross-probe to other Quartus II features such as the RTL Viewer, Chip Planner, or the design source code in the text editor.

Opening a new TimeQuest report resets the Incremental Compilation Advisor results, so you must rerun the Check Recommendations process.

Design Partition Planner

The Design Partition Planner allows you to view design connectivity and hierarchy, and can assist you in creating effective design partitions that follow the guidelines in this document. You can also use the Design Partition Planner to optimize design performance, by isolating and resolving failing paths on a partition-by-partition basis.

To view a design and create design partitions, first compile the design, or perform Analysis and Synthesis. On the Tools menu, click **Design Partition Planner**. The design is displayed as a single top-level design block, containing its lower-level instances as boxes.

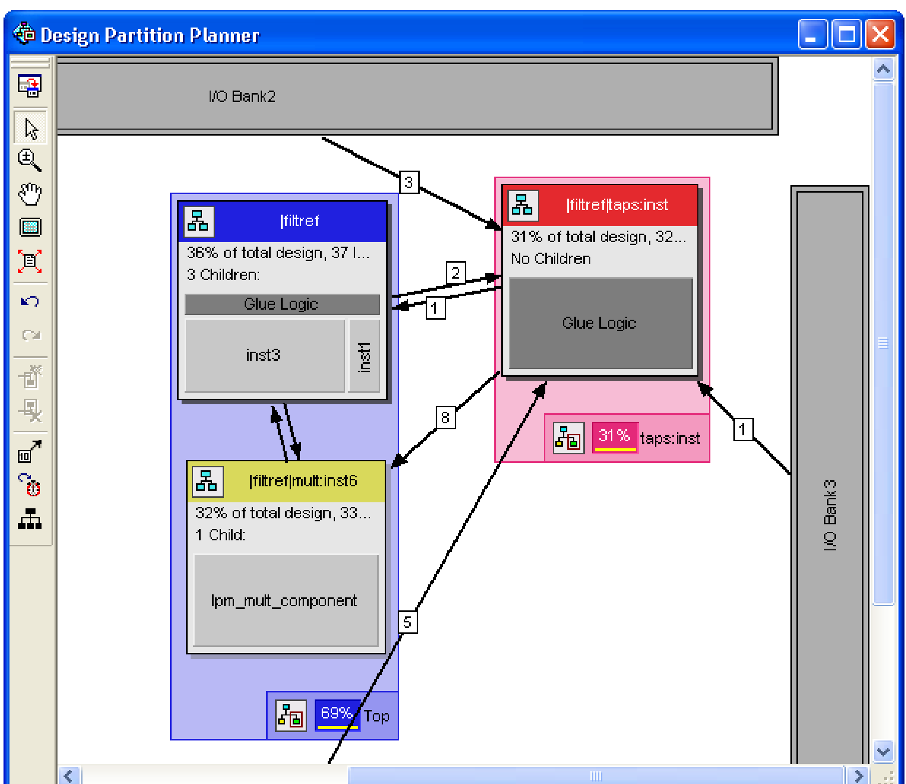
To show connectivity between blocks, extract instances from the top-level design block. Click on a design block and drag it into the surrounding white space, or right-click an entity and click **Extract from Parent** on the Shortcut menu.

When you extract entities, connection bundles are drawn between entities, showing the number of connections existing between pairs of entities. When you have extracted a design block that you want to set as a design partition, right-click on that design block and click **Create Design Partition**.

The Design Partition Planner also has an Auto-Partition feature that creates partitions based on the size and connectivity of the hierarchical design blocks. Right-click on the design block you want to partition (such as the top-level design hierarchy), and choose **Auto-Partition**. You can then analyze and adjust the partition assignments as required.

Figure 8–22 shows the Design Partition Planner after making a design partition assignment to one instance (in the pale red shaded box), and dragging another instance away from the top-level block within the same partition (two design blocks in the pale blue shaded box). The figure shows the number of connections between each partition and information about the size of each design instance.

Figure 8–22. Design Partition Planner



To switch between connectivity display mode and hierarchical display mode, click **Hierarchy Display** on the View menu. Alternately, to switch temporarily to a view-only hierarchy display, click and hold the hierarchy icon in the top-left corner of any entity.

To control the way the connection bundles are displayed, right-click in the white space and choose **Bundle Configuration**. For example, you can remove the connection lines between partitions and I/O banks by turning off **Display connections to I/O banks**. You can also use the settings on the **Connection Counting** tab to adjust how the connections are counted in the bundles.

To optimize design performance, it is desirable to confine failing paths within individual design partitions, so that there are no failing paths passing between partitions, as discussed in earlier sections. The following steps allow you to view the critical timing paths from a TimeQuest Timing Analysis:

1. Open the TimeQuest Timing Analyzer and perform a timing analysis on the design.
2. In the Design Partition Planner, click **Show Timing Data** on the View menu.

In the top-level entity, child entities containing failing paths are marked by a small red dot in the upper right corner of the entity box.

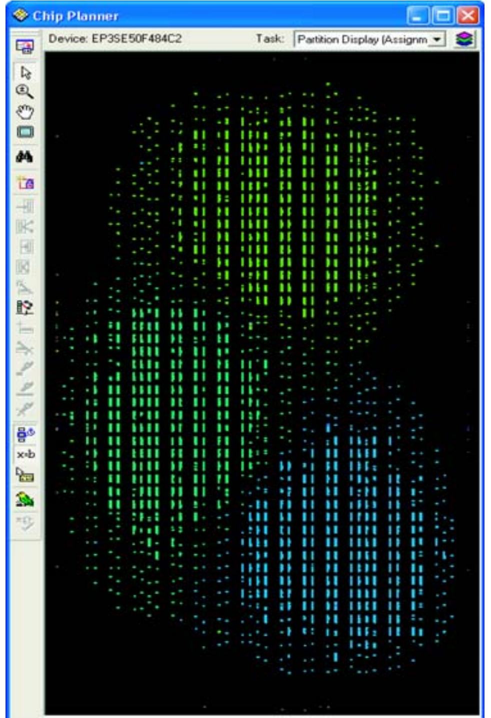
For more details about how to use the Design Partition Planner to analyze your design and create partitions, refer to "Using the Design Partition Planner" in the Quartus II Help.

Floorplan Partition Coloring

After making a set of partition assignments, it can be useful to view how the partitions are placed in the device. The Chip Planner can display nodes for each partition in a different color.

After compilation, in the Chip Planner **Task** list, select **Partition Display (Assignment)**, as shown in Figure 8–23. In this figure, you can see that the three different-colored partitions are grouped in three fairly independent areas of the device.

Figure 8–23. Partition Display in the Chip Planner Showing Three Partitions with Different Color Shades



Viewing Design Partition Planner and Floorplan Side-by-Side

You can view the Design Partition Planner together with the Chip Planner's Partition Planner, to analyze natural placement groupings in the floorplan view. This information can help you decide whether the design blocks should be grouped together in one partition, or whether they will make good partitions for the next compilation. It can also help determine whether the logic can easily be constrained by a LogicLock region to create a design floorplan. If logic naturally groups together when compiled without placement constraints, you can probably assign a reasonably sized LogicLock region to constrain the placement for future compilations. You can experiment by extracting different design blocks in the Design Partition Planner and viewing the placement results of those design blocks from the last compilation.

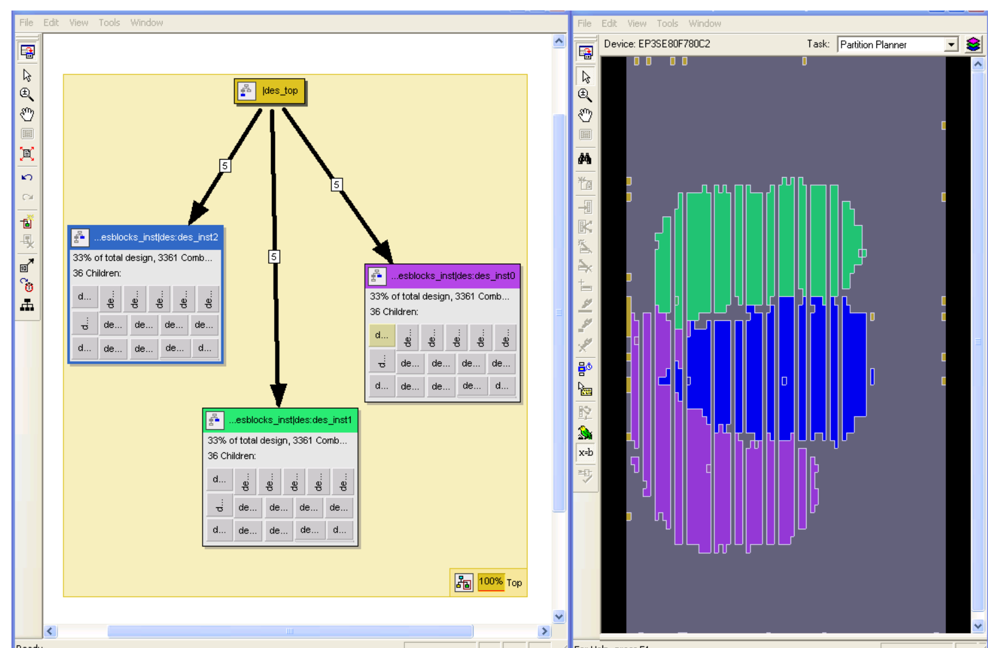
Open the Design Partition Planner, then open the Chip Planner and select the **Partition Planner** task in the **Task** list. This task selection displays the physical locations of design entities using the same colors as the Design Partition Planner display. For ease of viewing, drag and size the Chip Planner and Design partition Planner windows so they are side-by-side.

In the Design Partition Planner, extract instances of interest from their parents using the drag and drop method or the **Extract from Parent** command. Evaluate the physical locations of instances in the Chip Planner and the connectivity between instances displayed in the Design Partition Planner. An entity is generally not suitable to be set as a separate design partition or constrained in a LogicLock region if the Chip Planner shows it to be physically dispersed over a noncontiguous area of the device

after compilation. You can use the Design Partition Planner as described in “Design Partition Planner” on page 8–29 to analyze the design connections. For child instances that are unsuitable to be set as separate design partitions or placed in LogicLock regions, return those instances to their parent using the drag and drop method or the **Collapse to Parent** command.

Figure 8–24 shows a design displayed in both viewers, with different colors for the top-level design and the three major design instances.

Figure 8–24. Top-Level Design and Three Major Instances Shown in Both Viewers



Partition Statistics Report

You can view statistics about design partitions in the **Partition Merge Partition Statistics** compilation report and the **Statistics** tab in the **Design Partitions Properties** dialog box. These reports are useful when optimizing your design partitions in a top-down compilation flow, or when you are compiling the full top-level design in a bottom-up compilation flow, to ensure that the partitions meet the guidelines discussed in this document.

The **Partition Statistics** page under the **Partition Merge** folder of the Compilation Report lists statistics about each partition. The statistics for each partition (each row in the table) include the number of logic cells it contains, as well as the number of input and output pins and how many are registered. This report also lists how many ports are unconnected, or driven by a constant V_{CC} or GND. You can use this information to assess whether you have followed the guidelines for partition boundaries.

You can also view statistics about the resource and port connections for a particular partition on the **Statistics** tab of the **Design Partition Properties** dialog box. On the Assignments menu, click **Design Partitions Window**. Right-click on a partition and click **Properties** to open the dialog box. Click **Show All Partitions** to view all the partitions in the same report. The Design Partition Properties report also shows statistics for the Internal Congestion: Total Connections and Registered Connections. This represents how many signals are connected within the partition. It then lists the inter-partition connections for each partition, which helps you see how partitions are connected to each other.

Report Partition Timing in the TimeQuest Timing Analyzer

The TimeQuest Timing Analyzer includes a diagnostic report called Report Partitions, and the `report_partitions` SDC command. The resulting Partition Timing Overview lists the design partitions and provides the number of failing paths and the worst case timing slack within that partition. The function also creates a Partition Timing Details table that lists the number of failing paths and worst-case slack from each partition to the others.

You can use this report to analyze where the critical timing paths in the design are with respect to design partitions. If a certain partition contains many failing paths, or failing inter-partition paths, you may be able to change your partitioning scheme and improve your timing performance.

For more information about the TimeQuest `report_timing` command, see the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Ensure Partition Assignments Do Not Impact the Quality of Results

There is often a trade-off between compilation time and quality of results when you vary the number of partitions in a project. You can ensure that you limit any negative effect on the quality of results by following an iterative methodology during the partitioning process. In any incremental compilation flow in which you can compile the source code for every partition during the partition planning phase, Altera recommends the following iterative flow:

1. Start with a complete design that is not partitioned and has no location or LogicLock assignments.
2. To perform a placement and timing analysis estimate, on the Processing menu, point to **Start** and click **Start Early Timing Estimate**.

   You must perform Analysis and Synthesis and Partition Merge before performing an Early Timing Estimate.

   To run a full compilation instead of the Early Timing Estimate, on the Processing menu, click **Start Compilation**.
3. Record the quality of results from the Compilation Report (f_{MAX}, area, and any other relevant results).
4. Create design partitions following the guidelines described in this chapter.
5. Perform another Early Timing Estimate or a full compilation.

6. Record the quality of results from the Compilation Report. If the quality of results is significantly worse than those obtained in the previous compilation, repeat step 4 through step 6 to change your partition assignments and use a different partitioning scheme.
7. Even if the quality of results is acceptable, you can repeat step 4 through step 6 by further dividing a large partition into several smaller partitions. Doing so improves compilation time in future incremental compilations. You can repeat this step until you achieve a good trade-off point (that is, all critical paths are localized within partitions, the quality of results is not negatively affected, and the size of each partition is reasonable).

Introduction to Design Floorplans

A floorplan represents the layout of the physical resources on the device. The expressions "creating a design floorplan" and "floorplanning" describe the process of mapping the logical design hierarchy onto physical regions in the device floorplan.

In the Quartus II software, LogicLock regions are used to constrain blocks of a design to a particular region of the device. LogicLock regions represent a rectangular area of the device with a user-defined or Fitter-defined size and location on the device layout.

For more information about design floorplans and LogicLock regions, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

The Difference between Logical Partitions and Physical Regions

Design partitions are "logical" entities based on the design hierarchy. LogicLock regions are "physical" placement assignments that constrain logic to a rectangular region on the device.

It is a common misconception that logic from a design partition is always grouped together on the device when you use incremental compilation. This is not true. Logic from a partition can be placed anywhere in the device if it is not constrained to a LogicLock region. A logical design partition does not refer to any physical area of the device and does not directly control *where* instances are placed on the device.

If you want to control the placement of the logic from a design partition and isolate it to a particular part of the device, you can assign the logical design partition to a physical region in the device floorplan with a LogicLock region assignment. Creating a design floorplan by assigning design partitions to LogicLock regions is recommended to improve the quality of results and avoid placement conflicts in many situations for incremental compilation. Refer to "Why Create a Floorplan?" on page 8–36 for details.

Another misconception is that LogicLock assignments are used to preserve placement results for incremental compilation. This is also not true. LogicLock regions only *constrain* logic to a physical region of the device. Incremental compilation does not use LogicLock assignments or any location assignments to preserve the placement results; it simply reuses the results stored in the database netlist from the previous compilation.

Why Create a Floorplan?

Floorplan location planning can be important for a design that uses full incremental compilation, for the following two reasons:

- To avoid resource conflicts between partitions, predominantly when importing partitions from another Quartus II project
- To ensure a good quality of results when recompiling individual partitions in top-down flows

Why Create a Floorplan in Bottom-Up Flows?

Creating a design floorplan is required if you want to preserve placement for lower-level partitions in a bottom-up flow to avoid resource conflicts between partitions.

Location assignments for each partition ensure that there are no placement conflicts between different partitions. If there are no LogicLock region assignments, or if LogicLock regions are set to auto-size or floating, no device resources are specifically allocated for the logic associated with the region. If you do not clearly define this resource budget, logic placement can conflict when you import the partitions in a bottom-up flow.

Why Create a Floorplan in Top-Down Flows?

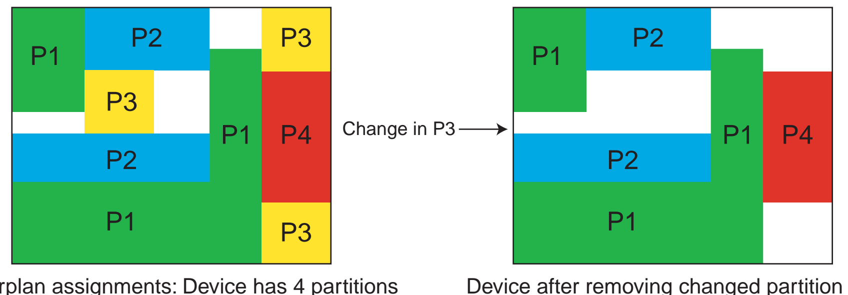
Creating a floorplan is highly recommended for timing-critical partitions to maintain good quality of results when the design changes.

Floorplan assignments are not required for non-critical partitions in a top-down flow. The logic for partitions that are not timing-critical (such as simple top-level glue logic) can be placed anywhere in the device on each recompilation if that is best for your design.

Design floorplan assignments prevent the situation in which the Fitter must place a partition in an area of the device where most resources are already used by other partitions. A LogicLock region provides a reasonable region to re-place logic after a change, so the Fitter does not have to scatter logic throughout the available space in the device.

Figure 8–25 illustrates the problems associated with refitting designs that do not have floorplan location assignments. It shows the initial placement of a four-partition design (P1-P4) without any floorplan location assignments. The second part of the figure shows the device if a change occurs to P3. After removing the logic for the changed partition, the Fitter must replace and reroute the new logic for P3 using the scattered white space shown in Figure 8–25. The placement of the post-fit netlists for other partitions forces the Fitter to implement P3 with the device resources that have not already been used.

Figure 8–25. Representation of Device Floorplan without Location Assignments



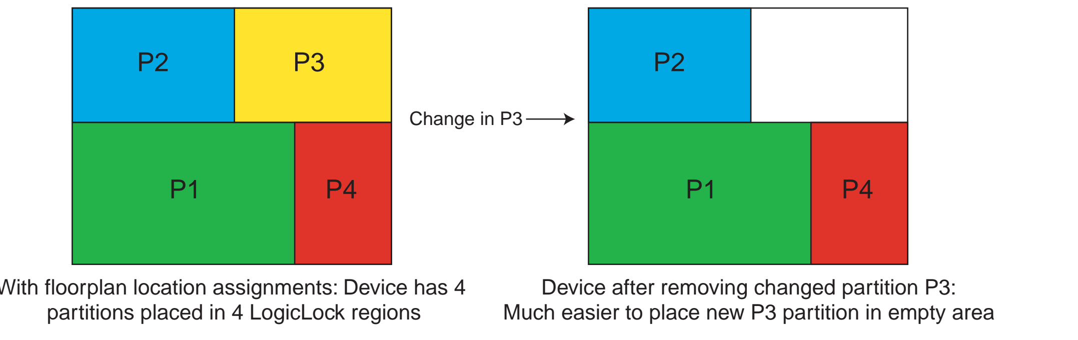
No floorplan assignments: Device has 4 partitions and the logic is placed throughout

Device after removing changed partition P3: New P3 must be placed in empty areas

The Fitter must work harder because of the more difficult physical constraints, and as a result, compilation time often increases. The Fitter might not be able to find any legal placement for the logic in partition P3, even if it could in the initial compilation. In addition, if the Fitter can find a legal placement, the quality of results often decreases in these cases, sometimes dramatically, because the new partition is now scattered throughout the device.

Figure 8–26 shows the initial placement of a four-partition design with floorplan location assignments. Each partition has been assigned to a LogicLock region. The second part of the figure shows the device after partition P3 is removed. This placement presents a much more reasonable task to the Fitter and yields better results.

Figure 8–26. Representation of Device Floorplan with Location Assignments



With floorplan location assignments: Device has 4 partitions placed in 4 LogicLock regions

Device after removing changed partition P3: Much easier to place new P3 partition in empty area

Altera recommends that you create a LogicLock floorplan assignment for any timing-critical blocks that will be recompiled as you make changes to the design.

When to Create a Floorplan

It is important that you plan early to incorporate partitions into the design, and ensure that each design partition follows the partitioning guidelines. You can make the floorplan assignments at different stages of the design flow, early or late in the flow. These guidelines help ensure better results when you start creating floorplan location assignments.

Early Floorplan

An early floorplan is created before the design stage. You can plan an early floorplan at the top level of a team-based design to give each designer a portion of the device resources. Doing so allows each designer to create the logic for their design partition without conflicting with other logic. Each design partition can be implemented independently and integrated later in the top-level project.

You can use an early floorplan as a rough draft of a floorplan for top-down flows as well, to roughly divide up the design partitions into LogicLock regions while iterating through the design cycle.

In a top-down flow, or after you have integrated the first version of all design partitions in a bottom-up flow, you can use the design information and Quartus II features to tune and improve the floorplan, as described in the following section.

Late Floorplan

A late floorplan is created or modified after the design has been created, when the code is close to complete and the design structure is likely to remain stable. When the design is complete, you can take advantage of the Quartus II analysis features to check the floorplan quality. To tune the floorplan, you can perform iterative compilations as required and assess the results of different assignments.

It may not be possible to create a good-quality late floorplan if you do not create partitions in the early stages of the design.

Creating a Design Floorplan: Placement Guidelines

The following guidelines are key to creating a good design floorplan:

- Capture correct resources in each region
- Use good region placement to maintain design performance compared to flat compilation

It is a common misconception that creating a floorplan enhances timing performance, as compared to a flat compilation with no location assignments. The Quartus II Fitter does not usually require guidance to get optimal results for a full design.

Floorplan assignments can help maintain good performance when designs change incrementally, as described in “Why Create a Floorplan in Top-Down Flows?” on page 8–36. However, bad placement assignments can often hurt performance results, as compared to a flat compilation, because the assignments limit the options for the Fitter. Investing some time to find good region placement is required to match the performance of a full flat compilation.

Use the following general strategy to create a floorplan:

1. Divide the design into partitions.
2. Assign the partitions to LogicLock Regions.
3. Compile the design.
4. Analyze the results.
5. Modify the placement and size of regions as required.

You may have to iterate through these steps several times to find the best combination of design partitions and LogicLock regions that meet the design's resource and timing goals.

For details about performing these steps, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Assigning Partitions to LogicLock Regions

To create a full floorplan: Create a LogicLock region for each partition (including the top-level) to assign all logic to a place in the device.

To create a partial floorplan: Create a LogicLock region for any critical or often-changing partitions.

Before compiling the design with new LogicLock assignments, ensure the affected partitions' Netlist Type is set so that the Fitter does not reuse previous placement results.

In most cases, each LogicLock region should contain logic from only one partition. This organization helps prevent resource conflicts in a bottom-up design and can lead to better performance preservation when locking down parts of a project in a top-down design.

The software is flexible and does allow exceptions to this rule. For example, you can place more than one partition in the same LogicLock region if the partitions are tightly connected. For best results, ensure that you recompile all such partitions every time the logic in one partition changes. In addition, if a partition contains multiple lower-level entities, you can place those entities in different areas of the device with multiple LogicLock regions (even though they are defined in the same partition).

You can use the **Reserved** LogicLock option to ensure that you avoid any conflicts with other logic which is not locked into any LogicLock region. This option prevents other logic from being placed in the region, and is useful if you have empty partitions at any point during your design flow, so that you can reserve space in the floorplan. Do not make reserved regions too large, to prevent unused area, because no other logic can be placed in a region with the **Reserved** LogicLock option.

How to Size and Place Regions

In an early floorplan, assign physical locations based on design specifications. Use information about the connections between partitions, the partition size, and the type of device resources required.

In a late floorplan when the design is complete, you can use Fitter-chosen regions as a guideline. Start with the default Auto size and Floating origin location. After compilation, lock the size and origin location. Instead of a full compilation, you can use the **Start Early Timing Estimate** command to perform a fast placement.

Alternatively, in a late floorplan, you can specify the size based on the synthesis results and use Fitter-chosen locations. Right-click on a region in the **LogicLock Regions** dialog box, and choose **Set to Estimated Size**. Like the previous option, start with Floating origin location. After compilation, lock the origin location. Again, instead of a full compilation, you can use the **Start Early Timing Estimate** command to perform a fast placement. You can also enable the **Fast Synthesis Effort** setting to reduce synthesis time.

After a compilation or early timing estimate, you should save the Fitter's size and/or origin location. Click on each LogicLock region in the LogicLock Regions Window while holding the Ctrl key to select all regions (including the top-level region). Right-click on the last selected LogicLock region and click **Set Size and Origin to Previous Fitter Results**.

It is important that you use the Fitter-chosen locations only as a starting point to give the regions a good fixed size and location. Ensure that all LogicLock regions in the design have a fixed size and have their origin locked to a specific location on the chip. On average, regions with fixed size and location yield better timing performance than auto-sized regions.

Modifying Region Size and Origin

After you have saved the Fitter's results from an initial compilation for a late floorplan, modify the regions using your knowledge of the design to set a specific size and location. If you have a good understanding of how the design fits together, you can often improve upon the regions placed in the initial compilation. In an early floorplan, you can use the guidelines in this section to set the size and origin, even though there is no initial Fitter placement for a basis.

The easiest way to move and resize regions is to drag the region location and borders in the Chip Planner. Ensure you select the **User-Defined** region in the floorplan (as opposed to the **Fitter-Placed** region from the last compilation) so that you can change the region.

Generally, you can keep the Fitter-determined relative placement of the regions, but make adjustments if required to meet timing performance. If you find that the Early Timing Estimate did not result in good relative placements, try performing a full compilation so that the Fitter can optimize for a full placement and routing.

If two LogicLock regions have several connections between them, ensure they are placed near each other to improve timing performance. By placing connected regions near each other, the Fitter has more opportunity to optimize inter-region paths when both partitions are recompiled. Reducing the criticality of inter-region paths also allows the Fitter more flexibility when placing the other logic in each region.

If resource utilization is low in the overall device, enlarge the regions. Doing so usually improves the final results because it gives the Fitter more freedom to place additional or modified logic added to the partition during future incremental compilations. It also allows room for optimizations such as pipelining and physical synthesis logic duplication.

Try to have each region evenly full, with the same "fullness" that the complete design would have without LogicLock regions. As a very rough suggestion, try to have each region approximately 75% full.

Allow more area for regions that are densely populated, because overly congested regions can lead to poor results. Allow more empty space for timing-critical partitions to improve results. However, do not make regions too large for their logic. Regions that are too large can result in wasted resources and also lead to suboptimal results.

Ideally, almost the entire device should be covered by LogicLock regions if all partitions are assigned to regions.

Regions should not overlap in the device floorplan. This is a requirement in bottom-up flows and a recommendation in top-down flows. In a bottom-up flow, if two partitions are allocated an overlapping portion of the chip, each may independently claim some common resources in this region. This leads to resource conflicts when importing bottom-up results into a final top-level design. In a top-down flow, overlapping regions give more difficult constraints to the Fitter and can lead to reduced quality of results.

You can create hierarchical LogicLock regions to ensure that the logic in a child partition is physically placed inside the LogicLock region for its parent partition. This can be useful when the parent partition does not contain registers at the boundary with the lower-level child partition and has a lot of signal connectivity. To create a hierarchical relationship between regions in the LogicLock Regions Window, drag and drop the child region to the parent region.

I/O Connections

Consider I/O timing when placing regions. Using I/O registers can minimize I/O timing problems, and using boundary registers on partitions can minimize problems connecting regions or partitions. However, I/O timing might still be a concern. It is most important for bottom-up flows where each partition is compiled independently, because the Fitter can optimize the placement for paths between partitions if the partitions are compiled at the same time.

Place regions close to the appropriate I/O, if necessary. For example, DDR memory interfaces have very strict placement rules to meet timing requirements. Incorporate any specific placement requirements into your floorplan as required. It is best to create LogicLock regions for internal logic only, and provide pin location assignments for external device I/O pins (instead of including the I/O cells in a LogicLock region to control placement).

LogicLock Resource Exclusions

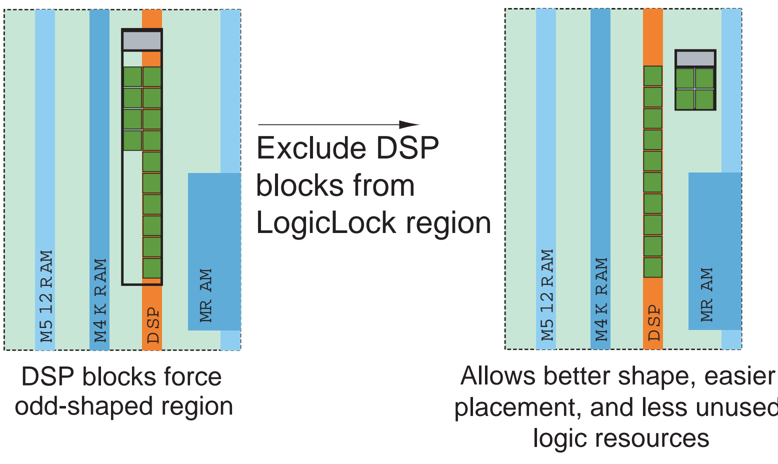
You can exclude certain resource types from a LogicLock region to manage the ratio of logic to dedicated DSP and RAM resources in the region.

If your design contains memory or digital signal processing (DSP) elements, you may want to exclude these elements from the LogicLock region. LogicLock resource exceptions prevent elements of certain types from being assigned to a region. Therefore, those elements are not required to be placed inside the region boundaries. The option does not *prevent* them from being placed inside the region boundaries unless the region's **Reserved** property is turned on.

Resource exceptions are useful in cases where it is difficult to place rectangular regions for design blocks that contain memory and DSP elements, because of their placement in columns throughout the device floorplan. Exclude RAMs, DSPs, or logic cells to give the Fitter more flexibility with region sizing and placement. Excluding RAM or DSP elements can help to resolve no-fit errors that are caused by regions

spanning too many resources, especially for designs that are memory-intensive, DSP-intensive, or both. Figure 8–27 shows an example of a design with an odd-shaped region to accommodate DSP blocks for a region that does not contain very much logic. The right side of the figure shows the result after excluding DSP blocks from the region. The region can be placed more easily without wasting logic resources. The DSP blocks are placed outside the region.

Figure 8–27. LogicLock Resource Exclusion Example



To view any resource exceptions, right-click in the LogicLock Regions Window and click **Properties**. In the **LogicLock Region Properties** dialog box, highlight the design element (module/entity) in the **Members** box and click **Edit**. To set up a resource exception, click the browse button under **Excluded element types**, then turn on the design element types to be excluded from the region. You can choose to exclude combinational logic or registers from logic cells, or any of the sizes of TriMatrix memory blocks, or DSP blocks.

If the excluded logic is in its own lower-level design entity (even if it is within the same design partition), you can assign the entity to a separate LogicLock region to constrain its placement in the device.

You can also use this feature with the LogicLock **Reserved** property to reserve specific resources for logic that will be added to the design.

Creating Non-Rectangular Regions

To constrain placement to non-rectangular areas of the device, you can limit entity placement to a sub-area of a LogicLock region. To do so, construct a LogicLock hierarchy by creating child regions inside of parent regions, and then use the **Reserved** option to control which logic can be placed inside these child regions.

Setting a region's **Reserved** option to **On** prevents the Fitter from placing nodes that are not assigned to the region inside the boundary of the region. Setting a region's **Reserved** option to **Limited** prevents the Fitter from placing nodes that are assigned to the immediate parent LogicLock region's hierarchy inside the boundary of the region. Any other logic can be placed inside the region. To create non-rectangular

regions for a specific entity, you can place child LogicLock regions inside a parent region and set the **Reserved** setting of the child regions to **Limited**. The child region prevents the parent region hierarchy from using that area of the device floorplan, but leaves it open for the rest of the design. You can assign other LogicLock regions to cover that area of the device if required.

For more information and examples of creating non-rectangular regions with the **Reserved** property, refer to *Examples of Using Limited Reserved Status to Constrain LogicLock Location Assignments* in the Quartus II Help.

Checking Floorplan Quality

This section provides an overview of tools that you can use as you create a floorplan in the Quartus II software. Take advantage of these tools to assess your floorplan quality and use the information to improve your design or assignments as required to achieve the best results.

Incremental Compilation Advisor

You can use the Incremental Compilation Advisor to check that your design follows the recommendations for creating floorplan location assignments that are presented in this document. Refer to “Incremental Compilation Advisor” on page 8–29 for more information.

LogicLock Region Resource Estimates

You can view resource estimates included in a LogicLock region to determine the region’s resource coverage. You can use this estimate before compilation to check region size. Using this estimate helps ensure adequate resources when you are sizing or moving regions.

Right-click in the LogicLock Regions Window, choose **Properties**, and select the **Size & Origin** tab. Specify a size and an origin to see the **Available resources** estimate in the dialog box.

LogicLock Region Properties Statistics Report

The LogicLock Region Properties Statistics are similar to the Design Partition Properties described in “Partition Statistics Report” on page 8–33, but include resource usage details after compilation.

The statistics report the number of resources used and the total resources covered by the region. The statistics also list the number of I/O connections and how many I/Os are registered (good), as well as the number of internal connections and the number of inter-region connections (bad).

Right-click in the LogicLock Regions Window, choose **Properties** and select the **Statistics** tab. Click **Show All Regions** to see all regions displayed in the same report.

Critical Path Settings for Chip Planner

The **Critical Path Settings** dialog box allows you to display the most critical paths from the Timing Analyzer report in the Chip Planner floorplan view. You can specify a threshold for which paths to highlight in the Chip Planner. Use this information to identify inter-region critical paths and improve your partition or floorplan assignments.

Locate the Quartus II TimeQuest Timing Analyzer Path in the Chip Planner

In the TimeQuest user interface, you can locate a specific path in the Chip Planner to view its placement. Perform a report timing operation (for example, report timing for all paths with less than 0 ns slack). Right-click in the detailed path report (**Data Path** tab) for a specific path and choose **Locate Path**. Click **OK** to choose the Chip Planner.

Inter-Region Connection Bundles

The Chip Planner can display bundles of connections between LogicLock regions, with filtering options that allow you to choose the relevant data for display. These bundles can help you visualize how many connections there are between each LogicLock region, to improve floorplan assignments, or to change partition assignments if required.

With the Chip Planner open, on the View menu, click **Generate Inter-region Bundles**.

Routing Utilization

The Chip Planner includes a mode to display a color map of routing congestion. This display helps identify areas of the chip that are too tightly packed.

In the Chip Planner, click the Layer Settings icon next to the **Task** list. Change the **Background Color Map** to **Routing Utilization** (the default is Block Utilization).

The darker-colored LAB blocks indicate higher routing congestion. Move your mouse pointer over a LAB to see a tool tip that reports the logic and routing utilization information.

Ensure Floorplan Assignments Do Not Impact Quality of Results

The end results of design partitioning and floorplan creation differ from design to design. However, it is important to evaluate your results to ensure that your scheme is successful. Compare the results before creating your floorplan location assignments to the results after doing so. Consider using another scheme if any of the following guidelines are not met:

- You should see no degradation in f_{MAX} after the design is partitioned and floorplan location assignments are created. In many cases, a slight increase in f_{MAX} is possible
- The area increase should be no more than 5% after the design is partitioned and floorplan location assignments are created
- The time spent in the routing stage should not significantly increase

The amount of compilation time spent in the routing stage is reported in the Messages window by an Info message that indicates the elapsed time for Fitter routing operations. If you notice a dramatic increase in routing time, the floorplan location assignments may be creating substantial routing congestion. In this case, decrease the number of LogicLock regions. Doing so typically reduces the compilation time in subsequent incremental compilations and may also improve design performance.

Recommended Design Flows and Application Examples

This section provides design flows for partitioning and creating a design floorplan during common timing closure and team-based design scenarios. Each flow describes the situation in which it should be used, and gives a step-by-step description of the commands required to implement the flow.

Create a Floorplan for the Entire Design in a Top-Down Flow

Use this flow for top-down incremental compilation designs in which you would like to assign a floorplan location for each design block that is assigned as a separate partition. This is the standard floorplan procedure described in the *Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. A full floorplan ensures that partitions do not interact as they are changed and recompiled—each partition has its own area of the device floorplan.

To create a LogicLock region for each design partition, use the following general methodology:

1. On the Assignments menu, click **Design Partitions Window** and ensure that all partitions have their Netlist Type set to **Source File** or **Post-Synthesis**. If the Netlist Type is set to **Post-Fit**, floorplan location assignments are not used when recompiling the design.
2. Create a LogicLock region for each partition (including the top-level entity, which is automatically considered a partition).
3. On the Processing menu, point to **Start** and click **Start Early Timing Estimate** to place auto-sized, floating-location LogicLock regions.

   You must perform Analysis and Synthesis, and Partition Merge before performing an Early Timing Estimate.

   To run a full compilation instead of the Early Timing Estimate, on the Processing menu, click **Start Compilation**.
4. On the Assignments menu, click **LogicLock Regions Window**, and click on each LogicLock region while holding the Ctrl key to select all regions (including the top-level region).
5. Right-click on the last selected LogicLock region, and click **Set Size and Origin to Previous Fitter Results**.
6. If required, modify the size and location with the LogicLock Regions Window or the Chip Planner. For example, make the regions bigger to fill up the device and allow for future logic changes.

7. On the Processing menu, point to **Start** and click **Start Early Timing Estimate** to estimate the timing performance of your design with these LogicLock regions.
8. Repeat step 6 and 7 until you are satisfied with the quality of results for your design floorplan. On the Processing menu, click **Start Compilation** to run a full compilation.

Create a Floorplan as the Project Lead in a Bottom-Up Flow

Use this approach when you have several lower-level subdesigns that will be implemented separately by different designers. The subdesign designers want to optimize their designs independently and pass the results on to you, the project lead.

As the project lead in this scenario, perform the following steps to prepare the design for a successful bottom-up design methodology with early floorplan planning:

1. Create a new Quartus II project that will ultimately contain the full implementation of the entire design.
2. To prepare for the bottom-up methodology, create a “skeleton” of the design that defines the hierarchy for the subdesigns that will be implemented by separate designers. Consider the partitioning guidelines in this chapter while determining the design hierarchy.
3. Make project-wide settings. Select the device, make global assignments for clocks and device I/O ports, and make any global signal constraints to specify which signals can use global routing resources.
4. Make design partition assignments for each major subdesign and set the Netlist Type for each design partition that will be imported to **Empty** in the Design Partitions window.
5. Create LogicLock regions for each of the lower-level partitions to create a design floorplan. This floorplan should consider the connectivity between partitions and estimates of the size of each partition based on any initial implementation numbers and knowledge of the design specifications. Use the guidelines described in this chapter to choose a size and location for each LogicLock region.
6. On the Project menu, click **Generate Bottom-Up Design Partition Scripts**, or run the script generator from a Tcl prompt or the command prompt.
7. Make changes to the default script options as desired. Altera recommends that you pass all the default constraints, including LogicLock regions, for all partitions and virtual pin location assignments. Altera further recommends that you add a maximum delay timing constraint for the virtual I/O connections in each partition to help timing closure during integration at the top level. If lower-level projects have not already been created by the other designers, use the partition script to set up the projects so that you can easily take advantage of makefiles.
8. Provide each lower-level designer with the Tcl file to create their project with the appropriate constraints. If you are using makefiles, provide the makefile for each partition.

Create a Floorplan Assignment for One Design Block with Difficult Timing

Use this flow when you have one timing-critical design block that requires more optimization than the rest of your design. You can take advantage of incremental compilation to reduce your compilation time without creating a full design floorplan.

In this scenario, you may not have to create floorplan assignments for the entire design. You can create a region to constrain the location of your critical design block, and allow the rest of the logic to be placed anywhere else in the device. Use the following general methodology:

1. Divide up your design into partitions to reduce compilation time. Consider the guidelines in this chapter while determining the partition boundaries. Ensure that you isolate the timing-critical logic in a separate design partition.
2. Define a LogicLock region for the timing-critical design partition. Ensure that you capture the correct amount of device resources in the region. Turn on the **Reserved** property to prevent any other logic from being placed in the region.
   - If the design block is not complete, reserve space in the design floorplan based on your knowledge of the design specifications, connectivity between design blocks, and estimates of the size of the partition based on any initial implementation numbers.
   - If the critical design block has initial source code ready, compile the design as in the scenario "Create a Floorplan for the Entire Design in a Top-Down Flow" on page 8–45 to place the LogicLock region. Save the Fitter-determined size and origin, then enlarge the region to provide more flexibility and allow for future design changes.
3. As the rest of the design is completed, and the device fills up, the timing-critical region has a reserved area of the floorplan. When you make changes to the design block, the logic can be re-placed in the same part of the device, which helps ensure good quality of results.

Potential Issues with Creating Partitions and Floorplan Assignments

There are some limitations and restrictions on using incremental compilation and using certain design flows with certain Altera features.

Refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook* for complete details about restrictions and limitations.

Consider documented limitations and restrictions as you plan your design flow and select partitions. Most limitations and restrictions do not affect most users, but it is helpful to know if you must modify your partitions or design flow to accommodate certain restrictions.

There are also possible utilization effects due to partitioning and creating a floorplan. These are described in the following subsections. Consider these effects if your design is close to using all of the device resources before adding partition or floorplan assignments.

Logic and Resource Utilization Effects

Partitions can increase resource utilization due to cross-partition optimization limitations. Floorplan assignments can increase resource utilization because regions sometimes lead to unused logic. Follow the recommendations in this document to reduce these effects.

If your device is very full with the flat version of design, you might not be able to use a complete incremental flow for the entire design. You can use a "partial" incremental flow instead to get compilation time and performance preservation benefits for key parts of the design. Focus on creating partitions and floorplan assignments for timing-critical or often-changing blocks to get the most benefit out of the feature.

Routing Utilization Effects

Partitions and floorplan assignments typically increase routing utilization compared to a flat design. Follow the recommendations in this document to reduce the effect.

If long compilation times are due to routing congestion, you might not be able to use incremental flows to reduce compilation time. Focus on creating partitions and floorplan assignments for parts of the design that are not routing-critical to get some benefit.

You can also use incremental compilation to lock routing for routing-critical blocks only (with other partitions empty), and then compile the rest of the design after the critical block meets its requirements.

Review the Fitter Messages to check how much time is spent during routing optimizations and to see the percentage of routing utilization. This information helps highlight routing issues.

Conclusion

Incremental compilation provides a number of benefits, especially to large, complex designs. To take advantage of the feature, it is worth spending some time to create quality partition and floorplan assignments.

Follow the guidelines to set up your design hierarchy and source code for incremental compilation. Keep partitions independent of each other and do not rely on any cross-boundary logic optimizations.

Floorplan location assignments are required for bottom-up flows and are recommended for timing-critical partitions in top-down flows. Follow the guidelines to create and modify LogicLock regions to create good placement assignments for your design partitions.

Take advantage of the numerous Quartus II software tools to assess partition quality and analyze the floorplan to make good partition and LogicLock location assignments. Remember that you do not have to follow all the guidelines exactly to implement an incremental compilation design flow, but following the guidelines can maximize your chances of success.

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*

Revision History

Table 8–1 shows the revision history for this chapter.

Table 8–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Added I/O register packing examples from *Incremental Compilation for Hierarchical and Team-Based Designs* chapter ■ Moved "Incremental Compilation Advisor" section ■ Added "Viewing Design Partition Planner and Floorplan Side-by-Side" section ■ Updated Figure 8–22 ■ Chapter 8 was previously Chapter 7 in software release 8.1.	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	Updated for the Quartus II software version 8.1 release.
May 2007 v8.0.0	Initial release.	This content of this chapter is based on information that was contained in Application Note 470.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

Section III. Synthesis

As programmable logic devices (PLDs) become more complex and require increased performance, advanced design synthesis has become an important part of the design flow. In the Quartus® II software you can use the Analysis and Synthesis module of the Compiler to analyze your design files and create the project database. You can also use other EDA synthesis tools to synthesize your designs, and then generate EDIF netlist files or Verilog Quartus Mapping Files (**.vqm**) that you can use with the Quartus II software. This section explains the options that are available for each of these flows and how they are supported in the Quartus II, version 9.0 software.

This section includes the following chapters:

- Chapter 9, Quartus II Integrated Synthesis
- Chapter 10, Synopsys Synplify Support
- Chapter 11, Mentor Graphics Precision Synthesis Support
- Chapter 12, Mentor Graphics LeonardoSpectrum Support
- Chapter 13, Analyzing Designs with Quartus II Netlist Viewers

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

9. Quartus II Integrated Synthesis

QII51008-9.0.0

Introduction

As programmable logic designs become more complex and require increased performance, advanced synthesis has become an important part of the design flow. The Quartus® II software includes advanced integrated synthesis that fully supports VHDL and Verilog HDL, as well as Altera®-specific design entry languages, and provides options to control the synthesis process. With this synthesis support, the Quartus II software provides a complete, easy-to-use solution.

This chapter documents the design flow and language support in the Quartus II software. It explains how you can use incremental compilation to reduce your compilation time, how you can improve synthesis results with Quartus II synthesis options, and how you can control the inference of architecture-specific megafunctions. This chapter also explains some of the node-naming conventions used during synthesis to help you better understand your synthesized design and the messages issued during synthesis to improve your HDL code. Scripting techniques for applying all the options and settings described are also provided.

This chapter contains the following sections:

- "Design Flow" on page 9–2
- "Language Support" on page 9–4
- "Incremental Compilation" on page 9–20
- "Quartus II Synthesis Options" on page 9–22
- "Analyzing Synthesis Results" on page 9–69
- "Analyzing and Controlling Synthesis Messages" on page 9–70
- "Node-Naming Conventions in Quartus II Integrated Synthesis" on page 9–74
- "Scripting Support" on page 9–80

This chapter provides examples of how to use attributes described within the chapter, but does not cover specific coding examples.

For examples of Verilog HDL and VHDL code synthesized for specific logic functions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. For information about coding with primitives that describe specific low-level functions in Altera devices, refer to the *Designing With Low-Level Primitives User Guide*.

Design Flow

The Quartus II Analysis and Synthesis process includes Quartus II integrated synthesis, which fully supports Verilog HDL and VHDL languages as well as Altera-specific languages, and supports major features in the SystemVerilog language (refer to "Language Support" on page 9–4 for details). This stage of the compilation flow performs logic synthesis to optimize design logic and performs technology mapping to implement the design logic using device resources, such as logic elements (LEs) or adaptive logic modules (ALMs) and other dedicated logic blocks. This stage also generates the single project database that integrates all the design files in a project (including any netlists from third-party synthesis tools).

You can use the Analysis and Synthesis stage of the Quartus II compilation to perform any of the following levels of Analysis and Synthesis:

- Analyze Current File—Parse the current design source file to check for syntax errors. This command does not report on many semantic errors that require further design synthesis. On the Processing menu, click **Analyze Current File**.
- Analysis & Elaboration—Check a design for syntax and semantic errors and perform elaboration to identify the design hierarchy. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration**.
- Analysis & Synthesis—Perform complete Analysis and Synthesis on a design, including technology mapping. On the Processing menu, point to **Start** and click **Start Analysis & Synthesis**. This is the most commonly used command and is part of the full compilation flow.

The Quartus II design and compilation flow using Quartus II integrated synthesis consists of the following steps:

1. Create a project in the Quartus II software and specify the general project information, including the top-level design entity name. On the File menu, click **New Project Wizard**.
2. Create design files in the Quartus II software or with a text editor.
3. On the Project menu, click **Add/Remove Files in Project** and add all design files to your Quartus II project using the **Files** page of the **Settings** dialog box.
4. Specify compiler settings that control the compilation and optimization of the design during synthesis and fitting. For synthesis settings, refer to "Quartus II Synthesis Options" on page 9–22. Add timing constraints to specify the timing requirements.
5. Compile the design in the Quartus II software. To synthesize the design, on the Processing menu, point to **Start**, and click **Start Analysis & Synthesis**.

   ☞ On the Processing menu, click **Start Compilation** to run a complete compilation flow including placement, routing, creation of a programming file, and timing analysis.

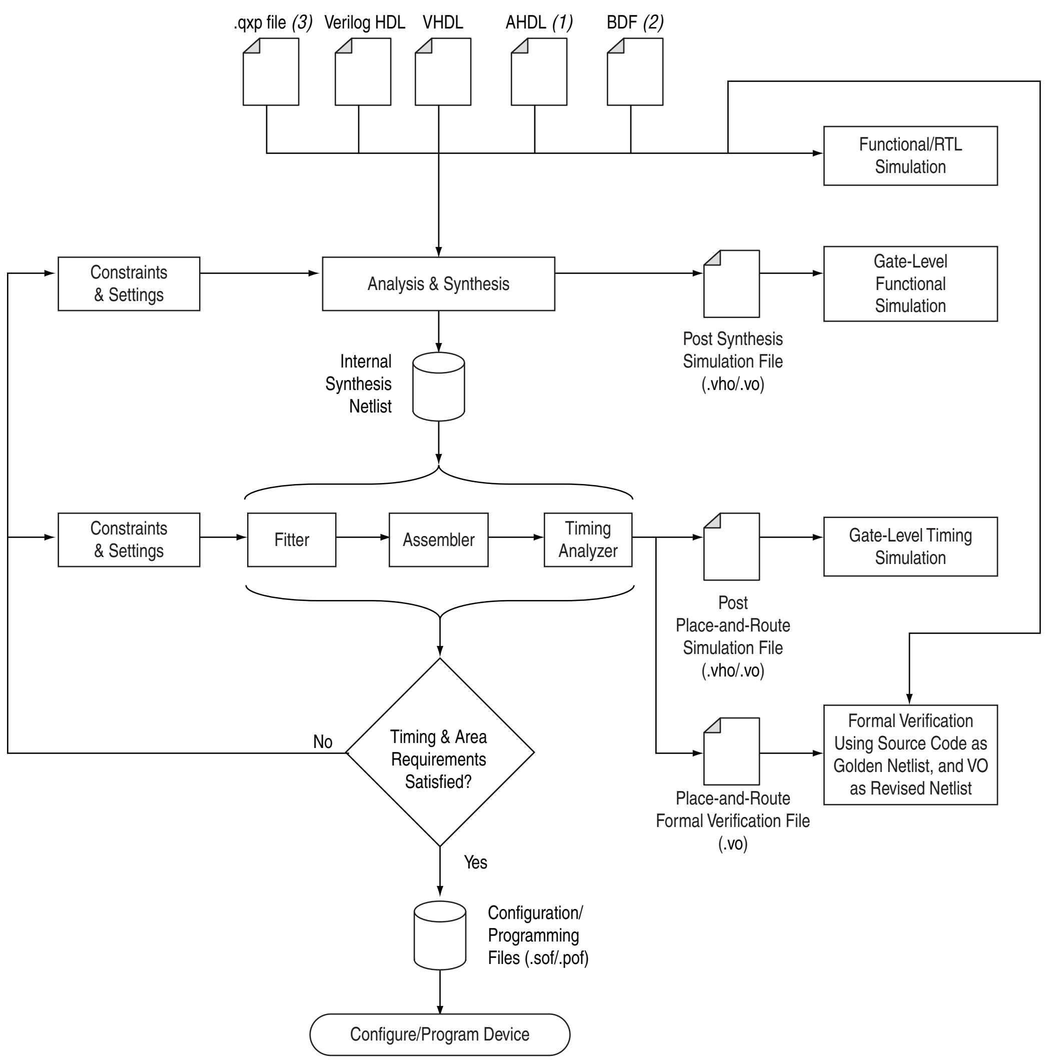
6. After obtaining synthesis and place-and-route results that meet your requirements, program or configure the Altera device.

The software provides netlists that allow you to perform functional simulation and gate-level timing simulation in the Quartus II simulator or a third-party simulator, perform timing analysis in a third-party timing analysis tool in addition to the TimeQuest Timing Analyzer or Classic Timing Analyzer, and/or perform formal verification in a third-party formal verification tool. The Quartus II software also provides many additional analysis and debugging features.

For more information about creating a project, compilation flow, and other features in the Quartus II software, refer to the Quartus II Help. For an overall summary of Quartus II features, refer to the *Introduction to the Quartus II Software* manual.

Figure 9–1 shows the basic design flow using Quartus II integrated synthesis.

Figure 9–1. Quartus II Design Flow Using Quartus II Integrated Synthesis



Notes to Figure 9–1:

(1) AHDL is the Altera Hardware Description Language.

(2) BDF is Altera's schematic Block Design File format (**.bdf**).

(3) The Quartus II eXported Partition (**.qxp**) file is a precompiled netlist that can be used as a design source file. For more information, refer to "Quartus II eXported Partition (.qxp) File as Source" on page 9–21.

Language Support

This section explains the Quartus II software's integrated synthesis support for HDL and schematic design entry, as well as graphical state machine entry, and explains how to specify the Verilog HDL or VHDL language version used in your design. It also documents language features such as Verilog HDL macros, initial constructs and memory system tasks, and VHDL libraries. "Design Libraries" on page 9–12 describes how to compile and reference design units in different custom libraries and "Using Parameters/Generics" on page 9–16 describes how to use parameters or generics and pass them between different languages.

To ensure that the software reads all associated project files, add each file to your Quartus II project. To add files to your project in the Quartus II GUI, on the Project menu, click **Add/Remove Files In Project**. Design files can be added to the project in any order. You can mix all supported languages and netlists generated by third-party synthesis tools in a single Quartus II project.

Verilog HDL Support

The Quartus II compiler's Analysis and Synthesis module supports the following Verilog HDL standards:

- Verilog-1995 (IEEE Standard 1364-1995)
- Verilog-2001 (IEEE Standard 1364-2001)
- SystemVerilog-2005 (IEEE Standard 1800-2005) (not all constructs are supported)

For complete information about specific Verilog HDL syntax features, and language constructs, refer to the Quartus II Help.

The Quartus II compiler uses the Verilog-2001 standard by default for files that have the extension **.v**, and the SystemVerilog standard for files that have the extension **.sv**.

The Verilog HDL code samples provided in this document follow the Verilog-2001 standard unless otherwise specified.

To specify a default Verilog HDL version for all files, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Settings** dialog box, under **Category**, expand **Analysis & Synthesis Settings**, and select **Verilog HDL Input**.
3. On the **Verilog HDL Input** page, under **Verilog version**, select the appropriate Verilog HDL version, then click **OK**.

You can override the default Verilog HDL version for each Verilog HDL design file by performing the following steps:

1. On the Project menu, click **Add/Remove Files in Project**. The **Settings** dialog box appears.
2. On the **Files** page, select the appropriate file in the list and click the **Properties** button.

 © March 2009 Altera Corporation

3. In the **HDL Version** list, select **SystemVerilog_2005**, **Verilog_2001**, or **Verilog_1995** and click **OK**.

You can also control the Verilog HDL version inside a design file using the `VERILOG_INPUT_VERSION` synthesis directive, as shown in Example 9–1. This directive overrides the default HDL version and any HDL version specified in the **File Properties** dialog box.

Example 9–1. Controlling the Verilog HDL Input Version with a Synthesis Directive

```
// synthesis VERILOG_INPUT_VERSION <language version>
```

The variable *<language version>* takes one of the following values:

- `VERILOG_1995`
- `VERILOG_2001`
- `SYSTEMVERILOG_2005`

When the software reads a `VERILOG_INPUT_VERSION` synthesis directive, the current language version changes as specified until the end of the file, or until the next `VERILOG_INPUT_VERSION` directive is reached.

You cannot change the language version in the middle of a Verilog HDL module.

For more information about specifying synthesis directives, refer to “Synthesis Directives” on page 9–27.

If you use scripts to add design files, you can use the `-HDL_VERSION` command to specify the HDL version for each design file. Refer to “Adding an HDL File to a Project and Setting the HDL Version” on page 9–80.

The Quartus II software support for Verilog HDL is case-sensitive in accordance with the Verilog HDL standard. The Quartus II software supports the compiler directive `` `define``, in accordance with the Verilog HDL standard.

The Quartus II software supports the `include` compiler directive to include files with absolute paths (with either “/” or “\” as the separator), or relative paths (relative to project root, user libraries, or current file location). When searching for a relative path, the Quartus II software initially searches relative to the project directory. If the software cannot find the file, it then searches relative to all user libraries, and finally relative to the directory location of the current file.

Verilog-2001 Support

The Quartus II software does not support Verilog-2001 libraries and configurations.

SystemVerilog Support

The Quartus II software supports the following SystemVerilog constructs:

- Parameterized interfaces, generic interfaces, and `modport` constructs
- Packages
- `Extern` module declarations
- Built-in data types `logic`, `bit`, `byte`, `shortint`, `longint`, `int`

- Unsized integer literals `'0`, `'1`, `'x`, `'z`, `'X`, and `'Z`
- Structure data types using `struct`
- Ports and parameters with unrestricted data types
- User-defined types using `typedef`
- Global declarations of task/functions/parameters/types (does not support global variables)
- Coding constructs `always_comb`, `always_latch`, `always_ff`
- Continuous assignments to nodes other than `nets`, and procedural assignments to nodes other than `reg`
- Enumeration methods `First`, `Last`, `Next(n)`, `Prev(n)`, `Num`, and `Name`
- Assignment operators `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `<<<=`, and `>>>=`
- Increment `++` and decrement `--`
- Jump statements `return`, `break`, and `continue`
- Enhanced `for` loop (declare loop variables inside initial condition)
- `Do-while` loop and local loop constructs
- Assignment patterns
- Keywords `unique` and `priority` in case statements
- Default values for function/task arguments
- Closing labels
- Extensions to directives `` `define `` and `` `include ``
- Expression size system function `$bits`
- Array query system functions `$dimensions`, `$unpacked_dimensions`, `$left`, `$right`, `$high`, `$low`, `$increment`, `$size`
- Packed array (include multidimensional packed array)
- Unpacked array (include single-valued range dimension)
- Implicit port connections with **.name** and **.***

Quartus II integrated synthesis also parses, but otherwise ignores, SystemVerilog assertions.

Designs written to comply with the Verilog-2001 standard might not compile successfully using the SystemVerilog setting because the SystemVerilog standard adds a number of new reserved keywords. For a list of reserved words in each language standard, refer to the Quartus II Help.

Initial Constructs and Memory System Tasks

The Quartus II software infers power-up conditions from Verilog HDL `initial` constructs. The software creates power-up settings for variables, including RAM blocks. If the Quartus II software encounters non-synthesizable constructs in an `initial` block, it generates an error. To avoid such errors, enclose non-synthesizable constructs (such as those intended only for simulation) in `translate_off` and `translate_on` synthesis directives, as described in "Translate Off and On / Synthesis Off and On" on page 9–62. Synthesis of initial constructs enables the power-up state of the synthesized design to match, as closely as possible, the power-up state of the original HDL code in simulation.

Initial blocks do not infer power-up conditions in some third-party EDA synthesis tools. If you are converting between synthesis tools, ensure that your power-up conditions are set correctly.

Quartus II integrated synthesis supports the `$readmemb` and `$readmemh` system tasks to initialize memories. Example 9–2 shows an initial construct that initializes an inferred RAM with `$readmemb`.

Example 9–2. Verilog HDL Code: Initializing RAM with the readmemb Command

```
reg [7:0] ram[0:15];
initial
begin
$readmemb("ram.txt", ram);
end
```

When creating a text file to use for memory initialization, specify the address using the format @*<location>* on a new line, then specify the memory word such as `110101` or `abcde` on the next line. Example 9–3shows a portion of a memory initialization file for the RAM in Example 9–2.

Example 9–3. Text File Format: Initializing RAM with the readmemb Command

```
@0
00000000
@1
00000001
@2
00000010
…
@e
00001110
@f
00001111
```

Verilog HDL Macros

The Quartus II software fully supports Verilog HDL macros, which you can define with the `'define` compiler directive in your source code. You can also define macros in the GUI or on the command line.

Setting a Verilog HDL Macro Default Value in the GUI

To specify a macro in the GUI, on the Assignments menu, click **Settings**. In the **Category** list, expand **Analysis & Synthesis Settings** and select **Verilog HDL Input**. Under **Verilog HDL macro**, type the macro name in the **Name** box, the value in the **Setting** box, and click **Add**.

Setting a Verilog HDL Macro Default Value on the Command Line

To set a default value for a Verilog HDL macro on the command line, use the `--verilog_macro` option, as shown in Example 9–4.

Example 9–4. Command Syntax for Specifying a Verilog HDL Macro

```
quartus_map <Design name> --verilog_macro= "<Macro name>=<Macro setting>" ↵
```

The command in Example 9–5 has the same effect as specifying `` `define a 2 `` in the Verilog HDL source code.

Example 9–5. Specifying a Verilog HDL Macro a = 2

```
quartus_map my_design --verilog_macro="a=2" ↵
```

To specify multiple macros, you can repeat the option more than once, as in Example 9–6.

Example 9–6. Specifying Verilog HDL Macros a = 2 and b = 3

```
quartus_map my_design --verilog_macro="a=2" --verilog_macro="b=3" ↵
```

VHDL Support

The Quartus II compiler's Analysis and Synthesis module supports the following VHDL standards:

- VHDL 1987 (IEEE Standard 1076-1987)
- VHDL 1993 (IEEE Standard 1076-1993)

For information about specific VHDL syntax features and language constructs, refer to the Quartus II Help.

The Quartus II compiler uses the VHDL 1993 standard by default for files that have the extension **.vhdl** or **.vhd**.

The VHDL code samples provided in this document follow the VHDL 1993 standard.

To specify a default VHDL version for all files, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, expand **Analysis & Synthesis Settings** and select **VHDL Input**.
3. On the **VHDL Input** page, under **VHDL version**, select the appropriate version, then click **OK**.

You can override the default VHDL version for each VHDL design file by performing the following steps:

1. On the Project menu, click **Add/Remove Files in Project**. The **Settings** dialog box appears.
2. On the **Files** page, select the appropriate file in the list and click **Properties**.
3. In the HDL version list, select **VHDL93** or **VHDL87** and click **OK**.

You can also specify the VHDL version for each design file using the VHDL_INPUT_VERSION synthesis directive, as shown in Example 9–7. This directive overrides the default HDL version and any HDL version specified in the **File Properties** dialog box.

Example 9–7. Controlling the VHDL Input Version with a Synthesis Directive

```
--synthesis VHDL_INPUT_VERSION <language version>
```

The variable <*language version*> takes one of the following values:

- VHDL87
- VHDL93

When the software reads a VHDL_INPUT_VERSION synthesis directive, it changes the current language version as specified until the end of the file, or until it reaches the next VHDL_INPUT_VERSION directive.

You cannot change the language version in the middle of a VHDL design unit.

For more information about specifying synthesis directives, refer to "Synthesis Directives" on page 9–27.

If you use scripts to add design files, you can use the -HDL_VERSION command to specify the HDL version for each design file. Refer to "Adding an HDL File to a Project and Setting the HDL Version" on page 9–80.

The Quartus II software reads default values for registered signals defined in the VHDL code and converts the default values into power-up level settings. This enables the power-up state of the synthesized design to match, as closely as possible, the power-up state of the original HDL code in simulation.

VHDL Standard Libraries and Packages

The Quartus II software includes the standard IEEE libraries and a number of vendor-specific VHDL libraries. For information about organizing your own design units into custom libraries, refer to "Design Libraries" on page 9–12.

The **IEEE** library includes the standard VHDL packages std_logic_1164, numeric_std, numeric_bit, and math_real. The **STD** library is part of the VHDL language standard and includes the packages standard (included in every project by default) and textio. For compatibility with older designs, the Quartus II software also supports the following vendor-specific packages and libraries:

- Synopsys packages such as std_logic_arith and std_logic_unsigned in the **IEEE** library

- Mentor Graphics® packages such as `std_logic_arith` in the **ARITHMETIC** library
- Altera primitive packages `altera_primitives_components` (for primitives such as `GLOBAL` and `DFFE`) and `maxplus2` (for legacy support of MAX+PLUS® II primitives) in the **ALTERA** library
- Altera megafunction packages `altera_mf_components` and `stratixgx_mf_components` in the **ALTERA_MF** library (for Altera-specific megafunctions including LCELL), and `lpm_components` in the **LPM** library for library of parameterized modules (LPM) functions.

For a complete listing of library and package support, refer to the Quartus II Help.

Beginning with the Quartus II software version 5.1, you should import component declarations for Altera primitives such as `GLOBAL` and `DFFE` from the `altera_primitives_components` package and not the `altera_mf_components` package.

VHDL wait Constructs

The Quartus II software supports only a single VHDL `wait until` statement in a process block. Other VHDL wait constructs, such as `wait for`, or `wait on` statements, or processes with multiple `wait` statements, are not synthesizable.

Example 9–8 is a VHDL code example of a supported `wait until` construct.

Example 9–8. VHDL Code: Supported wait until Construct

```
architecture dff_arch of ls_dff is
begin
output: process begin
wait until (CLK'event and CLK='1');
Q <= D;
Qbar <= not D;
end process output;
end dff_arch;
```

Example 9–9 is a VHDL code example of unsupported `wait for` construct. The process block a with `wait for`, or `wait on` statement is not synthesizable.

Example 9–9. VHDL Code: Unsupported wait for Construct

```
process
begin
CLK <= '0';
wait for 20 ns;
CLK <= '1';
wait for 12 ns;
end process;
```

The process block with multiple `wait until` statements is not synthesizable. Example 9–10 shows an example of multiple `wait until` statements in a process block.

Example 9–10. Multiple wait until Statements in a Process Block

```
output: process begin
wait until (CLK'event and CLK='1');
Q <= D;
Qbar <= not D;

wait until (CLK'event and CLK='0');
Q <= 0;
Qbar <= 1;
end process output;
```

AHDL Support

The Quartus II compiler's Analysis and Synthesis module fully supports the Altera Hardware Description Language (AHDL).

AHDL designs use Text Design Files (**.tdf**). You can import AHDL Include Files (**.inc**) into a **.tdf** file with an AHDL `include` statement. Altera provides **.inc** files for all megafunctions shipped with the Quartus II software.

For information about specific AHDL syntax features and language constructs, refer to the Quartus II Help.

The AHDL language does not support the synthesis directives or attributes described in this chapter.

Schematic Design Entry Support

The Quartus II compiler's Analysis and Synthesis module fully supports Block Design Files (**.bdf**) for schematic design entry.

You can use the Quartus II software's Block Editor to create and edit **.bdf** files and open Graphic Design Files (**.gdf**) imported from the MAX+PLUS II software. Use the Symbol Editor to create and edit Block Symbol Files (**.bsf**) and open MAX+PLUS II Symbol Files (**.sym**). You can read and edit these legacy MAX+PLUS II formats with the Quartus II Block and Symbol Editors; however, the Quartus II software saves them as **.bdf** or **.bsf** files.

For information about creating and editing schematic designs, refer to the Quartus II Help.

Schematic entry methods do not support the synthesis directives or attributes described in this chapter.

State Machine Editor

The Quartus II software supports graphical state machine entry. To create a new finite state machine (FSM) design, on the File menu, click **New**. In the **New** dialog box, expand the **Design Files** list and choose **State Machine File**.

In the editor, you can use the State Machine Wizard to step you through the state machine creation. Click the **State Machine Wizard** icon. Specify the reset information, define the input ports, states, and transitions, and then define the output ports and output conditions. Click **Finish** to create the state machine diagram.

You can also create the state machine diagram using the editor GUI. Use the icons or right-click menu options to insert new input and output signals and create states in the schematic display. To specify transitions, select the **Transition Tool** and click on the source state, then drag the mouse to the destination state. Double-click on a transition to specify the transition equation, using a syntax that conforms to Verilog HDL. Double-click on a state to open the **State Properties** dialog box, where you can change the state name, specify whether it acts as the reset state, and change the incoming and outgoing transition equations.

To view and edit state machine information in a table format, click the **State Machine Table** icon.

The state machine diagram is saved as a State Machine File (**.smf**). When you have finished defining the state machine logic, create a Verilog HDL or VHDL design file by clicking the **Generate HDL File** icon. You can then instantiate the state machine in your design using any design entry language.

For more information about creating and editing state machine diagrams, refer to the Quartus II Help.

Design Libraries

By default, the Quartus II software compiles all design files into the work library. If you do not specify a design library, or if a file refers to a library that does not exist, or if the library does not contain a referenced design unit, the software searches the work library. This behavior allows the Quartus II software to compile most designs with minimal setup. (Creating separate custom design libraries is optional.)

To compile your design files into specific libraries (for example, when you have two or more functionally different design entities that share the same name), you can specify a destination library for each design file in various ways, as described in the following subsections:

- "Specifying a Destination Library Name in the Settings Dialog Box"
- "Specifying a Destination Library Name in the Quartus II Settings File or Using Tcl"

When the Quartus II compiler analyzes the file, it stores the analyzed design units in the file's destination library.

Beginning with the Quartus II software version 6.1, a design can contain two or more entities with the same name if they are compiled into separate libraries.

When compiling a design instance, the Quartus II software initially searches for the entity in the library associated with the instance (which is the work library if no other library is specified). If the entity definition is not found, the software searches for a unique entity definition in all design libraries. If more than one entity with the same name is found, the software generates an error. If your design uses multiple entities with the same name, you must compile the entities into separate libraries.

In VHDL, there are several ways to associate an instance with a particular entity, as described in "Mapping a VHDL Instance to an Entity in a Specific Library". In Verilog HDL, BDF schematic entry, AHDL, as well as VQM and EDIF netlists, use different libraries for each of the entities that have the same name, and compile the instantiation into the same library as the appropriate entity.

Specifying a Destination Library Name in the Settings Dialog Box

To specify a library name for one of your design files, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Files**. The **Files** page appears.
3. Select the file in the **File Name** list.
4. Click **Properties**.
5. In the **File Properties** dialog box, select the type of design file from the **Type** list.
6. Type the desired library name in the **Library** field.
7. Click **OK**.

Specifying a Destination Library Name in the Quartus II Settings File or Using Tcl

You can specify the library name with the `-library` option to the <*language type*>`_FILE` assignment in the Quartus II Settings File (**.qsf**) or with Tcl commands.

For example, the following **.qsf** file or Tcl assignments specify that the Quartus II software analyze **my_file.vhd** and store its contents (design units) in the VHDL library **my_lib**, and analyze the Verilog HDL file **my_header_file.h** and store its contents in a library called **another_lib**. Refer to Example 9–11.

Example 9–11. Specifying a Destination Library Name

```
set_global_assignment -name VHDL_FILE my_file.vhd -library my_lib
set_global_assignment -name VERILOG_FILE my_header_file.h -library another_lib
```

For more information about Tcl scripting, refer to "Scripting Support" on page 9–80.

Specifying a Destination Library Name in a VHDL File

You can use the `library` synthesis directive to specify a library name in your VHDL source file. This directive takes a single string argument: the name of the destination library. Specify the `library` directive in a VHDL comment prior to the context clause for a primary design unit (that is, a package declaration, an entity declaration, or a configuration), using one of the supported keywords for synthesis directives, that is, `altera`, `synthesis`, `pragma`, `synopsys`, or `exemplar`.

For more information about specifying synthesis directives, refer to "Synthesis Directives" on page 9–27.

The `library` directive overrides the default library destination **work**, the library setting specified for the current file through the **Settings** dialog box, any existing **.qsf** file setting, any setting made through the Tcl interface, or any prior `library` directive in the current file. The directive remains effective until the end of the file or the next `library` synthesis directive.

Example 9–12 uses the `library` synthesis directive to create a library called **my_lib** that contains the design unit `my_entity`.

Example 9–12. Using the Library Synthesis Directive

```
-- synthesis library my_lib
library ieee;
use ieee.std_logic_1164.all;
entity my_entity(...)
end entity my_entity;
```

You can specify a single destination library for all the design units in a given source file by specifying the library name in the **Settings** dialog box, editing the **.qsf** file, or using the Tcl interface. Using the `library` directive to change the destination VHDL library within a source file gives you the option of organizing the design units in a single file into different libraries, rather than just a single library.

The Quartus II software produces an error if you use the library directive within a design unit.

Mapping a VHDL Instance to an Entity in a Specific Library

The VHDL language provides a number of ways to map or bind an instance to an entity in a specific library, as described in the following subsections.

Direct Entity Instantiation

In the direct entity instantiation method, the instantiation refers to an entity in a specific library, as shown in Example 9–13.

Example 9–13. VHDL Code: Direct Entity Instantiation

```
entity entity1 is
port(...);
end entity entity1;

architecture arch of entity1 is
begin
inst: entity lib1.foo
port map(...);
end architecture arch;
```

Component Instantiation—Explicit Binding Instantiation

There is more than one mechanism for binding a component to an entity. In an explicit binding indication, you bind a component instance to a specific entity, as shown in Example 9–14.

© March 2009 Altera Corporation

Example 9–14. VHDL Code: Binding Instantiation

```
entity entity1 is
port(...);
end entity entity1;

package components is
component entity1 is
port map (...);
end component entity1;
end package components;

entity top_entity is
port(...);
end entity top_entity;

use lib1.components.all;
architecture arch of top_entity is
-- Explicitly bind instance I1 to entity1 from lib1
for I1: entity1 use entity lib1.entity1
port map(...);
end for;
begin
I1: entity1 port map(...);
end architecture arch;
```

Component Instantiation—Default Binding

If you do not provide an explicit binding indication, a component instance is bound to the nearest visible entity with the same name. If no such entity is visible in the current scope, the instance is bound to the entity in the library in which the component was declared. For example, if the component is declared in a package in library `MY_LIB`, an instance of the component is bound to the entity in library `MY_LIB`. The portions of code in Example 9–15 and Example 9–16 show this instantiation method.

Example 9–15. VHDL Code: Default Binding to the Entity in the Same Library as the Component Declaration

```
use mylib.pkg.foo; -- import component declaration from package “pkg”
in                 -- library “mylib”
architecture rtl of top
...
begin
-- This instance will be bound to entity “foo” in library “mylib”
inst: foo
port map(...);
end architecture rtl;
```

Example 9–16. VHDL Code: Default Binding to the Directly Visible Entity

```
use mylib.foo; -- make entity "foo" in library "mylib" directly visible
architecture rtl of top
component foo is
generic (...)
port (...);
end component;
begin
-- This instance will be bound to entity "foo" in library "mylib"
inst: foo
port map(...);
end architecture rtl;
```

Using Parameters/Generics

This section describes how parameters (called generics in VHDL) are supported in the Quartus II software, and how you can pass these parameters between different design languages.

You can enter default parameter values for your design in the **Default Parameters** box in the **Analysis & Synthesis Settings** page of the **Settings** dialog box. Default parameters allow you to specify the parameter overrides for your top-level entity. In AHDL, parameters are inherited, so any default parameters apply to all AHDL instances in the design. You can also specify parameters for instantiated modules in a **.bdf** file. To modify parameters in a **.bdf** instance, double-click on the parameter value box for the instance symbol, or right-click on the symbol and choose **Properties**, then click the **Parameters** tab. For these GUI-based entry methods, information about how parameter values are interpreted, and recommendations about the format you should use, refer to "Setting Default Parameter Values and BDF Instance Parameter Values".

You can specify parameters for instantiated modules in your design source files, using the syntax provided for that language. Some designs instantiate entities in a different language; for example, they might instantiate a VHDL entity from a Verilog HDL design file. You can pass parameters or generics between VHDL, Verilog HDL, AHDL, and BDF schematic entry, and from EDIF or VQM to any of these languages. In most cases, you do not have to do anything special to pass parameters from one language to another. However, in some cases you might have to specify the type of parameter you are passing. In those cases, you should follow certain guidelines to ensure that the parameter value is interpreted correctly. Refer to "Passing Parameters Between Two Design Languages" on page 9–18 for parameter type rules.

Setting Default Parameter Values and BDF Instance Parameter Values

Default parameter values and BDF instance parameter values do not have an explicitly declared type. In most cases, the Quartus II software can correctly infer the type from the value without ambiguity. For example, "`ABC`" is interpreted as a string, `123` as an integer, and `15.4` as a floating-point value. In other cases, such as when the instantiated subdesign language is VHDL, the Quartus II software uses the type of the parameter/generic in the instantiated entity to determine how to interpret the value, so that a value of `123` is interpreted as a string if the VHDL parameter is of type

string. In addition, you can set the parameter value in a format that is legal in the language of the instantiated entity. For example, to pass an unsized bit literal value from BDF to Verilog HDL, you can use `'1` as the parameter value, and to pass a 4-bit binary vector from BDF to Verilog HDL, you can use `4'b1111` as the parameter value.

In a few cases, the Quartus II software cannot infer the correct type of parameter value. To avoid ambiguity, specify the parameter value in a type-encoded format where the first or first and second character of the parameter indicate the type of the parameter, and the rest of the string indicates the value in a quoted sub-string. For example, to pass a binary string `1001` from BDF to Verilog HDL, you cannot simply use the value `1001`, because the Quartus II software interprets it as a decimal value. You also cannot use the string `"1001"`, because the Quartus II software interprets it as an ASCII string. You must use the type-encoded string `B"1001"` for the Quartus II software to interpret the parameter value correctly. Table 9–1 provides a list of valid parameter strings and shows how they are interpreted within the Quartus II software. Altera recommends using the type-encoded format only when necessary to resolve ambiguity.

Table 9–1. Valid Parameter Strings and How They are Interpreted

Parameter String	Quartus II Parameter Type, Format, and Value
`S"abc"`, `s"abc"`	String value abc
`"abc123"`, `"123abc"`	String value abc123 or 123abc
`F"12.3"`, `f"12.3"`	Floating point number 12.3
`-5.4`	Floating point number -5.4
`D"123"`, `d"123"`	Decimal number 123
`123`, `-123`	Decimal number 123, -123
`X"ff"`, `H"ff"`	Hexadecimal value FF
`Q"77"`, `O"77"`	Octal value 77
`B"1010"`, `b"1010"`	Unsigned binary value 1010
`SB"1010"`, `sb"1010"`	Signed binary value 1010
`R"1"`, `R"0"`, `R"X"`, `R"Z"`, `r"1"`, `r"0"`, `r"X"`, `r"Z"`	Unsized bit literal
`E"apple"`, `e"apple"`	Enum type, value name is apple
`P"1 unit"`	Physical literal, the value is (1, unit)
`A(...)`, `a(...)`	Array type or record type, whose content is determined by the string (...)

Beginning in Quartus II software version 8.1, you can select the parameter type using the pull-down list in the **Parameter** tab of the **Symbol Properties** dialog box. You can select the parameter types for global parameters or global constants. The Quartus II software supports the following parameter types:

- **Unsigned Integer**
- **Signed Integer**
- **Unsigned Binary**
- **Signed Binary**
- **Octal**

- **Hexadecimal**
- **Float**
- **Enum**
- **String**
- **Boolean**
- **Char**
- **Untyped/Auto**

If you do not specify the parameter type, the Quartus II software interprets the parameter value and defines the parameter type. Altera recommends specifying parameter type by selecting from the pull-down list to avoid ambiguity.

If you open a **.bdf** file in the Quartus II software version 8.1 and above, the software automatically updates the parameter types of old symbol blocks by interpreting the parameter value based on the language-independent format. If the parameter value type is not recognized, the parameter type is set as untyped. You can specify the parameter type as described in the preceding list.

Passing Parameters Between Two Design Languages

When passing a parameter between two different languages, a design block that is higher in the design hierarchy instantiates a lower-level subdesign block and provides parameter information. It is essential that the parameter be correctly interpreted by the subdesign language (the design entity that is instantiated). Based on the information provided by the higher-level design and the value format, and sometimes by the parameter type of the subdesign entity, the Quartus II software interprets the type and value of the passed parameter.

When passing a parameter whose value is an enumerated type value or literal from a language that does not support enumerated types to one that does (for example, from Verilog HDL to VHDL), it is essential that the enumeration literal is spelled correctly in the higher-level design. The parameter value is passed as a string literal, and it is up to the language of the lower-level design to correctly convert the string literal into the correct enumeration literal.

If the lower-level language is SystemVerilog, it is essential that the `enum` value is used in the correct case. In SystemVerilog, it is recommended that two enumeration literals do not only differ in case. For example, `enum {item, ITEM}` is not a good choice of item names because these names can create confusion among users and it is more difficult to pass parameters from case-insensitive HDLs, such as VHDL.

Arrays have different support in different design languages. For details about the array parameter format, refer to the **Parameter** section in the Analysis & Synthesis Report of a design that contains array parameters or generics.

The following code shows examples of passing parameters from one design entry language to a subdesign written in another language. Example 9–17 shows a VHDL subdesign that is instantiated into a top-level Verilog HDL design in Example 9–18. Example 9–19 shows a Verilog HDL subdesign that is instantiated in a top-level VHDL design in Example 9–20.

Example 9–17. VHDL Parameterized Subdesign Entity

```
type fruit is (apple, orange, grape);
entity vhdl_sub is
generic (
name : string := "default",
width : integer := 8,
number_string : string := "123",
f : fruit := apple,
binary_vector : std_logic_vector(3 downto 0) := "0101",
signed_vector : signed (3 downto 0) := "1111");
```

Example 9–18. Verilog HDL Top-Level Design Instantiating and Passing Parameters to VHDL Entity from Example 9–17

```
vhdl_sub inst (...);
defparam inst.name = "lower";
defparam inst.width = 3;
defparam inst.num_string = "321";
defparam inst.f = "grape"; // Must exactly match enum value
defparam inst.binary_vector = 4'b1010;
      defparam inst.signed_vector = 4'sb1010;
```

Example 9–19. Verilog HDL Parameterized Subdesign Module

```
module veri_sub (...)
parameter name = "default";
parameter width = 8;
parameter number_string = "123";
parameter binary_vector = 4'b0101;
parameter signed_vector = 4'sb1111;
```

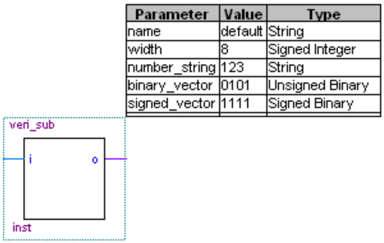
Example 9–20. VHDL Top-Level Design Instantiating and Passing Parameters to the Verilog HDL Module from Example 9–19

```
inst:veri_sub
generic map (
name => "lower",
width => 3,
number_string => "321"
binary_vector = "1010"
signed_vector = "1010")
```

To use an HDL subdesign such as the one shown in Example 9–19 in a top-level BDF design, you must first generate a symbol for the HDL file, as shown in Figure 9–2. Open the HDL file in the Quartus II software, and then, on the File menu, point to **Create/Update** and click **Create Symbol Files for Current File**.

To modify parameters on a BDF instance, double-click on the parameter value box for the instance symbol, or right-click on the symbol and choose **Properties**, then click the **Parameters** tab.

Figure 9–2. BDF Top-Level Design Instantiating and Passing Parameters to the Verilog HDL Module from Example 9–19.



Parameter	Value	Type
name	default	String
width	8	Signed Integer
number_string	123	String
binary_vector	0101	Unsigned Binary
signed_vector	1111	Signed Binary

Incremental Compilation

The incremental compilation feature in the Quartus II software manages a design hierarchy for incremental design by allowing you to divide the design into multiple partitions. Incremental compilation ensures that when a design is compiled, only those partitions of the design that have been updated are resynthesized, reducing compilation time and runtime memory usage. This also means that node names are maintained during synthesis for all registered and combinational nodes in unchanged partitions. You can perform incremental synthesis by setting the Netlist Type for all design partitions to **Post-Synthesis**.

You can also preserve the placement (and optionally routing) information for unchanged partitions. This feature allows you to preserve performance of unchanged blocks in your design and reduces the time required for placement and routing, which significantly reduces your design compilation time.

Partitions for Preserving Hierarchical Boundaries

A design partition represents a portion of the design that you want to synthesize and fit incrementally.

The **Preserve Hierarchical Boundary** logic option is available only in Quartus II software versions 8.1 and earlier. Incremental compilation maintains the hierarchical boundaries of design partitions, so you should use design partitions if you want to preserve hierarchical boundaries through the synthesis and fitting process.

Beginning with Quartus II software version 9.0, if you want to preserve the **Optimization Technique** and **Restructure Multiplexers** logic options set in any entity, you must create new partitions for the particular entity instead of using the **Preserve Hierarchical Boundary** logic option. If you have settings applied to specific existing design hierarchies, particularly those created in the Quartus II software versions before 9.0, you must create a design partition for the design hierarchy so that synthesis can optimize the design instance independently and preserve the hierarchical boundaries. Similarly, if you are performing formal verification, you must use partitions with incremental compilation to ensure that no optimizations occur across specific design hierarchies.

Parallel Synthesis

The **Parallel Synthesis** option is one of the Analysis and Synthesis options that you can use to reduce compilation time for synthesis. The feature enables the Quartus II software to use multi-processors to synthesize multiple partitions in parallel.

This feature is available only if the following requirements are met:

- The number of processors allowed is greater than 1

You can specify the maximum number of processors allowed under **Parallel Compilation** options in the **Compilation Process Settings** page of the **Settings** dialog box.

- Incremental compilation is enabled and your design has two or more partitions
- **Timing Driven Synthesis** is not enabled
- **Physical Synthesis** is not enabled
- **Parallel Synthesis** is enabled using one of the following procedures:

To enable parallel synthesis, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click **Analysis & Synthesis Settings** and click **More Settings** to select **Parallel Synthesis**.

You can also set the **Parallel Synthesis** option using the following Tcl command:

```
set_global_assignment -name parallel_synthesis on
```

You can view all messages generated during the parallel synthesis in the Message console. Messages from different partitions are interleaved at runtime, but the **Partition Column** displays the partition ID of the partition that has a message. After compilation, you can sort the messages by **Partition Column**—effectively grouping all the messages from a particular partition. To display the partition column, right click on the message console, point to **Message Column** and select **Show Partition Column**. You can also display the **Partition** column on the Tools menu, by clicking **Options** and selecting **Messages** in the **Category** list. In the Messages page, turn on **Show the Partition column**.

If you use the command line, you can differentiate among the interleaved messages by turning on the **Show partition that generated the message** option in **Messages** page. This option shows the partition ID in parenthesis for each message.

Quartus II eXported Partition (.qxp) File as Source

You can use a Quartus II eXported Partition (**.qxp**) file as a source file beginning with Quartus II software version 8.1. The **.qxp** file contains the precompiled design netlist exported from another Quartus II project, or from a design partition within the project, and fully defines the entity. Project team members or IP providers can use a **.qxp** file to send their design to the project lead, instead of sending the original HDL source code. Using this file preserves the previous compilation results and instance-specific assignments. Not all global assignments can be used in a different Quartus II project. You can override the assignments for the entity in the **.qxp** file by applying assignments in the full top-level project.

A **.qxp** file instance that is not assigned as a design partition does not preserve placement and routing results. If you want to preserve the placement (and optionally routing) results from another project or compilation, you must import a post-fitting **.qxp** file into a design partition in your project using the bottom-up incremental compilation flow.

Perform the following steps to create a **.qxp** file:

1. On the Project menu, click **Export Design Partition**.
2. In the **Export file** box, type the name of the **.qxp** file. By default, the directory path and file name are the same as the current project.
3. You can also select the Partition hierarchy to export. By default, the Top partition (the entire project) is exported, but you can choose to export the compilation results of any partition hierarchy in the project.
4. Under **Netlist to export**, select either **Post-fit netlist** or **Post-synthesis netlist**. The default is **Post-fit netlist**. For post-fit netlists, turn on or off the **Export routing** option as required.
5. Click **OK**. The Quartus II software creates the **.qxp** file in the specified directory.

The Quartus II software adds the file into the project and **.qxp** file into a specific library. The design entity in the **.qxp** file can also be instantiated multiple times in the design.

For more information about exporting design partitions and using **.qxp** files, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*

Quartus II Synthesis Options

The Quartus II software offers a number of options to help you control the synthesis process and achieve optimal results for your design. "Setting Synthesis Options" on page 9–24 describes the **Analysis & Synthesis Settings** page of the **Settings** dialog box, where you can set the most common global settings and options, and defines the following three types of synthesis options: Quartus II logic options, synthesis attributes, and synthesis directives. The other subsections describe the following common synthesis options in the Quartus II software, and provide HDL examples of how to use each option, where applicable:

- Major Optimization Settings
  - "Optimization Technique" on page 9–28
  - "Speed Optimization Technique for Clock Domains" on page 9–28
  - "PowerPlay Power Optimization" on page 9–31
  - "Restructure Multiplexers" on page 9–34
  - "Synthesis Effort" on page 9–35

- Settings Related to Timing Constraints
  - "Optimization Technique" on page 9–28
  - "Speed Optimization Technique for Clock Domains" on page 9–28
  - "Auto Gated Clock Conversion" on page 9–29
  - "Timing-Driven Synthesis" on page 9–30
  - "SDC Constraint Protection" on page 9–31
- State Machine Settings and Enumerated Types
  - "State Machine Processing" on page 9–36
  - "Manually Specifying State Assignments Using the syn_encoding Attribute" on page 9–37
  - "Manually Specifying Enumerated Types Using the enum_encoding Attribute" on page 9–39
  - "Safe State Machines" on page 9–41
- Register Power-Up Settings
  - "Power-Up Level" on page 9–42
  - "Power-Up Don't Care" on page 9–43
- Controlling, Preserving, Removing, and Duplicating Logic and Registers
  - "Limiting DSP Block Usage in Partitions" on page 9–32
  - "Remove Duplicate Registers" on page 9–44
  - "Remove Redundant Logic Cells" on page 9–44
  - "Preserve Registers" on page 9–44
  - "Disable Register Merging/Don't Merge Register" on page 9–45
  - "Noprune Synthesis Attribute/Preserve Fan-out Free Register Node" on page 9–46
  - "Keep Combinational Node/Implement as Output of Logic Cell" on page 9–46
  - "Don't Retime, Disabling Synthesis Netlist Optimizations" on page 9–47
  - "Don't Replicate, Disabling Synthesis Netlist Optimizations" on page 9–48
  - "Maximum Fan-Out" on page 9–49
  - "Controlling Clock Enable Signals with Auto Clock Enable Replacement and direct_enable" on page 9–50
  - "Auto Gated Clock Conversion" on page 9–29
  - To preserve design hierarchy, refer to "Partitions for Preserving Hierarchical Boundaries" on page 9–20

- Megafunction Inference Options
  - "Megafunction Inference Control" on page 9–51
  - "RAM Style and ROM Style—for Inferred Memory" on page 9–53
  - "Turning Off Add Pass-Through Logic to Inferred RAMs/ no_rw_check Attribute Setting" on page 9–54
  - "RAM Initialization File—for Inferred Memory" on page 9–57
  - "Multiplier Style—for Inferred Multipliers" on page 9–58
- Controlling Synthesis with Other Synthesis Directives
  - "Full Case" on page 9–60
  - "Parallel Case" on page 9–61
  - "Translate Off and On / Synthesis Off and On" on page 9–62
  - "Ignore translate_off and synthesis_off Directives" on page 9–63
  - "Read Comments as HDL" on page 9–63
- Specifying I/O-Related Assignments
  - "Use I/O Flipflops" on page 9–64
  - "Specifying Pin Locations with chip_pin" on page 9–65
- Setting Quartus II Logic Options in Your HDL Source Code
  - "Using altera_attribute to Set Quartus II Logic Options" on page 9–66

Setting Synthesis Options

You can set synthesis options in the **Settings** dialog box, or with logic options in the Quartus II software, or you can use synthesis attributes and directives within the HDL source code.

Analysis & Synthesis Settings Page of the Settings Dialog Box

On the Assignments menu, click **Settings**. The **Settings** dialog box appears. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page allows you to set global synthesis options that apply to the entire project. These options are described in later subsections.

Beginning with Quartus II software version 9.0, some of the advanced synthesis settings can be set in the **Physical Synthesis Optimization** page under **Compilation Process Settings**.

For more information about Physical Synthesis options, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

© March 2009 Altera Corporation

Quartus II Logic Options

Quartus II logic options control many aspects of the synthesis and place-and-route process. To set logic options in the Quartus II GUI, on the Assignments menu, click **Assignment Editor**. You can also use a corresponding Tcl command. Quartus II logic options allow you to set instance or node-specific assignments without editing the source HDL code. Logic options can be used with all design entry languages supported by the Quartus II software.

For more information about using the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

Synthesis Attributes

The Quartus II software supports synthesis attributes for Verilog HDL and VHDL, also commonly called pragmas. These attributes are not standard Verilog HDL or VHDL commands; synthesis tools use attributes to control the synthesis process in a particular manner. Attributes always apply to a specific design element, and are applied in the HDL source code. Some synthesis attributes are also available as Quartus II logic options via the Quartus II GUI or with Tcl. Each attribute description in this chapter indicates whether there is a corresponding setting or logic option that can be set in the GUI; some attributes can be specified only with HDL synthesis attributes.

Attributes specified in your HDL code are not visible in the Assignment Editor or in the **.qsf** file. Assignments or settings made through the Quartus II GUI, the **.qsf** file, or the Tcl interface take precedence over assignments or settings made with synthesis attributes in your HDL code. The Quartus II software generates warning messages if invalid attributes are found, but does not generate an error or stop the compilation. This behavior is required because attributes are specific to various design tools, and attributes not recognized in the Quartus II software might be intended for a different EDA tool. The Quartus II software lists the attributes specified in your HDL code in the Source assignments table in the Analysis & Synthesis report.

The Verilog-2001, SystemVerilog, and VHDL language definitions provide specific syntax for specifying attributes, but in Verilog-1995, you must embed attribute assignments in comments. You can enter attributes in your code using the syntax in Example 9–21 through Example 9–24, where *<attribute>*, *<attribute type>*, *<value>*, *<object>*, and *<object type>* are variables, and the entry in brackets is optional. The examples in this chapter demonstrate each syntax form.

Verilog HDL is case-sensitive; therefore, synthesis attributes are also case-sensitive.

Example 9–21. Synthesis Attributes in Verilog-1995

```
// synthesis <attribute> [ = <value> ]
or
/* synthesis <attribute> [ = <value> ] */
```

Verilog-1995 comment-embedded attributes, as shown in Example 9–21, must be used as a suffix to (that is, placed after) the declaration of an item and must appear before the semicolon when one is required.

You cannot use the open one-line comment in Verilog HDL when a semicolon is required at the end of the line, because it is not clear to which HDL element the attribute applies. For example, you cannot make an attribute assignment such as `reg r; // synthesis` *<attribute>* because the attribute could be read as part of the next line.

To apply multiple attributes to the same instance in Verilog-1995, separate the attributes with spaces, as follows:

`//synthesis` *<attribute1>* [= *<value>*] *<attribute2>* [= *<value>*]

For example, to set the `maxfan` attribute to `16` (Refer to "Maximum Fan-Out" on page 9–49 for details) and set the `preserve` attribute (refer to "Preserve Registers" on page 9–44 for details) on a register called `my_reg`, use the following syntax:

```
reg my_reg /* synthesis maxfan = 16 preserve */;
```

In addition to the `synthesis` keyword shown above, the keywords `pragma`, `synopsys`, and `exemplar` are supported for compatibility with other synthesis tools. The keyword `altera` is also supported, which allows you to add synthesis attributes that will be recognized only by Quartus II integrated synthesis and not by other tools that recognize the same synthesis attribute.

Because formal verification tools do not recognize the `exemplar`, `pragma`, and `altera` keywords, avoid using these attribute keywords when using formal verification.

Example 9–22. Synthesis Attributes in Verilog-2001 and SystemVerilog

```
(* <attribute> [ = <value> ] *)
```

Verilog-2001 attributes, as shown in Example 9–22, must be used as a prefix to (that is, placed before) a declaration, module item, statement, or port connection, and used as a suffix to (that is, placed after) an operator or a Verilog HDL function name in an expression.

Because formal verification tools do not recognize the syntax, the Verilog-2001 attribute syntax is not supported when using formal verification.

To apply multiple attributes to the same instance in Verilog-2001 or SystemVerilog, separate the attributes with commas, as shown in Example 9–23:

Example 9–23. Applying Multiple Attributes

```
(* <attribute1> [ = <value1>], <attribute2> [ = <value2> ] *)
```

For example, to set the `maxfan` attribute to `16` (refer to "Maximum Fan-Out" on page 9–49 for details) and set the `preserve` attribute (refer to "Preserve Registers" on page 9–44 for details) on a register called `my_reg`, use the following syntax:

```
(* preserve, maxfan = 16 *) reg my_reg;
```


© March 2009 Altera Corporation

Example 9–24. Synthesis Attributes in VHDL

```
attribute <attribute> : <attribute type> ;
attribute <attribute> of <object> : <object type> is <value>;
```

VHDL attributes, as shown in Example 9–24, declare the attribute type and then apply it to a specific object. For VHDL designs, all supported synthesis attributes are declared in the `altera_syn_attributes` package in the **Altera** library. You can call this library from your VHDL code to declare the synthesis attributes, as follows:

```
LIBRARY altera;
USE altera.altera_syn_attributes.all;
```

Synthesis Directives

The Quartus II software supports synthesis directives, also commonly called compiler directives or pragmas. You can include synthesis directives in Verilog HDL or VHDL code as comments. These directives are not standard Verilog HDL or VHDL commands; synthesis tools use directives to control the synthesis process in a particular manner. Directives do not apply to a specific design node but change the behavior of the synthesis tool from the point where they occur in the HDL source code. Other tools, such as simulators, ignore these directives and treat them as comments.

You can enter synthesis directives in your code using the syntax shown in Example 9–25 and Example 9–26, where *<directive>* and *<value>* are variables, and the entry in brackets is optional. Notice that for synthesis directives there is no = sign before the value; this is different than the syntax for synthesis attributes. The examples in this chapter demonstrate each syntax form.

Verilog HDL is case-sensitive; therefore, all synthesis directives are also case-sensitive.

Example 9–25. Verilog HDL Code: Synthesis Directives

```
// synthesis <directive> [ <value> ]
or
/* synthesis <directive> [ <value> ] */
```

Example 9–26. VHDL Code: Synthesis Directives

```
-- synthesis <directive> [ <value> ]
```

In addition to the `synthesis` keyword shown above, the `pragma`, `synopsys`, and `exemplar` keywords are supported in both Verilog HDL and VHDL for compatibility with other synthesis tools. The keyword `altera` is also supported, which allows you to add synthesis directives that are recognized only by Quartus II integrated synthesis and not by other tools that recognize the same synthesis directives.

Because formal verification tools ignore keywords `exemplar`, `pragma`, and `altera`, avoid using these directive keywords when you are using formal verification to prevent mismatches with the Quartus II results.

Optimization Technique

The **Optimization Technique** logic option specifies the goal for logic optimization during compilation; that is, whether to attempt to achieve maximum speed performance or minimum area usage, or a balance between the two. Table 9–2 lists the settings for this logic option, which you can apply only to a design entity. You can also set this logic option for your whole project on the **Analysis & Synthesis Settings** page in the **Settings** dialog box. If you want to set this logic option for an entity, you must create a design partition for the entity before setting the **Optimization Technique** logic option. Beginning in Quartus II version 9.0, this option is ignored when set on an entity that is not a design partition.

Table 9–2. Optimization Technique Settings

Setting	Description
Area	The compiler makes the design as small as possible to minimize resource usage.
Speed	The compiler chooses a design implementation that has the fastest f_{MAX}.
Balanced *(1)*	The compiler maps part of the design for area and part for speed, providing better area utilization than optimizing for speed, with only a slightly slower f_{MAX} than optimizing for speed.

Note to Table 9–2:

(1) The balanced optimization technique is not supported for all device families.

The default setting varies by device family and is generally optimized for the best area/speed trade-off. Results are design-dependent and vary depending on which device family you use.

Speed Optimization Technique for Clock Domains

The **Speed Optimization Technique for Clock Domains** logic option specifies that all combinational logic in or between the specified clock domain(s) is optimized for speed.

When this option is set on a particular clock signal, all the logic in this clock domain is optimized for speed during synthesis. The remainder of the design in other clock domains is synthesized with the project-wide **Optimization Technique** that is set in the **Analysis & Synthesis Settings** page. The option can also be set from one clock to another clock signal, in which case the logic in paths from registers in the first clock domain to registers in the second clock domain are synthesized for speed. The advantage of using this option over the project-wide setting to optimize for speed is that there is less penalty to the area of the design because a smaller part of the circuit is optimized for speed. This can also have a positive effect on clock speed. This option also has an advantage over setting the **Optimization Technique** on a design entity because that option forces the hierarchical blocks to be synthesized separately. Doing so can increase area and decrease performance due to the lack of optimizations across hierarchies. The **Speed Optimization Technique for Clock Domains** option does not treat hierarchical entities separately, and can optimize across hierarchical boundaries for logic within the same clock domain.

This option is useful if you have one or more clock domains that do not meet your timing requirements. When there are failing paths within a clock domain, the option can be set on the clock of that clock domain. When there are failing paths between clock domains, the option can be set from one clock domain to the other clock domain.

This option is available for the following device families: Arria® GX, Stratix® series, Cyclone® series, HardCopy® II, HardCopy Stratix, and MAX® II.

Auto Gated Clock Conversion

Clock gating is a common optimization technique used in ASIC designs to minimize power consumption. You can use the **Auto Gated Clock Conversion** option to optimize your prototype ASIC designs by converting gated clocks into clock enables when you use FPGAs in your ASIC prototyping. The automatic conversion of gated clocks to clock enables is more efficient than manually modifying source code. However, this feature should not be used when migrating FPGA designs to HardCopy ASICs. The **Auto Gated Clock Conversion** option automatically converts qualified gated clocks (base clocks as defined in the SDC assignments) to clock enables. To use **Auto Gated Clock Conversion**, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. The **More Analysis & Synthesis Settings** dialog box appears.
4. Under **Option**, in the **Name** list, select **Auto Gated Clock Conversion** and in the **Setting** list, select **On**.
5. Click **OK**.
6. Click **OK** again.

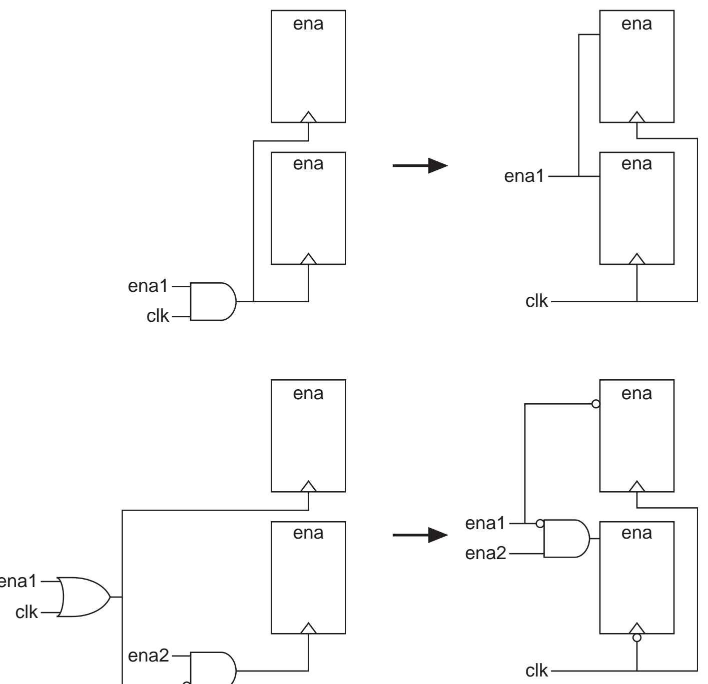
This feature is available only for the TimeQuest Timing Analyzer and supports the following device families: Arria II GX, Arria GX, Stratix series (except for Stratix) and Cyclone series (except for Cyclone), HardCopy II, and MAX II devices.

The gated clock conversion occurs when the following conditions are met:

- Only one base clock drives a gated-clock
- For one set of gating input values, the value output of the gated clock remains constant and does not change as the base clock changes
- For one value of the base clock, changes in the gating inputs do not change the value output for the gated clock

The feature supports combinational gates in clock gating network. Figure 9–3 shows examples of gated clock conversions.

Figure 9–3. Gated Clock Conversion



This feature does not support registers in RAM, DSP blocks, or I/O related WYSIWYG. The gated clock conversion does not support multiple design partitions from incremental compilation where the gated clock and base clock are not in the same hierarchical partition because the gated-clock conversion cannot trace the base clock from the gated clock. Thus, base clocks and gated clocks must be in the same hierarchical design partition. If a gated clock that is derived from a root gated clock of a multiple cascaded gated clock cannot be converted, the whole gated clock tree will not be converted, because each conversion is based on a gated clock tree instead of every gated clock.

The **Info** tab in the Messages window lists all the converted gated clocks. You can view a list of converted and non-converted gated clocks from the Compilation Report under the **Optimization Results** of the Analysis & Synthesis Report. The reasons for non-converted gated clocks are listed in the **Gated Clock Conversion Details** table.

Timing-Driven Synthesis

The **Timing-Driven Synthesis** option specifies whether synthesis should use the design's SDC timing constraints to better optimize the circuit. This feature enables synthesis to take into account the SDC timing constraints to focus on the truly critical parts of the design when optimizing for performance. Altera recommends using **Timing-Driven Synthesis** with **Optimization Technique Balanced**. When you turn on **Timing-Driven Synthesis**, Synthesis improves logic depth on parts of the design that require meeting performance requirements at the cost of area increase. Compared to **Optimization Technique** option for speed, **Timing-Driven Synthesis** gets similar performance but saves area.

To use the **Timing-Driven Synthesis** option, perform the following steps:

1. On the Assignment menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. In the **Analysis & Synthesis Settings** page, select **Timing-Driven Synthesis**.

The feature is available only for the TimeQuest Timing Analyzer and supports Arria II GX, Arria GX, Stratix series (except Stratix devices) and Cyclone series (except Cyclone devices), and HardCopy II devices. Altera recommends that you select a specific device for timing-driven synthesis to have the most accurate timing information. When auto device is selected, timing-driven synthesis uses the smallest device for the selected family to obtain timing information.

SDC Constraint Protection

The **SDC constraint protection** option allows you to preserve timing constraints when you use the TimeQuest Timing Analyzer. The feature checks on register merging and retiming. It prevents register merging on registers with incompatible SDC constraints and prevents register retiming on constrained registers. It helps maintain the validity of SDC constraints throughout compilation. To use the **SDC constraint protection** option, perform the following steps:

1. On the Assignment menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. The **More Settings** dialog box appears.
4. Under **Option**, in the **Name** list, select **SDC constraint protection** and in the **Setting** list, select **On**.

This feature is available only for the TimeQuest Timing Analyzer and supports the following device families: Arria GX, Stratix series (except Stratix devices), Cyclone series (except Cyclone devices), HardCopy II, and MAX II devices.

PowerPlay Power Optimization

This logic option controls the power-driven compilation setting of Analysis and Synthesis and determines how aggressively Analysis and Synthesis optimizes the design for power. On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**. This displays the **Analysis & Synthesis Settings** page. The following three settings are available for the **PowerPlay Power Optimization** option:

- **Off**—Analysis and Synthesis does not perform any power optimizations.
- **Normal Compilation**—Analysis and Synthesis performs power optimizations, without reducing design performance.
- **Extra Effort**—Analysis and Synthesis performs additional power optimizations, which can reduce design performance.

This logic option is available for the following device families: Arria GX, Stratix series, Cyclone series, HardCopy II, and MAX II.

For more information about optimizing your design for power utilization, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*. For information about analyzing your power results, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Limiting DSP Block Usage in Partitions

One important step of Analysis and Synthesis is resource balancing. In this step, Quartus II integrated synthesis looks at the digital signal processing (DSP) block used in the design and balances it against the resources available in the targeted device that are converting the DSP blocks that cannot fit in the device into equivalent logic. For incremental compilation, each partition has a separate balancing step.

By default, the Quartus II integrated synthesis looks at the targeted device information to find out the number of DSP blocks available for use. However, in incremental compilation, each partition looks at the device information independently and consequently assumes that it has all the DSP blocks in the device available for use. This can result in over-allocation of DSP blocks in the design, which means that the total number of DSP blocks used by all the partitions is greater than the number of DSP blocks available in the device. This can eventually lead to a no-fit error during the fitting process.

To avoid this, Altera recommends that you set the **Maximum DSP Block Usage** assignment on each partition to manually limit the number of DSP blocks used. You can set this assignment on a partition using the Assignment Editor by selecting the **Maximum DSP Block Usage** assignment, and setting it on the root of a partition. Set any positive integer as the value of this assignment. If this assignment is set on a name other than a partition root, the Quartus II integrated synthesis gives an error.

The **Maximum DSP Block Usage** assignment is available only for supported device families. Refer to the Quartus II Help for a list of the devices.

For more information about using the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook.*

You can also set this assignment globally as a project-wide option by performing the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. The **More Analysis & Synthesis Settings** dialog box appears.
4. From the pull-down menu, point to **Maximum DSP Block Usage**, and from the **Settings** pull-down menu, select your desired value.

The partition-specific assignment overrides the global assignment, if any. However, each partition that does not have a partition-specific **Maximum DSP Block Usage** assignment limits the number of the DSP blocks to the value set by the global assignment. This can also lead to over-allocation of DSP blocks. Therefore, Altera recommends that you always set this assignment on each partition when you use the incremental compilation.

Manually limiting the DSP blocks usage is also useful for HardCopy II device migration, where the number of DSP blocks that can be implemented in a HardCopy II device is more than the number of DSP blocks that can be implemented in its equivalent Stratix II device.

In Quartus II software version 8.1 and later, the floorplan aware synthesis feature enables you to use LogicLock regions to define resource allocation for DSPs and RAMs before setting the maximum resource allocation assignment.

For more information about using LogicLock regions to create a floorplan for incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*, or refer to the Quartus II Help.

Altera recommends that you always use LogicLock assignments first before setting the maximum resource allocation assignments per partition. However, this recommendation might not affect the resource balancing if you manually assign nodes in a partition to different LogicLock regions and if there are some unassigned nodes which fall in the root LogicLock region where nodes are often from more than one partition. Thus, you can move the unassigned nodes to the defined LogicLock regions in the respective partitions and use the floorplan aware synthesis feature for better DSP and RAM balancing.

The floorplan aware synthesis feature is turned on by default. If you do not want the software to consider the LogicLock floorplan constraints when performing DSP and RAM balancing, you can turn off the floorplan aware synthesis feature. Set the **Use LogicLock Constraints During Resource Balancing** option to **Off** in the **Analysis & Synthesis Settings** page by clicking **More Settings**.

DSP balancing converts extra DSP blocks in the design into equivalent logic to meet Fitter requirements where the number of DSP blocks in design is less than or equal to the number of DSP blocks available. RAM balancing converts RAMs from one RAM type to another to meet Fitter requirements where the RAM block utilization of each RAM type is within limits of the available blocks for each RAM type. The floorplan aware synthesis option also allows you to specify maximum resources for different RAM types, such as **Maximum Number of M4K/M9K Memory Blocks**, **Maximum Number of M512 Memory Blocks**, or **Maximum Number of M-RAM/M144K Memory Blocks**.

You can specify the maximum DSP and RAM resource allocation by performing the following steps:

1. On the Assignment menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. The **More Settings** dialog box appears.
4. Under **Option**, in the **Name** list, select **Maximum DSP Block Usage** or **Maximum Number** <*block type*> **Memory Blocks** and specify the resource count in the **Setting** list.
5. Click **OK**.
6. Click **OK**.

HardCopy II devices have limited RAM blocks and there is no assignment to limit the RAM blocks usage in the HardCopy II device migration. Thus, Altera recommends that the maximum resource options are set to the default value of **-1 (UNLIMITED)** for the migration flow.

You can view the DSP and RAM block usage after balancing from the Compilation Report.

Restructure Multiplexers

This option specifies whether the Quartus II software should extract and optimize buses of multiplexers during synthesis.

This option is useful if your design contains buses of fragmented multiplexers. This option restructures multiplexers more efficiently for area, allowing the design to implement multiplexers with a reduced number of LEs or ALMs. This option is available for the following device families: Arria GX, Stratix series, Cyclone series, HardCopy II, and MAX II.

The **Restructure Multiplexers** option works on entire trees of multiplexers. Multiplexers may arise in different parts of the design through Verilog HDL or VHDL constructs such as the "`if`," "`case`," or "`?:`" statements. When multiplexers from one part of the design feed multiplexers in another part of the design, trees of multiplexers are formed. Multiplexer buses occur most often as a result of multiplexing together vectors in Verilog HDL, or `STD_LOGIC_VECTOR` signals in VHDL. The **Restructure Multiplexers** option identifies buses of multiplexer trees that have a similar structure. When it is turned on, the **Restructure Multiplexers** option optimizes the structure of each multiplexer bus for the target device to reduce the overall amount of logic used in the design.

Results of the multiplexer optimizations are design dependent, but area reductions as high as 20% are possible. The option can negatively affect your design's f_{MAX}.

Table 9–3 lists the settings for the logic option, which you can apply only to a design entity individual node, or to an entity that is a design partition. Beginning in Quartus II version 9.0, this option is only valid when set on an entity that is a design partition. You can also specify this option on the **Analysis & Synthesis Settings** page in the **Settings** dialog box for your whole project by clicking **More Settings** and setting the option value.

Table 9–3. Restructure Multiplexer Settings

Setting	Description
On	Enables multiplexer restructuring to minimize your design area. This setting can reduce the f_{MAX}.
Off	Disables multiplexer restructuring to avoid possible reductions in f_{MAX}.
Auto (Default)	Allows the compiler to determine whether to enable the option based on your other Quartus II synthesis settings. The option is **On** when the **Optimization Technique** option is set to **Area**, **Balanced**, or **Speed**. When the **Optimization Technique** option is set to **Speed**, Quartus II integrated synthesis attempts to restructure the multiplexers selectively and makes a good trade-off between area and f_{MAX}.

After you have compiled your design, you can view multiplexer restructuring information in the **Multiplexer Restructuring Statistics** report in the **Multiplexer Statistics** folder under **Analysis & Synthesis Optimization Results** in the **Analysis & Synthesis** section of the Compilation Report. Table 9–4 describes the information that is listed in the **Multiplexer Restructuring Statistics** report table for each bus of multiplexers.

Table 9–4. Multiplexer Information in the Multiplexer Restructuring Statistics Report

Heading	Description
Multiplexer Inputs	The number of different choices that are multiplexed together.
Bus Width	The width of the bus in bits.
Baseline Area	An estimate of how many logic cells are required to implement the bus of multiplexers (before any multiplexer restructuring takes place). This estimate can be used to identify any large multiplexers in the design.
Area if Restructured	An estimate of how many logic cells are required to implement the bus of multiplexers if **Multiplexer Restructuring** is applied.
Saving if Restructured	An estimate of how many logic cells are saved if **Multiplexer Restructuring** is applied.
Registered	An indication of whether registers are present on the multiplexer outputs. **Multiplexer Restructuring** uses the secondary control signals of a register (such as synchronous clear and synchronous load) to further reduce the amount of logic required to implement the bus of multiplexers.
Example Multiplexer Output	The name of one of the multiplexers' outputs. This name can help determine where in the design the multiplexer bus originated.

For more information about optimizing for multiplexers, refer to the *Multiplexers* section of the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Synthesis Effort

This option specifies the overall synthesis effort level in the Quartus II software. The level can be either **Fast** or **Auto**.

Auto is the default, which means synthesis goes through the normal flow and tries to optimize your design as much as possible.

When the effort level is set to **Fast**, Quartus II integrated synthesis skips a number of steps to make synthesis run much faster (at the cost of performance and resource utilization). This option is especially useful if you perform an early timing estimate. The early timing estimate feature gives you preliminary timing estimates before running a full compilation, which results in a quicker iteration time; therefore, you can save significant compilation time to get a good estimation of the final timing of your design.

Altera recommends using the **Fast** synthesis effort level with the Fitter early timing estimate feature. When the **Fast** synthesis effort level is used with the full Fitter, the Fitter runtime might increase because fast synthesis produces a netlist that is slightly harder for the Fitter to route as compared to the netlist from a normal synthesis.

To set the **Synthesis Effort** option from the Quartus II GUI, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.

2. Under **Category**, click **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**.
4. Next to **Name**, from the pull-down menu, select **Synthesis Effort**.
5. Next to **Setting**, from the pull-down menu, select **Auto** or **Fast**. Click **OK**.
6. Click **OK** to close the **Settings** dialog box.

To set the **Synthesis Effort** option at the command line, use the `--effort` option, as shown in Example 9–27.

Example 9–27. Command Syntax for Specifying Synthesis Effort Option

```
quartus_map <Design name> --effort= "auto | fast"
```

If you want to run fast synthesis with the Fitter **Early Timing Estimate** option, use the command shown in Example 9–28. This command runs the full flow with Timing Analysis.

Example 9–28. Command Syntax for running fast synthesis with Early Timing Estimate Option

```
quartus_sh --flow early_timing_estimate_with_synthesis <Design name>
```

You can also run this flow from the Tasks pane in the Quartus II software. Select and expand **Compile Design**, then **Analysis & Synthesis**. Double-click **Early Timing Estimate** to start the flow.

State Machine Processing

This logic option specifies the processing style used to compile a state machine. Table 9–5 lists the settings for this logic option, which you can apply to a state machine name or to a design entity containing a state machine. You can also set this option for your whole project on the **Analysis & Synthesis Settings** page in the **Settings** dialog box.

Table 9–5. State Machine Processing Settings

Setting	Description
Auto (Default)	Allows the compiler to choose what it determines to be the best encoding for the state machine
Minimal Bits	Uses the least number of bits to encode the state machine
One-Hot	Encodes the state machine in the one-hot style. See the example below for details.
User-Encoded	Encodes the state machine in the manner specified by the user
Sequential	Uses a binary encoding in which the first enumeration literal in the Enumeration Type has encoding `0`, the second `1`, and so on.
Gray	Uses an encoding in which the encodings for adjacent enumeration literals differ by exactly one bit. An N-bit gray code can represent 2^N values.
Johnson	Uses an encoding similar to a gray code, in which each state only has one bit different from its neighboring states. Each state is generated by shifting the previous state's bits to the right by 1; the most significant bit of each state is the negation of the least significant bit of the previous state. An N-bit Johnson code can represent at most 2N states but requires less logic than a gray encoding.

The default state machine encoding, which is **Auto**, uses one-hot encoding for FPGA devices and minimal-bits encoding for CPLDs. These settings achieve the best results on average, but another encoding style might be more appropriate for your design, so this option allows you to control the state machine encoding.

For guidelines to ensure that your state machine is inferred and encoded correctly, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

For one-hot encoding, the Quartus II software does not guarantee that each state has one bit set to one and all other bits to zero. Quartus II integrated synthesis creates one-hot register encoding by using standard one-hot encoding and then inverting the first bit. This results in an initial state with all zero values, and the remaining states have two 1 values. Quartus II integrated synthesis encodes the initial state with all zeros for the state machine power-up because all device registers power up to a low value. This encoding has the same properties as true one-hot encoding: each state can be recognized by the value of one bit. For example, in a one-hot-encoded state machine with five states including an initial or reset state, the software uses the following register encoding:

```
State 0     0 0 0 0 0
State 1     0 0 0 1 1
State 2     0 0 1 0 1
State 3     0 1 0 0 1
State 4     1 0 0 0 1
```

If the **State Machine Processing** logic option is set to **User-Encoded** in a Verilog HDL design, the software starts with the original design values for the state constants. For example, a Verilog HDL design can contain a declaration such as the following example:

```
parameter S0 = 4'b1010, S1 = 4'b0101, ...
```

If the software infers states `S0, S1,...` it uses the encoding `4'b1010, 4'b0101,...`. If necessary, the software inverts bits in a user-encoded state machine to ensure that all bits of the reset state of the state machine are zero.

To assign your own state encoding with the **User-Encoded** setting of the **State Machine Processing** option in a VHDL design, you must apply specific binary encoding to the elements of an enumerated type because enumeration literals have no numeric values in VHDL. Use the `syn_encoding` synthesis attribute to apply your encoding values. Refer to "Manually Specifying State Assignments Using the syn_encoding Attribute" for more information.

For information about the **Safe State Machine** option, refer to "Safe State Machines" on page 9–41.

Manually Specifying State Assignments Using the syn_encoding Attribute

The Quartus II software infers state machines from enumerated types and automatically assigns state encoding based on "State Machine Processing" on page 9–36. With this logic option, you can choose the value **User-Encoded** to use the encoding from your HDL code. However, in standard VHDL code, you cannot specify user encoding in the state machine description because enumeration literals have no numeric values in VHDL.

To assign your own state encoding for the **User-Encoded State Machine Processing** setting, use the `syn_encoding` synthesis attribute to apply specific binary encodings to the elements of an enumerated type or to specify an encoding style. The Quartus II software can implement Enumeration Types with the different encoding styles shown in Table 9–6.

Table 9–6. syn_encoding Attribute Values

Attribute Value	Description
"default"	Use an encoding based on the number of enumeration literals in the Enumeration Type. If there are fewer than five literals, use the "sequential" encoding. If there are more than five but fewer than 50 literals, use a "one-hot" encoding. Otherwise, use a "gray" encoding.
"sequential"	Use a binary encoding in which the first enumeration literal in the Enumeration Type has encoding 0, the second 1, and so on.
"gray"	Use an encoding in which the encodings for adjacent enumeration literals differ by exactly one bit. An N-bit gray code can represent 2^N values.
"johnson"	Use an encoding similar to a gray code. An N-bit Johnson code can represent at most 2N states but requires less logic than a gray encoding.
"one-hot"	The default encoding style requiring N bits, where N is the number of enumeration literals in the Enumeration Type.
"compact"	Use an encoding with the fewest bits.

The `syn_encoding` attribute must follow the enumeration type definition but precede its use.

In Example 9–29, the `syn_encoding` attribute associates a binary encoding with the states in the enumerated type `count_state`. In this example, the states are encoded with the following values: zero = "11", one = "01", two = "10", three = "00".

Example 9–29. Specifying User-Encoded States with the syn_encoding Attribute in VHDL

```
ARCHITECTURE rtl OF my_fsm IS
   TYPE count_state is (zero, one, two, three);
   ATTRIBUTE syn_encoding : STRING;
   ATTRIBUTE syn_encoding OF count_state : TYPE IS "11 01 10 00";
   SIGNAL present_state, next_state : count_state;
BEGIN
```

You can also use the `syn_encoding` attribute in Verilog HDL to direct the synthesis tool to use the encoding from your HDL code, instead of using the **State Machine Processing** option.

The `syn_encoding` value "user" instructs the Quartus II software to encode each state with its corresponding value from the Verilog HDL source code. By changing the values of your state constants, you can change the encoding of your state machine.

Example 9–30. Specifying User-Encoded States with the syn_encoding Attribute in Verilog-2001

```
(* syn_encoding = "user" *) reg [1:0] state;
parameter init = 0, last = 3, next = 1, later = 2;
always @ (state) begin
case (state)
init:
out = 2'b01;
next:
out = 2'b10;
later:
out = 2'b11;
last:
out = 2'b00;
endcase
end
```

In Example 9–30, the states are encoded as follows:

```
init = "00"
last = "11"
next = "01"
later = "10"
```

Without the `syn_encoding` attribute, the Quartus II software would encode the state machine based on the current value of the **State Machine Processing** logic option.

If you are also specifying a safe state machine (as described in "Safe State Machines" on page 9–41), separate the encoding style value in the quotation marks with the safe value with a comma, as follows: "`safe, one-hot`" or "`safe, gray`".

Manually Specifying Enumerated Types Using the enum_encoding Attribute

By default, the Quartus II software one-hot encodes all user-defined Enumerated Types. With the `enum_encoding` attribute, you can specify the logic encoding for an Enumerated Type and override the default `one-hot` encoding to improve the logic efficiency.

If an Enumerated Type represents the states of a state machine, using the `enum_encoding` attribute to specify a manual state encoding prevents the compiler from recognizing state machines based on the Enumerated Type. Instead, the compiler processes these state machines as "regular" logic using the encoding specified by the attribute, and they are not listed as state machines in the Report window for the project. If you want to control the encoding for a recognized state machine, use the **State Machine Processing** logic option and the `syn_encoding` synthesis attribute.

To use the `enum_encoding` attribute in a VHDL design file, associate the attribute with the Enumeration Type whose encoding you want to control. The `enum_encoding` attribute must follow the Enumeration Type Definition but precede its use. In addition, the attribute value must be a string literal that specifies either an arbitrary user encoding or an encoding style of `"default"`, `"sequential"`, `"gray"`, `"johnson"`, or `"one-hot"`.

An arbitrary user encoding consists of a space-delimited list of encodings. The list must contain as many encodings as there are enumeration literals in your Enumeration Type. In addition, the encodings must all have the same length, and each encoding must consist solely of values from the `std_ulogic` type declared by the `std_logic_1164` package in the **IEEE** library. In the code fragment of Example 9–31, the `enum_encoding` attribute specifies an arbitrary user encoding for the Enumeration Type `fruit`.

Example 9–31. Specifying an Arbitrary User Encoding for Enumerated Type

```
type fruit is (apple, orange, pear, mango);
attribute enum_encoding : string;
attribute enum_encoding of fruit : type is "11 01 10 00";
```

In this example, the enumeration literals are encoded as:

```
apple   = "11"
orange  = "01"
pear    = "10"
mango   = "00"
```

You might want to specify an encoding style, rather than a manual user encoding, especially when the Enumeration Type has a large number of enumeration literals. The Quartus II software can implement Enumeration Types with the different encoding styles shown in Table 9–7.

Table 9–7. enum_encoding Attribute Values

Attribute Value	Description
`"default"`	Use an encoding based on the number of enumeration literals in the Enumeration Type. If there are fewer than five literals, use the `"sequential"` encoding. If there are more than five but fewer than 50 literals, use a `"one-hot"` encoding. Otherwise, use a `"gray"` encoding.
`"sequential"`	Use a binary encoding in which the first enumeration literal in the Enumeration Type has encoding `0`, the second `1`, and so on.
`"gray"`	Use an encoding in which the encodings for adjacent enumeration literals differ by exactly one bit. An N-bit gray code can represent 2^N values.
`"johnson"`	Use an encoding similar to a gray code. An N-bit Johnson code can represent at most 2^N states but requires less logic than a gray encoding.
`"one-hot"`	The default encoding style requiring *N* bits, where *N* is the number of enumeration literals in the Enumeration Type.

Observe that in Example 9–31, the `enum_encoding` attribute manually specified a gray encoding for the Enumeration Type `fruit`. This example could be written more concisely by specifying the `"gray"` encoding style instead of a manual encoding, as shown in Example 9–32.

Example 9–32. Specifying the “gray” Encoding Style or Enumeration Type

```
type fruit is (apple, orange, pear, mango);
attribute enum_encoding : string;
attribute enum_encoding of fruit : type is "gray";
```

Safe State Machines

The **Safe State Machine** option and corresponding `syn_encoding` attribute value `safe` specify that the software should insert extra logic to detect an illegal state and force the state machine's transition to the reset state.

It is possible for a finite state machine to enter an illegal state—meaning the state registers contain a value that does not correspond to any defined state. By default, the behavior of the state machine that enters an illegal state is undefined. However, you can set the `syn_encoding` attribute to `safe` or use the **Safe State Machine** logic option if you want the state machine to recover deterministically from an illegal state. Use this option if you have asynchronous inputs to your state machine. The most common cause of this situation is a state machine that has control inputs that come from another clock domain, such as the control logic for a clock-crossing FIFO, because the state machine must have inputs from another clock domain. An alternative is to add synchronizer registers to the inputs.

The `safe` state machine value does not use any user-defined default logic from your HDL code that corresponds to unreachable states. Verilog HDL and VHDL allow you to explicitly specify a behavior for all states in the state machine, including unreachable states. However, synthesis tools detect if state machine logic is unreachable and minimize or remove the logic. Any flag signals or logic used in the design to indicate such an illegal state are also removed. If the state machine is implemented as safe, the recovery logic forces its transition from an illegal state to the reset state.

The **Safe State Machine** option can be set globally, or on individual state machines. To set this option, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. The **More Analysis & Synthesis Settings** dialog box appears.
4. In the **Existing option settings** list, select **Safe State Machine**.
5. Under **Option**, in the **Setting** list, select **On**.
6. Click **OK**.
7. Click **OK** to close the **Settings** dialog box.

You can also use the Assignment Editor to turn on the **Safe State Machine** option for specific state machines.

You can set the `syn_encoding` `safe` attribute on a state machine in HDL, as shown in Example 9–33 through Example 9–35.

Example 9–33. Verilog HDL Code: a Safe State Machine Attribute

```
reg [2:0] my_fsm /* synthesis syn_encoding = "safe" */;
```

Example 9–34. Verilog-2001 Code: a Safe State Machine Attribute

```
(* syn_encoding = "safe" *) reg [2:0] my_fsm;
```

Example 9–35. VHDL Code: a Safe State Machine Attribute

```
ATTRIBUTE syn_encoding OF my_fsm : TYPE IS "safe";
```

If you are also specifying an encoding style (as described in "Manually Specifying State Assignments Using the syn_encoding Attribute" on page 9–37), separate the encoding style value in the quotation marks with the `safe` value with a comma, as follows: `"safe, one-hot"` or `"safe, gray"`.

Safe state machine implementation can result in a noticeable area increase for the design. Therefore, Altera recommends that you set this option only on the critical state machines in the design where the safe mode is required, such as a state machine that uses inputs from asynchronous clock domains. You can also reduce the necessity of this option by correctly synchronizing inputs coming from other clock domains.

If the `safe` state machine assignment is made on an instance that is not recognized as a state machine, or an entity that contains a state machine, the software takes no action. You must restructure the code so that the instance is recognized and properly inferred as a state machine.

For guidelines to ensure that your state machine is inferred correctly, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Power-Up Level

This logic option causes a register (flipflop) to power up with the specified logic level, either **High** (1) or **Low** (0). Registers in the device core hardware power up to 0 in all Altera devices. For the register to power up with a logic level High specified using this option, the compiler performs an optimization referred to as NOT-gate push back on the register. NOT-gate push back adds an inverter to the input and the output of the register so that the reset and power-up conditions appear to be high and the device operates as expected. The register itself still powers up low, but the register output is inverted so the signal arriving at all destinations is high. This option is available for all Altera devices supported by the Quartus II software except MAX® 3000A and MAX 7000S devices.

The **Power-Up Level** option supports wildcard characters, and you can apply this option to any register, registered logic cell WYSIWYG primitive, or to a design entity containing registers if you want to set the power level for all registers in the design entity. If this option is assigned to a registered logic cell WYSIWYG primitive, such as an atom primitive from a third-party synthesis tool, you must turn on the **Perform WYSIWYG Primitive Resynthesis** logic option for it to take effect. You can also apply the option to a pin with the logic configurations described in the following list:

- If this option is turned on for an input pin, the option is transferred automatically to the register that is driven by the pin if the following conditions are present:
  - There is no logic, other than inversion, between the pin and the register
  - The input pin drives the data input of the register
  - The input pin does not fan-out to any other logic

- If this option is turned on for an output or bidirectional pin, it is transferred automatically to the register that feeds the pin, if the following conditions are present:
  - There is no logic, other than inversion, between the register and the pin
  - The register does not fan-out to any other logic

Inferred Power-Up Levels

Quartus II integrated synthesis reads default values for registered signals defined in Verilog HDL and VHDL code, and converts the default values into Power-Up Level settings. The software also synthesizes variables that are assigned values in Verilog HDL initial blocks into power-up conditions. Synthesis of these default and initial constructs enables the design's synthesized behavior to match, as closely as possible, the power-up state of the HDL code during a functional simulation.

For example, the following register declarations all set a power-up level of V_{CC} or a logic value "1":

```
signal q : std_logic = '1';  -- power-up to VCC

reg q = 1'b1;  // power-up to VCC

reg q;
initial begin q = 1'b1; end  // power-up to VCC
```

For more information about NOT-gate push back, the power-up states for Altera devices, and how the power-up level is affected by set and reset control signals, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Power-Up Don't Care

This logic option allows the compiler to optimize registers in the design that do not have a defined power-up condition. This option is turned on by default.

For example, your design might have a register with its D input tied to V_{CC}, and with no clear signal or other secondary signals. If this option is enabled, the compiler can choose for the register to power up to V_{CC}. Therefore, the output of the register is always V_{CC}. The compiler can remove the register and connect its output to V_{CC}. If you turn this option off or if you set a **Power-Up Level** assignment of **Low** for this register, the register transitions from GND to V_{CC} when the design starts up on the first clock signal. Thus, the register is not stuck at V_{CC} and cannot be removed. Similarly, if the register has a clear signal, it is not removed because after the clear is asserted, the register transitions again to GND and back to V_{CC}.

If the compiler performs a Power-Up Don't Care optimization that allows it to remove a register, it issues a message indicating it is doing so.

This project-wide option does not apply to registers that have the **Power-Up Level** logic option set to either **High** or **Low**.

Remove Duplicate Registers

If you turn on this logic option, the compiler removes registers that are identical to another register. If two registers generate the same logic, the compiler removes the second one, and the first one fans out to the second one's destinations. Also, if the deleted register has different logic option assignments, the compiler ignores them. This option is turned on by default.

Typically, you should use this option only if you want to prevent the compiler from removing duplicate registers. That is, you should use this option only with the **Off** setting. You can apply this option to an individual register or a design entity that contains registers.

Remove Redundant Logic Cells

This logic option removes redundant `LCELL` primitives or WYSIWYG cells. The option is off by default to preserve logic cells that have been used intentionally. If you turn on this option, the compiler optimizes a circuit for area and speed. You can set this option globally or apply it to individual nodes and entities. If you turn on the option at the global level, you can use the `keep` attribute or **Implement as Output of Logic Cell** logic option to preserve specific wire signals or nodes (refer to "Keep Combinational Node/Implement as Output of Logic Cell" on page 9–46).

Preserve Registers

This attribute and logic option directs the compiler not to minimize or remove a specified register during synthesis optimizations or register netlist optimizations. Optimizations can eliminate redundant registers and registers with constant drivers; this option prevents a register from being reduced to a constant or merged with a duplicate register. This option can preserve a register so you can observe it during simulation or with the SignalTap II Embedded Logic Analyzer. Additionally, it can preserve registers if you are creating a preliminary version of the design in which secondary signals are not specified. You can also use the attribute to preserve a duplicate of an I/O register so that one copy can be placed in an I/O cell and the second can be placed in the core. By default, the software might remove one of the two duplicate registers. In this case, the `preserve` attribute can be added to both registers to prevent this.

This option cannot preserve registers that have no fan-out. To prevent the removal of registers with no fan-out, refer to "Noprune Synthesis Attribute/Preserve Fan-out Free Register Node" on page 9–46.

The **Preserve Registers** option prevents a register from being inferred as a state machine.

You can set the **Preserve Registers** logic option in the Quartus II GUI or you can set the `preserve` attribute in your HDL code, as shown in Example 9–36 through Example 9–38. In these examples, the `my_reg` register is preserved.

In addition to `preserve`, the Quartus II software supports the `syn_preserve` attribute name for compatibility with other synthesis tools.

Example 9–36. Verilog HDL Code: syn_preserve Attribute

```
reg my_reg /* synthesis syn_preserve = 1 */;
```

Example 9–37. Verilog-2001 Code: syn_preserve Attribute

```
(* syn_preserve = 1 *) reg my_reg;
```

The `= 1` after the `preserve` in Example 9–36 and Example 9–37 is optional, because the assignment uses a default value of `1` when it is specified.

Example 9–38. VHDL Code: preserve Attribute

```
signal my_reg : stdlogic;
attribute preserve : boolean;
attribute preserve of my_reg : signal is true;
```

Disable Register Merging/Don't Merge Register

This logic option and attribute prevents the specified register from being merged with other registers and prevents other registers from being merged with the specified register. When applied to a design entity, it applies to all registers in the entity.

You can use this option to instruct the compiler to correctly use your timing constraints for the register during synthesis. For example, if the register has a multicycle constraint, this option prevents the compiler from merging other registers into the specified register, avoiding unintended timing effects and functional differences.

This option differs from the **Preserve Register** option because it does not prevent a register with constant drivers or a redundant register from being removed. In addition, this option prevents other registers from merging with the specified register.

You can set the **Disable Register Merging** logic option in the Quartus II GUI, or you can set the `dont_merge` attribute in your HDL code, as shown in Example 9–39 through Example 9–41. In these examples, the `my_reg` register is prevented from merges.

Example 9–39. Verilog HDL Code: dont_merge Attribute

```
reg my_reg /* synthesis dont_merge */;
```

Example 9–40. Verilog-2001 Code: dont_merge Attribute

```
(* dont_merge *) reg my_reg;
```

Example 9–41. VHDL Code: dont_merge Attribute

```
signal my_reg : stdlogic;
attribute dont_merge : boolean;
attribute dont_merge of my_reg : signal is true;
```

Noprune Synthesis Attribute/Preserve Fan-out Free Register Node

This synthesis attribute and corresponding logic option direct the compiler to preserve a fan-out-free register through the entire compilation flow. This is different from the **Preserve Registers** option, which prevents a register from being reduced to a constant or merged with a duplicate register. Standard synthesis optimizations remove nodes that do not directly or indirectly feed a top-level output pin. This option can retain a register so you can observe it in the Simulator or the SignalTap II Embedded Logic Analyzer. Additionally, it can retain registers if you are creating a preliminary version of the design in which the registers' fan-out logic is not specified. This option is supported for inferred registers in the Arria GX, Stratix series, Cyclone series, and MAX II device families.

You can set the **Preserve Fan-out Free Register Node** logic option in the Quartus II GUI, or you can set the `noprune` attribute in your HDL code, as shown in Example 9–42 though Example 9–44. In these examples, the `my_reg` register is preserved.

You must use the `noprune` attribute instead of the logic option if the register has no immediate fan-out in its module or entity. If you do not use the synthesis attribute, registers with no fan-out are removed (or "pruned") during Analysis and Elaboration before the logic synthesis stage applies any logic options. If the register has no fan-out in the full design, but has fan-out within its module or entity, you can use the logic option to retain the register through compilation.

The attribute name `syn_noprune` is supported for compatibility with other synthesis tools.

Example 9–42. Verilog HDL Code: syn_noprune Attribute

```
reg my_reg /* synthesis syn_noprune */;
```

Example 9–43. Verilog-2001 Code: noprune Attribute

```
(* noprune *) reg my_reg;
```

Example 9–44. VHDL Code: noprune Attribute

```
signal my_reg : stdlogic;
attribute noprune: boolean;
attribute noprune of my_reg : signal is true;
```

Keep Combinational Node/Implement as Output of Logic Cell

This synthesis attribute and corresponding logic option direct the compiler to keep a wire or combinational node through logic synthesis minimizations and netlist optimizations. A wire that has a `keep` attribute or a node that has the **Implement as Output of Logic Cell** logic option applied becomes the output of a logic cell in the final synthesis netlist, and the name of the logic cell will be the same as the name of the wire or node. You can use this directive to make combinational nodes visible to the SignalTap II Embedded Logic Analyzer.

The option cannot keep nodes that have no fan-out. Node names cannot be maintained for wires with tri-state drivers, or if the signal feeds a top-level pin of the same name (in this case, the node name is changed to a name such as <*net name*>`~buf0`).

You can set the **Implement as Output of Logic Cell** logic option in the Quartus II GUI, or you can set the `keep` attribute in your HDL code, as shown in Example 9–45 through Example 9–47. In these examples, the compiler maintains the node name `my_wire`.

In addition to `keep`, the Quartus II software supports the `syn_keep` attribute name for compatibility with other synthesis tools.

Example 9–45. Verilog HDL Code: keep Attribute

```
wire my_wire /* synthesis keep = 1 */;
```

Example 9–46. Verilog-2001 Code: keep Attribute

```
(* keep = 1 *) wire my_wire;
```

Example 9–47. VHDL Code: syn_keep Attribute

```
signal my_wire: bit;
attribute syn_keep: boolean;
attribute syn_keep of my_wire: signal is true;
```

Don't Retime, Disabling Synthesis Netlist Optimizations

This attribute disables synthesis retiming optimizations on the specified register. When applied to a design entity, it applies to all registers in the entity.

You can use this option to turn off retiming optimizations and prevent node name changes so that the compiler can correctly use your timing constraints for the register.

You can set the **Netlist Optimizations** logic option to **Never Allow** in the Quartus II GUI to disable retiming along with other synthesis netlist optimizations, or you can set the `dont_retime` attribute in your HDL code, as shown in Example 9–48 through Example 9–50. In these examples, the `my_reg` register is prevented from being retimed.

Example 9–48. Verilog HDL Code: dont_retime Attribute

```
reg my_reg /* synthesis dont_retime */;
```

Example 9–49. Verilog-2001 Code: dont_retime Attribute

```
(* dont_retime *) reg my_reg;
```

Example 9–50. VHDL Code: dont_retime Attribute

```
signal my_reg : std_logic;
attribute dont_retime : boolean;
attribute dont_retime of my_reg : signal is true;
```

For compatibility with third-party synthesis tools, Quartus II integrated synthesis also supports the attribute `syn_allow_retiming`. To disable retiming, set `syn_allow_retiming` to `0` (Verilog HDL) or `false` (VHDL). This attribute does not have any effect when set to `1` or `true`.

Don't Replicate, Disabling Synthesis Netlist Optimizations

This attribute disables synthesis replication optimizations on the specified register. When applied to a design entity, it applies to all registers in the entity.

You can use this option to turn off register replication (or duplication) optimizations so that the compiler can use your timing constraints for the register.

You can set the **Netlist Optimizations** logic option to **Never Allow** in the Quartus II GUI to disable replication along with other synthesis netlist optimizations, or you can set the `dont_replicate` attribute in your HDL code, as shown in Example 9–51 through Example 9–53. In these examples, the `my_reg` register is prevented from being replicated.

Example 9–51. Verilog HDL Code: dont_replicate Attribute

```
reg my_reg /* synthesis dont_replicate  */;
```

Example 9–52. Verilog-2001 Code: dont_replicate Attribute

```
(* dont_replicate *) reg my_reg;
```

Example 9–53. VHDL Code: dont_replicate Attribute

```
signal my_reg : std_logic;
attribute dont_replicate : boolean;
attribute dont_replicate of my_reg : signal is true;
```

For compatibility with third-party synthesis tools, Quartus II integrated synthesis also supports the attribute `syn_replicate`. To disable replication, set `syn_replicate` to `0` (Verilog HDL) or `false` (VHDL). This attribute does not have any effect when set to `1` or `true`.

© March 2009 Altera Corporation

Maximum Fan-Out

This attribute and logic option directs the compiler to control the number of destinations fed by a node. The compiler duplicates a node and splits its fan-out until the individual fan-out of each copy falls below the maximum fan-out restriction. You can apply this option to a register or a logic cell buffer, or to a design entity that contains these elements. You can use this option to reduce the load of critical signals, which can improve performance. You can use the option to instruct the compiler to duplicate (or replicate) a register that feeds nodes in different locations on the target device. Duplicating the register can allow the Fitter to place these new registers closer to their destination logic, minimizing routing delay.

This option is available for all devices supported in the Quartus II software except MAX 3000, MAX 7000, FLEX 10K®, and ACEX® 1K devices. To turn off the option for a given node if the option is set at a higher level of the design hierarchy, in the **Netlist Optimizations** logic option, select **Never Allow**. If not disabled by the **Netlist Optimizations** option, the maximum fan-out constraint is honored as long as the following conditions are met:

- The node is not part of a cascade, carry, or register cascade chain
- The node does not feed itself
- The node feeds other logic cells, DSP blocks, RAM blocks, and/or pins through data, address, clock enable, and so on, but not through any asynchronous control ports (such as asynchronous clear)

The software does not create duplicate nodes in these cases, either because there is no clear way to duplicate the node, or to avoid the possible situation in which small differences in timing could produce functional differences in the implementation (in the third condition above where asynchronous control signals are involved). If the constraint cannot be applied because one of these conditions is not met, the Quartus II software issues a message indicating that it ignored maximum fan-out assignment. To instruct the software not to check the node’s destinations for possible problems like the third condition, you can set the **Netlist Optimizations** logic option to **Always Allow** for a given node.

If you have enabled any of the Quartus II netlist optimizations that affect registers, add the `preserve` attribute to any registers to which you have set a `maxfan` attribute. The `preserve` attribute ensures that the registers are not affected by any of the netlist optimization algorithms, such as register retiming.

For details about netlist optimizations, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

You can set the **Maximum Fan-Out** logic option in the Quartus II GUI; this option supports wildcard characters. You can also set the `maxfan` attribute in your HDL code, as shown in Example 9–54 through Example 9–56. In these examples, the compiler duplicates the `clk_gen` register, so its fan-out is not greater than 50.

In addition to `maxfan`, the Quartus II software supports the `syn_maxfan` attribute name for compatibility with other synthesis tools.

Example 9–54. Verilog HDL Code: syn_maxfan Attribute

```
reg clk_gen /* synthesis syn_maxfan = 50 */;
```

Example 9–55. Verilog-2001 Code: maxfan Attribute

```
(* maxfan = 50 *) reg clk_gen;
```

Example 9–56. VHDL Code: maxfan Attribute

```
signal clk_gen : stdlogic;
attribute maxfan : signal ;
attribute maxfan of clk_gen : signal is 50;
```

Controlling Clock Enable Signals with Auto Clock Enable Replacement and direct_enable

The **Auto Clock Enable Replacement** logic option allows the software to find logic that feeds a register and move the logic to the register's clock enable input port. The option is on by default. You can set this option to **Off** for individual registers or design entities to solve fitting or performance issues with designs that have many clock enables. Turning the option off prevents the software from using the register's clock enable port. The software implements the clock enable functionality using multiplexers in logic cells.

If specific logic is not automatically moved to a clock enable input with the **Auto Clock Enable Replacement** logic option, you can instruct the software to use a direct clock enable signal. Applying the `direct_enable` attribute to a specific signal instructs the software to use the clock enable port of a register to implement the signal. The attribute ensures that the clock enable port is driven directly by the signal, and the signal is not optimized or combined with any other logic.

Example 9–57 through Example 9–59 show how to set this attribute to ensure that the signal is preserved and used directly as a clock enable.

In addition to `direct_enable`, the Quartus II software supports the `syn_direct_enable` attribute name for compatibility with other synthesis tools.

Example 9–57. Verilog HDL Code: direct_enable attribute

```
wire my_enable /* synthesis direct_enable = 1 */ ;
```

Example 9–58. Verilog-2001 Code: syn_direct_enable attribute

```
(* syn_direct_enable *) wire my_enable;
```

Example 9–59. VHDL Code: direct_enable attribute

```
attribute direct_enable: boolean;
attribute direct_enable of my_enable: signal is true;
```

Megafunction Inference Control

The Quartus II compiler automatically recognizes certain types of HDL code and infers the appropriate megafunction. The software uses the Altera megafunction code when compiling your design, even when you do not specifically instantiate the megafunction. The software infers megafunctions to take advantage of logic that is optimized for Altera devices. The area and performance of such logic can be better than the results obtained by inferring generic logic from the same HDL code.

Additionally, you must use megafunctions to access certain architecture-specific features, such as RAM, DSP blocks, and shift registers that generally provide improved performance compared with basic logic cells.

For details about coding style recommendations when targeting megafunctions in Altera devices, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

The Quartus II software provides options to control the inference of certain types of megafunctions, as described in the following subsections.

Multiply-Accumulators and Multiply-Adders

Use the **Auto DSP Block Replacement** logic option to control DSP block inference for multiply-accumulations and multiply-adders. This option is turned on by default. To disable inference, turn off this option for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box, or disable the option for a specific block with the Assignment Editor.

Any registers that the software maps to the ALTMULT_ACCUM and ALTMULT_ADD megafunctions and places in DSP blocks are not available in the Simulator because their node names do not exist after synthesis.

Shift Registers

Use the **Auto Shift Register Replacement** logic option to control shift register inference. This option is turned on by default. To disable inference, turn off this option for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box by clicking **More Settings** and setting the option to **Off**. You can also disable the option for a specific block with the Assignment Editor. The software might not infer small shift registers because small shift registers typically do not benefit from implementation in dedicated memory. However, you can use the **Allow Any Shift Register Size for Recognition** logic option to instruct synthesis to infer a shift register even when its size is considered too small.

The registers that the software maps to the ALTSHIFT_TAPS megafunction and places in RAM are not available in the Simulator because their node names do not exist after synthesis.

The **Auto Shift Register Replacement** logic option is turned off automatically when a formal verification tool is selected on the **EDA Tool Settings** page. The software issues a warning and lists shift registers that would have been inferred if no formal verification tool was selected in the compilation report. To allow the use of a megafunction for the shift register in the formal verification flow, you can either instantiate a shift register explicitly using the MegaWizard™ Plug-In Manager or make the shift register into a black box in a separate entity/module.

RAM and ROM

Use the **Auto RAM Replacement** and **Auto ROM Replacement** logic options to control RAM and ROM inference, respectively. These options are turned on by default. To disable inference, turn off the appropriate option for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box by clicking **More Settings** and setting the option to **Off**. You can also disable the option for a specific block with the Assignment Editor.

Although inferred shift registers are implemented in RAM blocks, you cannot turn off the **Auto RAM Replacement** option to disable shift register replacement. Use the **Auto Shift Register Replacement** option (refer to “Shift Registers”).

The software might not infer very small RAM or ROM blocks because very small memory blocks can typically be implemented more efficiently by using the registers in the logic. However, you can use the **Allow Any RAM Size for Recognition** and **Allow Any ROM Size for Recognition** logic options to instruct synthesis to infer a memory block even when its size is considered too small.

The **Auto ROM Replacement** logic option is automatically turned off when a formal verification tool is selected in the **EDA Tool Settings** page. A warning is issued and a report panel lists ROMs that would have been inferred if no formal verification tool was selected. To allow the use of a megafunction for the shift register in the formal verification flow, you can either instantiate a ROM explicitly using the MegaWizard Plug-In Manager or create a black box for the ROM in a separate entity/module.

Although formal verification tools do not support inferred RAM blocks, because of the importance of inferring RAM in many designs, the **Auto RAM Replacement** logic option remains on when a formal verification tool is selected in the **EDA Tool Settings** page. The Quartus II software automatically performs black box instance for any module or entity that contains a RAM block that is inferred. The software issues a warning and lists the black box that is created in the compilation report. This block box allows formal verification tools to proceed; however, the entire module or entity containing the RAM cannot be verified in the tool. Altera recommends that you explicitly instantiate RAM blocks in separate modules or entities so that as much logic as possible can be verified by the formal verification tool.

 © March 2009 Altera Corporation

RAM to Logic Cell Conversion

The **Auto RAM to Logic Cell Conversion** option allows Quartus II integrated synthesis to convert RAM blocks that are small in size to logic cells if the logic cell implementation is deemed to give better quality of results. Only single-port or simple-dual port RAMs with no initialization files can be converted to logic cells. This option is off by default. You can set this option globally or apply it to individual RAM nodes. You can enable this option by turning on the appropriate option for your whole project in the **More Analysis & Synthesis Settings** dialog box.

For the FLEX 10K, APEX series, Arria GX, and the Stratix series of devices, the software uses the following rules to determine whether a RAM should be placed in logic cells or a dedicated RAM block:

- If the number of words is less than 16, use a RAM block if the total number of bits is greater than or equal to 64
- If the number of words is greater than or equal to 16, use a RAM block if the total number of bits is greater than or equal to 32
- Otherwise, implement the RAM in logic cells

For the Cyclone series of devices, the software uses the following rules:

- If the number of words is greater than or equal to 64, use a RAM block
- If the number of words is greater than or equal to 16 and less than 64, use a RAM block if the total number of bits is greater than or equal to 128
- Otherwise, implement the RAM in logic cells

RAM Style and ROM Style—for Inferred Memory

These attributes specify the implementation for an inferred RAM or ROM block. You can specify the type of TriMatrix embedded memory block to be used, or specify the use of standard logic cells (LEs or ALMs). The attributes are supported only for device families with TriMatrix embedded memory blocks.

The `ramstyle` and `romstyle` attributes take a single string value. The values "`M512`", "`M4K`", "`M-RAM`", "`MLAB`", "`M9K`", and "`M144K`" (as applicable for the target device family) indicate the type of memory block to use for the inferred RAM or ROM. If you set the attribute to a block type that does not exist in the target device family, the software generates a warning and ignores the assignment. The value `logic` indicates that the RAM or ROM should be implemented in regular logic rather than dedicated memory blocks. You can set the attribute on a module or entity, in which case it specifies the default implementation style for all inferred memory blocks in the immediate hierarchy. You can also set the attribute on a specific signal (VHDL) or variable (Verilog HDL) declaration, in which case it specifies the preferred implementation style for that specific memory, overriding the default implementation style.

If you specify a value of `logic`, the memory still appears as a RAM or ROM block in the RTL Viewer, but it is converted to regular logic during a later synthesis step.

In addition to `ramstyle` and `romstyle`, the Quartus II software supports the `syn_ramstyle` attribute name for compatibility with other synthesis tools.

Example 9–60 through Example 9–62 specify that all memory in the module or entity my_memory_blocks should be implemented using a specific type of block.

Example 9–60. Verilog-1995 Code: Applying a romstyle Attribute to a Module Declaration

```
module my_memory_blocks (...) /* synthesis romstyle = "M4K" */;
```

Example 9–61. Verilog-2001 Code: Applying a ramstyle Attribute to a Module Declaration

```
(* ramstyle = "M512" *) module my_memory_blocks (...);
```

Example 9–62. VHDL Code: Applying a romstyle Attribute to an Architecture

```
architecture rtl of my_ my_memory_blocks is
attribute romstyle : string;
attribute romstyle of rtl : architecture is "M-RAM";
begin
```

Example 9–63 through Example 9–65 specify that the inferred memory my_ram or my_rom should be implemented using regular logic instead of a TriMatrix memory block.

Example 9–63. Verilog-1995 Code: Applying a syn_ramstyle Attribute to a Variable Declaration

```
reg [0:7] my_ram[0:63] /* synthesis syn_ramstyle = "logic" */;
```

Example 9–64. Verilog-2001 Code: Applying a romstyle Attribute to a Variable Declaration

```
(* romstyle = "logic" *) reg [0:7] my_rom[0:63];
```

Example 9–65. VHDL Code: Applying a ramstyle Attribute to a Signal Declaration

```
type memory_t is array (0 to 63) of std_logic_vector (0 to 7);
signal my_ram : memory_t;
attribute ramstyle : string;
attribute ramstyle of my_ram : signal is "logic";
```

Turning Off Add Pass-Through Logic to Inferred RAMs/ no_rw_check Attribute Setting

Setting the no_rw_check value for the ramstyle attribute, or turning off the corresponding global **Add Pass-Through Logic to Inferred RAMs** logic option indicates that your design does not depend on the behavior of the inferred RAM when there are reads and writes to the same address in the same clock cycle. If you specify the attribute or turn off the logic option, the Quartus II software can choose a read-during-write behavior instead of using the read-during-write behavior of your HDL source code.

In some cases, an inferred RAM must be mapped into regular logic cells because it has a read-during-write behavior that is not supported by the TriMatrix memory blocks in your target device. In other cases, the Quartus II software must insert extra logic to mimic read-during-write behavior of the HDL source, increasing the area of your design and potentially reducing its performance. In these cases, you can use the

attribute to specify that the software can implement the RAM directly in a TriMatrix memory block without using logic. You can also use the attribute to prevent a warning message for dual-clock RAMs in the case that the inferred behavior in the device does not exactly match the read-during-write conditions described in the HDL code.

For more information about recommended styles for inferring RAM and some of the issues involved with different read-during-write conditions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

To set the **Add Pass-Through Logic to Inferred RAMs** logic option through the Quartus II GUI, click **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box. Example 9–66 and Example 9–67 use two addresses and normally require extra logic after the RAM to ensure that the read-during-write conditions in the device match the HDL code. If you don't require a defined read-during-write condition in your design, this extra logic is not required. With the `no_rw_check` attribute, Quartus II integrated synthesis won't generate the extra logic.

Example 9–66. Verilog HDL Inferred RAM Using no_rw_check Attribute

```
module ram_infer (q, wa, ra, d, we, clk);
    output [7:0] q;
    input [7:0] d;
    input [6:0] wa;
    input [6:0] ra;
    input we, clk;
    reg [6:0] read_add;
    (* ramstyle = "no_rw_check" *) reg [7:0] mem [127:0];
    always @ (posedge clk) begin
        if (we)
            mem[wa] <= d;
        read_add <= ra;
    end
    assign q = mem[read_add];
endmodule
```

Example 9–67. VHDL Inferred RAM Using no_rw_check Attribute

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ram IS
    PORT (
        clock: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        write_address: IN INTEGER RANGE 0 to 31;
        read_address: IN INTEGER RANGE 0 to 31;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
END ram;

ARCHITECTURE rtl OF ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL ram_block: MEM;
    ATTRIBUTE ramstyle : string;
    ATTRIBUTE ramstyle of ram_block : signal is "no_rw_check";
    SIGNAL read_address_reg: INTEGER RANGE 0 to 31;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
            read_address_reg <= read_address;
        END IF;
    END PROCESS;
    q <= ram_block(read_address_reg);
END rtl;
```

You can use a ramstyle attribute with the MLAB value so that the Quartus II software can infer a small RAM block and place it in an MLAB.

This attribute is also useful in cases where some asynchronous RAM blocks might be coded with read-during-write behavior that does not match the Stratix III architecture. Thus, the device behavior would not exactly match the behavior described in the code. If the difference in behavior is acceptable in your design, use the ramstyle attribute with the `no_rw_check` value to specify that the software should not check the read-during-write behavior when inferring the RAM. When this attribute is set, Quartus II integrated synthesis allows the behavior of the output to be different when the asynchronous read occurs on an address that had a write on the most recent clock edge. That is, functional HDL simulation results will not match the hardware behavior if you write to an address that is being read.

To include both attributes, set the value of the ramstyle attribute to "`MLAB, no_rw_check`".

Example 9–68 and Example 9–69 show the method of setting two values to the ramstyle attribute using a small asyncronous RAM block, with the ramstyle synthesis attribute set so that the memory can be implemented in the MLAB memory block and the read-during-write behavior is not important. Without the attribute, this design requires 512 registers and 240 ALUTs. With the attribute, the design requires 8 memory ALUTs and just 15 registers.

© March 2009 Altera Corporation

Example 9–68. Verilog HDL Inferred RAM Using no_rw_check and MLAB Attributes

```
module async_ram (
    input   [5:0] addr,
    input   [7:0] data_in,
    input         clk,
    input         write,
    output  [7:0] data_out );

   (* ramstyle = "MLAB, no_rw_check" *) reg [7:0] mem[0:63];

   assign  data_out = mem[addr];

   always @ (posedge clk)
   begin
      if (write)
         mem[addr] = data_in;
   end
endmodule
```

Example 9–69. VHDL Inferred RAM Using no_rw_check and MLAB Attributes

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY ram IS
   PORT (
      clock: IN STD_LOGIC;
      data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
      write_address: IN INTEGER RANGE 0 to 31;
      read_address: IN INTEGER RANGE 0 to 31;
      we: IN STD_LOGIC;
      q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
END ram;

ARCHITECTURE rtl OF ram IS
   TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL ram_block: MEM;
   ATTRIBUTE ramstyle : string;
   ATTRIBUTE ramstyle of ram_block : signal is "MLAB , no_rw_check";
   SIGNAL read_address_reg: INTEGER RANGE 0 to 31;
BEGIN
   PROCESS (clock)
   BEGIN
      IF (clock'event AND clock = '1') THEN
         IF (we = '1') THEN
            ram_block(write_address) <= data;
         END IF;
         read_address_reg <= read_address;
      END IF;
   END PROCESS;
   q <= ram_block(read_address_reg);
END rtl;
```

RAM Initialization File—for Inferred Memory

The `ram_init_file` attribute specifies the initial contents of an inferred memory in the form of a Memory Initialization File (**.mif**). The attribute takes a string value containing the name of the RAM initialization file.

Example 9–70. Verilog-1995 Code: Applying a ram_init_file Attribute

```
reg [7:0] mem[0:255] /* synthesis ram_init_file
= " my_init_file.mif" */;
```

Example 9–71. Verilog-2001 Code: Applying a ram_init_file Attribute

```
(* ram_init_file = "my_init_file.mif" *) reg [7:0] mem[0:255];
```

Example 9–72. VHDL Code: Applying a ram_init_file Attribute

```
type mem_t is array(0 to 255) of unsigned(7 downto 0);
signal ram : mem_t;
attribute ram_init_file : string;
attribute ram_init_file of ram :
signal is "my_init_file.mif";
```

In VHDL, you can also initialize the contents of an inferred memory by specifying a default value for the corresponding signal. In Verilog HDL, you can use an initial block to specify the memory contents. Quartus II integrated synthesis automatically converts the default value into a **.mif** file for the inferred RAM.

Multiplier Style—for Inferred Multipliers

The `multstyle` attribute specifies the implementation style for multiplication operations (`*`) in your HDL source code. You can use this attribute to specify whether you prefer the compiler to implement a multiplication operation in general logic or dedicated hardware, if available in the target device.

The `multstyle` attribute takes a string value of `"logic"` or `"dsp"`, indicating a preferred implementation in logic or in dedicated hardware, respectively. In Verilog HDL, apply the attribute to a module declaration, a variable declaration, or a specific binary expression containing the `*` operator. In VHDL, apply the synthesis attribute to a signal, variable, entity, or architecture.

Specifying a `multstyle` of `"dsp"` does not guarantee that the Quartus II software can implement a multiplication in dedicated DSP hardware. The final implementation depends on several conditions, including the availability of dedicated hardware in the target device, the size of the operands, and whether or not one or both operands are constant.

In addition to `multstyle`, the Quartus II software supports the `syn_multstyle` attribute name for compatibility with other synthesis tools.

When applied to a Verilog HDL module declaration, the attribute specifies the default implementation style for all instances of the `*` operator in the module. For example, in the following code examples, the multstyle attribute directs the Quartus II software to implement all multiplications inside module `my_module` in dedicated multiplication hardware.

Example 9–73. Verilog-1995 Code: Applying a multstyle Attribute to a Module Declaration

```
module my_module (...) /* synthesis multstyle = "dsp" */;
```

Example 9–74. Verilog-2001 Code: Applying a multstyle Attribute to a Module Declaration

```
(* multstyle = "dsp" *) module my_module(...);
```

When applied to a Verilog HDL variable declaration, the attribute specifies the implementation style to be used for a multiplication operator whose result is directly assigned to the variable. It overrides the `multstyle` attribute associated with the enclosing module, if present. In Example 9–75 and Example 9–76, the `multstyle` attribute applied to variable `result` directs the Quartus II software to implement `a * b` in general logic rather than dedicated hardware.

Example 9–75. Verilog-2001 Code: Applying a multstyle Attribute to a Variable Declaration

```
wire [8:0] a, b;
(* multstyle = "logic" *) wire [17:0] result;
assign result = a * b;  //Multiplication must be
                        //directly assigned to result
```

Example 9–76. Verilog-1995 Code: Applying a multstyle Attribute to a Variable Declaration

```
wire [8:0] a, b;
wire [17:0] result /* synthesis multstyle = "logic" */;
assign result = a * b;  //Multiplication must be
                        //directly assigned to result
```

When applied directly to a binary expression containing the `*` operator, the attribute specifies the implementation style for that specific operator alone and overrides any `multstyle` attribute associated with the target variable or enclosing module. In Example 9–77, the `multstyle` attribute indicates that `a * b` should be implemented in dedicated hardware.

Example 9–77. Verilog-2001 Code: Applying a multstyle Attribute to a Binary Expression

```
wire [8:0] a, b;
wire [17:0] result;
assign result = a * (* multstyle = "dsp" *) b;
```

You cannot use Verilog-1995 attribute syntax to apply the `multstyle` attribute to a binary expression.

When applied to a VHDL entity or architecture, the attribute specifies the default implementation style for all instances of the `*` operator in the entity or architecture. In Example 9–78, the `multstyle` attribute directs the Quartus II software to use dedicated hardware, if possible, for all multiplications inside architecture `rtl` of entity `my_entity`.

Example 9–78. VHDL Code: Applying a multstyle Attribute to an Architecture

```
architecture rtl of my_entity is
   attribute multstyle : string;
   attribute multstyle of rtl : architecture is "dsp";
begin
```

When applied to a VHDL signal or variable, the attribute specifies the implementation style to be used for all instances of the * operator whose result is directly assigned to the signal or variable. It overrides the `multstyle` attribute associated with the enclosing entity or architecture, if present. In Example 9–79, the `multstyle` attribute associated with signal `result` directs the Quartus II software to implement `a * b` in general logic rather than dedicated hardware.

Example 9–79. VHDL Code: Applying a multstyle Attribute to a Signal or Variable

```
signal a, b : unsigned(8 downto 0);
signal result : unsigned(17 downto 0);

attribute multstyle : string;
attribute multstyle of result : signal is "logic";
result <= a * b;
```

Full Case

A Verilog HDL case statement is considered full when its case items cover all possible binary values of the case expression or when a default case statement is present. A `full_case` attribute attached to a case statement header that is not full forces the unspecified states to be treated as a "don't care" value. VHDL case statements must be full, so the attribute does not apply to VHDL.

Using this attribute on a case statement that is not full avoids the latch inference problems discussed in the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

Latches have limited support in formal verification tools. It is important to ensure that you do not infer latches unintentionally; for example, through an incomplete case statement when using formal verification. Formal verification tools do support the `full_case` synthesis attribute (with limited support for attribute syntax, as described in "Synthesis Attributes" on page 9–25).

When you use the `full_case` attribute, there is a potential cause for a simulation mismatch between Verilog HDL functional and post-Quartus II simulation because unknown case statement cases can still function like latches during functional simulation. For example, a simulation mismatch can occur with the code in Example 9–80 when `sel` is `2'b11` because a functional HDL simulation output behaves like a latch while the Quartus II simulation output behaves like "don't care."

Altera recommends making the case statement "full" in your regular HDL code, instead of using the `full_case` attribute.

The case statement in Example 9–80 is not full because not all binary values for `sel` are specified. Because the `full_case` attribute is used, synthesis treats the output as "don't care" when the `sel` input is `2'b11`.

Example 9–80. Verilog HDL Code: a full_case Attribute

```
module full_case (a, sel, y);
   input [3:0] a;
   input [1:0] sel;
   output y;
   reg y;
   always @ (a or sel)
   case (sel) // synthesis full_case
      2'b00: y=a[0];
      2'b01: y=a[1];
      2'b10: y=a[2];
   endcase
endmodule
```

Verilog-2001 syntax also accepts the statements in Example 9–81 in the `case` header instead of the comment form shown in Example 9–80.

Example 9–81. Verilog-2001 Syntax for the full_case Attribute

```
(* full_case *) case (sel)
```

Parallel Case

The `parallel_case` attribute indicates that a Verilog HDL case statement should be considered parallel; that is, only one case item can be matched at a time. Case items in Verilog HDL case statements might overlap. To resolve multiple matching case items, the Verilog HDL language defines a priority relationship among case items in which the case statement always executes the first case item that matches the case expression value. By default, the Quartus II software implements the extra logic required to satisfy this priority relationship.

Attaching a `parallel_case` attribute to a case statement's header allows the Quartus II software to consider its case items as inherently parallel; that is, at most one case item matches the case expression value. Parallel case items reduce the complexity of the generated logic.

In VHDL, the individual choices in a case statement might not overlap, so they are always parallel and this attribute does not apply.

Use this attribute only when the `case` statement is truly parallel. If you use the attribute in any other situation, the generated logic will not match the functional simulation behavior of the Verilog HDL.

Altera recommends that you avoid using the `parallel_case` attribute, due to the possibility of introducing mismatches between Verilog HDL functional and post-Quartus II simulation.

If you specify SystemVerilog-2005 as the supported Verilog HDL version for your design, you can use the SystemVerilog keyword `unique` to achieve the same result as the `parallel_case` directive without causing simulation mismatches.

Example 9–82 shows a casez statement with overlapping case items. In functional HDL simulation, the three case items have a priority order that depends on the bits in sel. For example, sel[2] takes priority over sel[1], which takes priority over sel[0]. However, the synthesized design can simulate differently because the parallel_case attribute eliminates this priority order. If more than one bit of sel is high, more than one output (a, b, or c) is high as well, a situation that cannot occur in functional HDL simulation.

Example 9–82. Verilog HDL Code: a parallel_case Attribute

```
module parallel_case (sel, a, b, c);
    input [2:0] sel;
    output a, b, c;
    reg a, b, c;
    always @ (sel)
    begin
        {a, b, c} = 3'b0;
        casez (sel) // synthesis parallel_case
            3'b1??: a = 1'b1;
            3'b?1?: b = 1'b1;
            3'b??1: c = 1'b1;
        endcase
    end
endmodule
```

Verilog-2001 syntax also accepts the statements shown in Example 9–83 in the case (or casez) header instead of the comment form, as shown in Example 9–82.

Example 9–83. Verilog-2001 Syntax

```
(* parallel_case *) casez (sel)
```

Translate Off and On / Synthesis Off and On

The translate_off and translate_on synthesis directives indicate whether the Quartus II software or a third-party synthesis tool should compile a portion of HDL code that is not relevant for synthesis. The translate_off directive marks the beginning of code that the synthesis tool should ignore; the translate_on directive indicates that synthesis should resume. You can also use the synthesis_on and synthesis_off directives as a synonym for translate on and off.

A common use of these directives is to indicate a portion of code that is intended for simulation only. The synthesis tool reads synthesis-specific directives and processes them during synthesis; however, third-party simulation tools read the directives as comments and ignore them. Example 9–84 and Example 9–85 show these directives.

Example 9–84. Verilog HDL Code: Translate Off and On

```
// synthesis translate_off
parameter tpd = 2;    // Delay for simulation
#tpd;
// synthesis translate_on
```

Example 9–85. VHDL Code: Translate Off and On

```
-- synthesis translate_off
use std.textio.all;
-- synthesis translate_on
```

If you wish to ignore a portion of code in Quartus II integrated synthesis only, you can use the Altera-specific attribute keyword `altera`. For example, use the `// altera translate_off` and `// altera translate_on` directives to direct Quartus II integrated synthesis to ignore a portion of code that is intended only for other synthesis tools.

Ignore translate_off and synthesis_off Directives

The **Ignore translate_off and synthesis_off directives** logic option directs Quartus II integrated synthesis to ignore the `translate_off` and `synthesis_off` directives described in the previous section. This allows you to compile code that was previously intended to be ignored by third-party synthesis tools; for example, megafunction declarations that were treated as black boxes in other tools but can be compiled in the Quartus II software. To set the **Ignore translate_off and synthesis_off directives** logic option, click **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box.

Read Comments as HDL

The `read_comments_as_HDL` synthesis directive indicates that the Quartus II software should compile a portion of HDL code that is commented out. This directive allows you to comment out portions of HDL source code that are not relevant for simulation, while instructing the Quartus II software to read and synthesize that same source code. Setting the `read_comments_as_HDL` directive to `on` marks the beginning of commented code that the synthesis tool should read; setting the `read_comments_as_HDL` directive to `off` indicates the end of the code.

You can use this directive with `translate_off` and `translate_on` to create one HDL source file that includes both a megafunction instantiation for synthesis and a behavioral description for simulation.

Because formal verification tools do not recognize the `read_comments_as_HDL` directive, it is not supported when you are using formal verification.

In Example 9–86 and Example 9–87, the commented code enclosed by `read_comments_as_HDL` is visible to the Quartus II compiler and is synthesized.

Because synthesis directives are case-sensitive in Verilog HDL, you must match the case of the directive, as shown in the following examples.

Example 9–86. Verilog HDL Code: Read Comments as HDL

```
// synthesis read_comments_as_HDL on
// my_rom lpm_rom (.address (address),
//                  .data    (data));
// synthesis read_comments_as_HDL off
```

Example 9–87. VHDL Code: Read Comments as HDL

```
-- synthesis read_comments_as_HDL on
-- my_rom : entity lpm_rom
--   port map (
--    address => address,
--    data    => data,       );
-- synthesis read_comments_as_HDL off
```

Use I/O Flipflops

This attribute directs the Quartus II software to implement input, output, and output enable flipflops (or registers) in I/O cells that have fast, direct connections to an I/O pin, when possible. Applying the `useioff` synthesis attribute can improve I/O performance by minimizing setup, clock-to-output, and clock-to-output enable times. This synthesis attribute is supported using the **Fast Input Register**, **Fast Output Register**, and **Fast Output Enable Register** logic options that can also be set in the Assignment Editor.

For more information about which device families support fast input, output, and output enable registers, refer to the device family data sheet, device handbook, or the Quartus II Help.

The `useioff` synthesis attribute takes a Boolean value and can only be applied to the port declarations of a top-level Verilog HDL module or VHDL entity (it is ignored if applied elsewhere). Setting the value to `1` (Verilog HDL) or `TRUE` (VHDL) instructs the Quartus II software to pack registers into I/O cells. Setting the value to `0` (Verilog HDL) or `FALSE` (VHDL) prevents register packing into I/O cells.

In Example 9–88 and Example 9–89, the `useioff` synthesis attribute directs the Quartus II software to implement the registers `a_reg`, `b_reg`, and `o_reg` in the I/O cells corresponding to the ports `a`, `b`, and `o`, respectively.

Example 9–88. Verilog HDL Code: the useioff Attribute

```
module top_level(clk, a, b, o);
       input clk;
       input [1:0] a, b /* synthesis useioff = 1 */;
       output [2:0] o /* synthesis useioff = 1 */;
       reg [1:0] a_reg, b_reg;
       reg [2:0] o_reg;
       always @ (posedge clk)
       begin
           a_reg <= a;
           b_reg <= b;
           o_reg <= a_reg + b_reg;
       end
       assign o = o_reg;
endmodule
```

Verilog-2001 syntax also accepts the type of statements shown in Example 9–89 and Example 9–90 instead of the comment form shown in Example 9–88.

Example 9–89. Verilog-2001 Code: the useioff Attribute

```
(* useioff = 1 *)   input [1:0] a, b;
(* useioff = 1 *)   output [2:0] o;
```

Example 9–90. VHDL Code: the useioff Attribute

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity useioff_example is
   port (
      clk  : in  std_logic;
      a, b : in  unsigned(1 downto 0);
      o    : out unsigned(1 downto 0));
   attribute useioff : boolean;
   attribute useioff of a : signal is true;
   attribute useioff of b : signal is true;
   attribute useioff of o : signal is true;
end useioff_example;
architecture rtl of useioff_example is
   signal o_reg, a_reg, b_reg : unsigned(1 downto 0);
begin
   process(clk)
   begin
      if (clk = '1' AND clk'event) then
          a_reg <= a;
          b_reg <= b;
          o_reg <= a_reg + b_reg;
      end if;
   end process;
o <= o_reg;
end rtl;
```

Specifying Pin Locations with chip_pin

This attribute enables you to assign pin locations in your HDL source. The attribute can be used only on the ports of the top-level entity or module in the design, and cannot be used to assign pin locations from entities at lower levels of the design hierarchy. You can assign pins only to single-bit or one-dimensional bus ports in your design.

For single-bit ports, the value of the `chip_pin` attribute is the name of the pin on the target device, as specified by the device's pin table.

In addition to `chip_pin`, the Quartus II software supports the `altera_chip_pin_lc` attribute name for compatibility with other synthesis tools. When using this attribute in other synthesis tools, some older device families require an "@" symbol in front of each pin assignment. In the Quartus II software, the "@" is optional.

Example 9–91 through Example 9–93 show different ways of assigning input pin `my_pin1` to Pin C1 and `my_pin2` to Pin 4 on a different target device.

Example 9–91. Verilog-1995 Code: Applying Chip Pin to a Single Pin

```
input my_pin1 /* synthesis chip_pin = "C1" */;
input my_pin2 /* synthesis altera_chip_pin_lc = "@4" */;
```

Example 9–92. Verilog-2001 Code: Applying Chip Pin to a Single Pin

```
(* chip_pin = "C1" *) input my_pin1;
(* altera_chip_pin_lc = "@4" *) input my_pin2;
```

Example 9–93. VHDL Code: Applying Chip Pin to a Single Pin

```
entity my_entity is
port(my_pin1: in std_logic; my_pin2: in std_logic;…);
end my_entity;
attribute chip_pin : string;
attribute altera_chip_pin_lc : string;
attribute chip_pin of my_pin1 : signal is "C1";
attribute altera_chip_pin_lc of my_pin2 : signal is "@4";
```

For bus I/O ports, the value of the chip pin attribute is a comma-delimited list of pin assignments. The order in which you declare the port's range determines the mapping of assignments to individual bits in the port. To leave a particular bit unassigned, simply leave its corresponding pin assignment blank.

Example 9–94 assigns `my_pin[2]` to `Pin_4`, `my_pin[1]` to `Pin_5`, and `my_pin[0]` to `Pin_6`.

Example 9–94. Verilog-1995 Code: Applying Chip Pin to a Bus of Pins

```
input [2:0]  my_pin /* synthesis chip_pin = "4, 5, 6" */;
```

Example 9–95 reverses the order of the signals in the bus, assigning `my_pin[0]` to `Pin_4` and `my_pin[2]` to `Pin_6` but leaves `my_pin[1]` unassigned.

Example 9–95. Verilog-1995 Code: Applying Chip Pin to Part of a Bus

```
input [0:2]  my_pin /* synthesis chip_pin = "4, ,6" */;
```

Example 9–96 assigns `my_pin[2]` to Pin `4` and `my_pin[0]` to Pin `6`, but leaves `my_pin[1]` unassigned.

Example 9–96. VHDL Code: Applying Chip Pin to Part of a Bus of Pins

```
entity my_entity is
port(my_pin: in std_logic_vector(2 downto 0);…);
end my_entity;

attribute chip_pin of my_pin: signal is "4, , 6";
```

Using altera_attribute to Set Quartus II Logic Options

This attribute enables you to apply Quartus II options and assignments to an object in your HDL source code. You can set this attribute on an entity, architecture, instance, register, RAM block, or I/O pin. You cannot set it on an arbitrary combinational node such as a net. With `altera_attribute`, you can control synthesis options from your HDL source even when the options lack a specific HDL synthesis attribute (such as many of the logic options presented earlier in this chapter). You can also use this attribute to pass entity-level settings and assignments to phases of the compiler flow beyond Analysis and Synthesis, such as Fitting.

Assignments or settings made through the Quartus II GUI, the **.qsf** file, or the Tcl interface take precedence over assignments or settings made with the `altera_attribute` synthesis attribute in your HDL code.

The syntax for setting this attribute in HDL is the same as the syntax for other synthesis attributes, as shown in "Synthesis Attributes" on page 9–25.

The attribute value is a single string containing a list of **.qsf** file variable assignments separated by semicolons, as shown in Example 9–97.

Example 9–97. variable Assignments Separated by Semicolons

```
-name <variable_1> <value_1>;-name <variable_2> <value_2>[;…]
```

If the Quartus II option or assignment includes a target, source, and/or section tag, use the syntax in Example 9–98 for each **.qsf** file variable assignment.

Example 9–98. Syntax for Each .qsf File Variable Assignment

```
-name <variable> <value>
-from <source> -to <target> -section_id <section>
```

The syntax for the full attribute value, including the optional target, source, and section tags for two different **.qsf** file assignments, is shown in Example 9–99.

Example 9–99. Syntax for Fill Attribute Value

```
" -name <variable_1> <value_1> [-from <source_1>] [-to <target_1>] [-section_id \
<section_1>]; -name <variable_2> <value_2> [-from <source_2>] [-to <target_2>] \
[-section_id <section_2>] "
```

If a variable's assigned value is a string of text, you must use escaped quotes around the value in Verilog HDL, or double-quotes in VHDL, as in the following examples (using non-existent variable and value terms):

Verilog HDL

```
"VARIABLE_NAME \"STRING_VALUE\""
```

VHDL

```
"VARIABLE_NAME ""STRING_VALUE"""
```

To find the **.qsf** file variable name or value corresponding to a specific Quartus II option or assignment, you can make the option setting or assignment in the Quartus II GUI and then note the changes in the **.qsf** file. You can also refer to the *Quartus II Settings File Reference Manual*, which documents all variable names.

Example 9–100 through Example 9–102 use `altera_attribute` to set the power-up level of an inferred register.

For inferred instances, you cannot apply the attribute to the instance directly, so you should apply the attribute to one of the instance's output nets. The Quartus II software moves the attribute to the inferred instance automatically.

Example 9–100. Verilog-1995 Code: Applying Altera Attribute to an Instance

```
reg my_reg /* synthesis altera_attribute = "-name POWER_UP_LEVEL HIGH"
*/;
```

Example 9–101. Verilog-2001 Code: Applying Altera Attribute to an Instance

```
(* altera_attribute = "-name POWER_UP_LEVEL HIGH" *) reg my_reg;
```

Example 9–102. VHDL Code: Applying Altera Attribute to an Instance

```
signal my_reg : std_logic;
attribute altera_attribute : string;
attribute altera_attribute of my_reg: signal is "-name POWER_UP_LEVEL
HIGH";
```

Example 9–103 through Example 9–105 use the `altera_attribute` to disable the **Auto Shift Register Replacement** synthesis option for an entity. To apply the Altera Attribute to a VHDL entity, you must set the attribute on its architecture rather than on the entity itself.

Example 9–103. Verilog-1995 Code: Applying Altera Attribute to an Entity

```
module my_entity(…) /* synthesis altera_attribute = "-name
AUTO_SHIFT_REGISTER_RECOGNITION OFF" */;
```

Example 9–104. Verilog-2001 Code: Applying Altera Attribute to an Entity

```
(* altera_attribute = "-name AUTO_SHIFT_REGISTER_RECOGNITION OFF" *)
module my_entity(…) ;
```

Example 9–105. VHDL Code: Applying Altera Attribute to an Entity

```
entity my_entity is
-- Declare generics and ports
end my_entity;
architecture rtl of my_entity is
attribute altera_attribute : string;
-- Attribute set on architecture, not entity
attribute altera_attribute of rtl: architecture is "-name
AUTO_SHIFT_REGISTER_RECOGNITION OFF";
begin
-- The architecture body
end rtl;
```

You can also use `altera_attribute` for more complex assignments involving more than one instance. In Example 9–106 through Example 9–108, the `altera_attribute` is used to cut all timing paths from `reg1` to `reg2`, equivalent to this Tcl or QSF command:

```
set_instance_assignment -name CUT ON -from reg1 -to reg2 ↵
```

Example 9–106. Verilog-1995 Code: Applying Altera Attribute with -to

```
reg reg2;
reg reg1 /* synthesis altera_attribute = "-name CUT ON -to reg2" */;
```

Example 9–107. Verilog-2001 Code: Applying Altera Attribute with -to

```
reg reg2;
(* altera_attribute = "-name CUT ON -to reg2" *) reg reg1;
```

Example 9–108. VHDL Code: Applying Altera Attribute with -to

```
signal reg1, reg2 : std_logic;
attribute altera_attribute: string;
attribute altera_attribute of reg1 : signal is "-name CUT ON -to reg2";
```

You can specify either the `-to` option or the `-from` option in a single `altera_attribute`; integrated synthesis automatically sets the remaining option to the target of the `altera_attribute`. You can also specify wildcards for either option. For example, if you specify "`*`" for the `-to` option instead of `reg2` in these examples, the Quartus II software cuts all timing paths from `reg1` to every other register in this design entity.

The `altera_attribute` can be used only for entity-level settings, and the assignments (including wildcards) apply only to the current entity.

Analyzing Synthesis Results

After you have performed synthesis, you can check your synthesis results in the **Analysis & Synthesis** section of the Compilation Report and the Project Navigator.

Analysis and Synthesis Section of the Compilation Report

The Compilation Report, which provides a summary of results for the project, appears after a successful compilation, or you can choose it from the Processing menu. After Analysis and Synthesis, before the Fitter begins, the Summary information provides a summary of utilization based on synthesis data, before Fitter optimizations have occurred. Synthesis-specific information is listed in the **Analysis & Synthesis** section.

There are various report sections under Analysis and Synthesis, including a list of the source files read for the project, the resource utilization by entity after synthesis, and information about state machines, latches, optimization results, and parameter settings.

For more information about each report section, refer to the Quartus II Help.

Project Navigator

The **Hierarchy** tab of the Project Navigator provides a summary of resource information about the entities in the project. After Analysis and Synthesis, before the Fitter begins, the Project Navigator provides a summary of utilization based on synthesis data, before Fitter optimizations have occurred.

If you hold your mouse pointer over one of the entities in the **Hierarchy** tab, a tooltip appears that shows parameter information for each instance.

Analyzing and Controlling Synthesis Messages

This section provides information about the messages generated during synthesis, and how you can control which messages appear during compilation.

Quartus II Messages

The messages that appear during Analysis and Synthesis describe many of the optimizations that the software performs during the synthesis stage, and provide information about how the design is interpreted. You should always check the messages to analyze **Critical Warnings** and **Warnings**, because these messages can relate to important design problems. It is also useful to read the information messages **Info** and **Extra Info** to get more information about how the software processes your design.

The **Info**, **Extra Info**, **Warning**, **Critical Warning**, and **Error** tabs display messages grouped by type.

You can right-click on a message in the Messages window and get help on the message, locate the source of the message in your design, and manage messages.

You can use message suppression to reduce the number of messages listed after a compilation by preventing individual messages and entire categories of messages from being displayed. For example, if you review a particular message and determine that it is not caused by something in your design that should be changed or fixed, you can suppress the message so it is not displayed during subsequent compilations. This saves time because you see only new messages during subsequent compilations.

You can right-click on an individual message in the Messages window and choose commands in the **Suppress** submenu. Another way to achieve the same goal is to open the Message Suppression Manager. To do this, right-click in the Messages window, point to **Suppress**, and click **Message Suppression Manager**.

For more information about messages and how to suppress them, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook.*

Beginning with Quartus II software version 8.1, you can specify the type of Analysis and Synthesis messages that you want to view by selecting the **Analysis & Synthesis Message Level** option. You can specify the display level by performing the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. Select the level for the **Analysis & Synthesis Message Level** option.

For more information about the **Analysis & Synthesis Message Level** option, refer to the Quartus II Help File.

VHDL and Verilog HDL Messages

The Quartus II software issues a variety of messages when it is analyzing and elaborating the Verilog HDL and VHDL files in your design. These HDL messages are a subset of all Quartus II messages that help you identify potential problems early in the design process.

HDL messages fall into the following three categories:

- **Info message**—Lists a property of your design.
- **Warning message**—Indicates a potential problem in your design. Potential problems come from a variety of sources, including typos, inappropriate design practices, or the functional limitations of your target device. Though HDL warning messages do not always identify actual problems, you should always investigate code that generates an HDL warning. Otherwise, the synthesized behavior of your design might not match your original intent or its simulated behavior.
- **Error message**—Indicates an actual problem with your design. Your HDL code can be invalid due to a syntax or semantic error, or it might not be synthesizable as written. Consult the Help associated with any HDL error messages for assistance in removing the error from your design.

In Example 9–109, the sensitivity list contains multiple copies of the variable `i`. While the Verilog HDL language does not prohibit duplicate entries in a sensitivity list, it is clear that this design has a typo: Variable `j` should be listed on the sensitivity list to avoid a possible simulation/synthesis mismatch.

Example 9–109. Generating an HDL Warning Message

```
//dup.v
module dup(input i, input j, output reg o);
always @ (i or i)
    o = i & j;
endmodule
```

When processing this HDL code, the Quartus II software generates the following warning message:

```
Warning: (10276) Verilog HDL sensitivity list warning at dup.v(2):
sensitivity list contains multiple entries for "i".
```

In Verilog HDL, variable names are case-sensitive, so the variables `my_reg` and `MY_REG` in Example 9–110 are two different variables. However, declaring variables whose names only differ in case might confuse some users, especially those users who use VHDL, where variables are not case-sensitive.

Example 9–110. Generating HDL Info Messages

```
// namecase.v
module namecase (input i, output o);
    reg my_reg;
    reg MY_REG;
    assign o = i;
endmodule
```

When processing this HDL code, the Quartus II software generates the following informational message:

```
Info: (10281) Verilog HDL information at namecase.v(3): variable name
"MY_REG" and variable name "my_reg" should not differ only in case.
```

In addition, the Quartus II software generates additional HDL info messages to inform you that neither `my_reg` or `MY_REG` are used in this small design:

```
Info: (10035) Verilog HDL or VHDL information at namecase.v(3): object
"my_reg" declared but not used
Info: (10035) Verilog HDL or VHDL information at namecase.v(4): object
"MY_REG" declared but not used
```

The Quartus II software allows you to control how many HDL messages you see during the Analysis and Elaboration of your design files. You can set the HDL Message Level to enable or disable groups of HDL messages, or you can enable or disable specific messages, as described in the following sections.

For more information about synthesis directives and their syntax, refer to "Synthesis Directives" on page 9–27.

Setting the HDL Message Level

The HDL Message Level specifies the types of messages that the Quartus II software displays when it is analyzing and elaborating your design files. Table 9–8 details the information about the HDL message levels.

Table 9–8. HDL Info Message Level

Level	Purpose	Description
Level1	Displays high-severity messages only	If you want to see only those HDL messages that identify likely problems with your design, select Level1. When Level1 is selected, the Quartus II software issues a message only if there is a high probability that it points to an actual problem with your design.
Level2	Displays high-severity and medium-severity messages	If you want to see additional HDL messages that identify possible problems with your design, select Level2. This is the default setting.
Level3	Displays all messages, including low-severity messages	If you want to see all HDL info and warning messages, select Level3. This level includes extra "LINT" messages that suggest changes to improve the style of your HDL code or make it easier to understand.

You should address all issues reported at the **Level1** setting. The default HDL message level is **Level2**.

To set the HDL Message Level in the GUI, on the Assignments menu, click **Settings**. In the **Category** list, click **Analysis & Synthesis Settings**. Set the desired message level from the pull-down menu in the **HDL Message Level** list, and click **OK**.

You can override this default setting in a source file with the `message_level synthesis` directive, which takes the values `level1`, `level2`, and `level3`, as shown in Example 9–111 and Example 9–112.

Example 9–111. Verilog HDL Examples of message_level Directive

```
// altera message_level level1
or
/* altera message_level level3 */
```

Example 9–112. VHDL Code: message_level Directive

```
-- altera message_level level2
```

A `message_level` synthesis directive remains effective until the end of a file or until the next `message_level` directive. In VHDL, you can use the `message_level` synthesis directive to set the HDL Message Level for entities and architectures, but not for other design units. An HDL Message Level for an entity applies to its architectures, unless overridden by another `message_level` directive. In Verilog HDL, you can use the `message_level` directive to set the HDL Message Level for a module.

Enabling or Disabling Specific HDL Messages by Module/Entity

You can enable or disable a specific HDL info or warning message with its Message ID, which is displayed in parentheses at the beginning of the message. Enabling or disabling a specific message overrides its HDL Message Level. This method is different from the message suppression in the Messages window because you can use this method to disable messages for a specific module or entity. This method applies only the HDL messages, and if you disable a message with this method, the message is listed as a Suppressed message in the Quartus II GUI.

To disable specific HDL messages in the GUI, on the Assignments menu, click **Settings**. In the **Category** list, expand **Analysis & Synthesis Settings** and select **Advanced**. In the **Advanced Message Settings** dialog box, add the Message IDs you wish to enable or disable.

To enable or disable specific HDL messages in your HDL, use the `message_on` and `message_off` synthesis directives. Both directives take a space-separated list of Message IDs. You can enable or disable messages with these synthesis directives immediately before Verilog HDL modules, VHDL entities, or VHDL architectures. You cannot enable or disable a message in the middle of an HDL construct.

A message enabled or disabled via a `message_on` or `message_off` synthesis directive overrides its HDL Message Level or any `message_level` synthesis directive. The message remains disabled until the end of the source file or until its status is changed by another `message_on` or `message_off` directive.

Example 9–113. Verilog HDL message_off Directive for Message with ID 10000

```
// altera message_off 10000
or
/* altera message_off 10000 */
```

Example 9–114. VHDL message_off Directive for Message with ID 10000

```
-- altera message_off 10000
```

Node-Naming Conventions in Quartus II Integrated Synthesis

Being able to find the logic node names after synthesis can be useful during verification or while debugging a design. This section provides an overview of the conventions used by the Quartus II software when it names the nodes created from your HDL design. The section focuses on the conventions for Verilog HDL and VHDL code, but AHDL and BDFs are discussed when appropriate.

Whenever possible, Quartus II integrated synthesis uses wire or signal names from your source code to name nodes such as LEs or ALMs. Some nodes, such as registers, have predictable names that typically do not change when a design is resynthesized, although certain optimizations can affect register names. The names of other nodes, particularly LEs or ALMs that contain only combinational logic, can change due to logic optimizations that the software performs.

This section discusses the following topics:

- "Hierarchical Node-Naming Conventions"
- "Node-Naming Conventions for Registers (DFF or D Flipflop Atoms)"
- "Register Changes During Synthesis" on page 9–76
- "Preserving Register Names" on page 9–78
- "Node-Naming Conventions for Combinational Logic Cells" on page 9–78
- "Preserving Combinational Logic Names" on page 9–79

Hierarchical Node-Naming Conventions

To make each name in the design unique, the Quartus II software adds the hierarchy path to the beginning of each name. The "|" separator is used to indicate a level of hierarchy. For each instance in the hierarchy, the software adds the entity name and the instance name of that entity, using the ":" separator between each entity name and its instance name. For example, if a design instantiates entity `A` with the name `my_A_inst`, the hierarchy path of that entity would be `A:my_A_inst`. The full name of any node is obtained by starting with the hierarchical instance path, followed by a "|", and ending with the node name inside that entity, using the following convention:

<entity 0>:*<instance_name 0>*|*<entity 1>*:
<instance_name 1>| . . . |*<instance_name n>*

For example, if entity A contains a register (DFF atom) called `my_dff`, its full hierarchy name would be `A:my_A_inst|my_dff`.

On the **Compilation Process Settings** page of the **Settings** dialog box, click **More Settings** and turn off **Display entity name for node name** to instruct the compiler to generate node names that do not contain the name for each level of the hierarchy. With this option off, the node names use the following convention:

<instance_name 0>|*<instance_name 1>*| . . . |*<instance_name n>*

Node-Naming Conventions for Registers (DFF or D Flipflop Atoms)

In Verilog HDL and VHDL, inferred registers are named after the `reg` or `signal` connected to the output.

© March 2009 Altera Corporation

Example 9–115 is a description of a register in Verilog HDL that creates a DFF primitive called `my_dff_out`:

Example 9–115. Verilog HDL Register

```
wire dff_in, my_dff_out, clk;

always @ (posedge clk)
my_dff_out <= dff_in;
```

Similarly, Example 9–116 is a description of a register in VHDL that creates a DFF primitive called `my_dff_out`.

Example 9–116. VHDL Register

```
signal dff_in, my_dff_out, clk;
process (clk)
begin
if (rising_edge(clk)) then
my_dff_out <= dff_in;
end if;
end process;
```

In AHDL designs, DFF registers are declared explicitly rather than inferred, so the software uses the user-declared name for the register.

For schematic designs using a **.bdf** file, all elements are given a name when they are instantiated in the design, so the software uses the user-defined name for the register or DFF.

In the special case that a wire or signal (such as `my_dff_out` in the preceding examples) is also an output pin of your top-level design, the Quartus II software cannot use that name for the register (for example, cannot use `my_dff_out`) because the software requires that all logic and I/O cells have unique names. In this case, the Quartus II integrated synthesis appends `~reg0` to the register name.

For example, the Verilog HDL code in Example 9–117 produces a register called `q~reg0`:

Example 9–117. Verilog HDL Register Feeding Output Pin

```
module my_dff (input clk, input d, output q);
always @ (posedge clk)
q <= d;
endmodule
```

This situation occurs only for registers driving top-level pins. If a register drives a port of a lower level of the hierarchy, the port is removed during hierarchy flattening and the register retains its original name, in this case, `q`.

Register Changes During Synthesis

On some occasions, you might not be able to find registers that you expect to see in the synthesis netlist. Registers might be removed by logic optimization, or their names might be changed due to synthesis optimization. Common optimizations include inference of a state machine, counter, adder-subtractor, or shift register from registers and surrounding logic. Other common register changes occur when registers are packed into dedicated hardware on the FPGA, such as a DSP block or a RAM block.

This section describes the following factors that can affect register names:

- "Synthesis and Fitting Optimizations"
- "State Machines"
- "Inferred Adder-Subtractors, Shift Registers, Memory, and DSP Functions" on page 9–77
- "Packed Input and Output Registers of RAM and DSP Blocks" on page 9–77
- "Preserving Register Names" on page 9–78
- "Preserving Combinational Logic Names" on page 9–79

Synthesis and Fitting Optimizations

Registers might be removed by synthesis logic optimization if they are not connected to inputs or outputs in the design, or if the logic can be simplified due to constant signal values. Register names might also be changed due to synthesis optimizations, such as when duplicate registers are merged together to reduce resource utilization.

NOT-gate push back optimizations can affect registers that use preset signals. This type of optimization can impact your timing assignments when registers are used as clock dividers. If this situation occurs in your design, change the clock settings to work on the new register name.

Synthesis netlist optimizations often change node names because registers can be combined or duplicated to optimize the design.

For more information about the type of optimizations performed by synthesis netlist optimizations, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

The Quartus II Compilation Report provides a list of registers that are removed during synthesis optimizations, and a brief reason for the removal. In the **Analysis & Synthesis** folder, open **Optimization Results**, and then open **Register Statistics**, and click on the **Registers Removed During Synthesis** report, and the **Removed Registers Triggering Further Register Optimizations** report. The second report contains a list of registers that are the cause of other registers being removed in the design. It provides a brief reason for the removal, and a list of registers that were removed due to the removal of the initial register.

Synthesis creates synonyms for registers duplicated with the **Maximum Fan-Out** option (or `maxfan` attribute). Therefore, timing assignments applied to nodes that are duplicated with this option are applied to the new nodes as well.

The Quartus II Fitter can also change node names after synthesis (for example, when the Fitter uses register packing to pack a register into an I/O element, or when logic is modified by physical synthesis). The Fitter creates synonyms for duplicated registers so timing analysis can use the existing node name when applying assignments.

You can instruct the Quartus II software to preserve certain nodes throughout compilation so you can use them for verification or making assignments. For more information, refer to "Preserving Register Names" on page 9–78.

State Machines

If a state machine is inferred from your HDL code, the registers that represent the states are mapped into a new set of registers that implement the state machine. Most commonly, the software converts the state machine into a one-hot form where each state is represented by one register. In this case, for Verilog HDL or VHDL designs, the registers are named according to the name of the state register and the states, where possible.

For example, consider a Verilog HDL state machine where the states are `parameter state0 = 1, state1 = 2, state2 = 3`, and where the state machine register is declared as `reg [1:0] my_fsm`. In this example, the three one-hot state registers are named `my_fsm.state0`, `my_fsm.state1`, and `my_fsm.state2`.

In AHDL, state machines are explicitly specified with a machine name. State machine registers are given synthesized names based on the state machine name but not the state names. For example, if a state machine is called `my_fsm` and has four state bits, they might be synthesized with names such as `my_fsm~12`, `my_fsm~13`, `my_fsm~14`, and `my_fsm~15`.

Inferred Adder-Subtractors, Shift Registers, Memory, and DSP Functions

The Quartus II software infers megafunctions from Verilog HDL and VHDL code for logic that forms adder-subtractors, shift registers, RAM, ROM, and arithmetic functions that can be placed in DSP blocks.

For information about inferring megafunctions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Because adder-subtractors are part of a megafunction instead of generic logic, the combinational logic exists in the design with different names. For shift registers, memory, and DSP functions, the registers and logic are typically implemented inside the dedicated RAM or DSP blocks in the device. Thus, the registers are not visible as separate LEs or ALMs.

Packed Input and Output Registers of RAM and DSP Blocks

Registers can be packed into the input registers and output registers of RAM and DSP blocks, so that they are not visible as separate registers in LEs or ALMs.

For information about packing registers into RAM and DSP megafunctions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook.*

Preserving Register Names

You might want to preserve certain register names for verification or debugging, or to ensure that timing assignments are applied correctly. Quartus II integrated synthesis preserves certain nodes automatically if they are likely to be used in a timing constraint.

Use the `preserve` attribute to instruct the compiler not to minimize or remove a specified register during synthesis optimizations or register netlist optimizations. Refer to "Preserve Registers" on page 9–44 for details.

Use the `noprune` attribute to preserve a fan-out-free register through the entire compilation flow. Refer to "Noprune Synthesis Attribute/Preserve Fan-out Free Register Node" on page 9–46 for details.

Use the synthesis attribute `syn_dont_merge` to make sure registers are not merged with other registers, and other registers are not merged with them. Refer to "Disable Register Merging/Don't Merge Register" on page 9–45 for details.

Node-Naming Conventions for Combinational Logic Cells

Whenever possible for Verilog HDL, VHDL, and AHDL code, the Quartus II software uses wire names that are the targets of assignments, but can change the node names due to synthesis optimizations.

For example, consider the Verilog HDL code in Example 9–118. Quartus II integrated synthesis uses the names `c`, `d`, `e`, and `f` for the combinational logic cells that are produced.

Example 9–118. Naming Nodes for Combinational Logic Cells in Verilog HDL

```
wire c;
reg d, e, f;

assign c = a | b;
always @ (a or b)
d = a & b;
always @ (a or b) begin : my_label
e = a ^ b;
end

always @ (a or b)
f = ~(a | b);
```

For schematic designs using a **.bdf** file, all elements are given a name when they are instantiated in the design and the software uses the user-defined name when possible.

Node naming conventions for schematic buses in the Quartus II software version 7.2 and later are different than the MAX+PLUS II software and older versions of the Quartus II software. In most cases, the Quartus II software uses the appropriate naming convention for the design source file. Designs created using the Quartus II software version 7.1 or earlier use the MAX+PLUS II naming convention. Designs created in the Quartus II software version 7.2 and later use the Quartus II naming

 © March 2009 Altera Corporation

convention that matches the behavior of standard HDLs. In some cases, however, a design might contain files created in various versions. To set an assignment for a particular instance in the Assignment Editor, enter the instance name in the **To** field, choose **Block Design Naming** from the **Assignment Name** list, and set the value to **MaxPlusII** or **QuartusII**.

If logic cells, such as those created in Example 9–118, are packed with registers in device architectures such as the Stratix and Cyclone device families, those names might not appear in the netlist after fitting. In other devices, such as newer families in the Stratix and Cyclone series device families, the register and combinational nodes are kept separate throughout the compilation, so these names are more often maintained through fitting.

When logic optimizations occur during synthesis, it is not always possible to retain the initial names as described. In some cases, synthesized names are used, which are the wire names with a tilde (~) and a number appended. For example, if a complex expression is assigned to wire `w` and that expression generates several logic cells, those cells can have names such as `w`, `w~1`, `w~2`, and so on. Sometimes the original wire name `w` is removed, and an arbitrary name such as `rtl~123` is created. It is a goal of Quartus II integrated synthesis to retain user names whenever possible. Any node name ending with ~<*number*> is a name created during synthesis, which can change if the design is changed and re-synthesized. Knowing these naming conventions can help you understand your post-synthesis results and make it easier to debug your design or make assignments.

The software maintains combinational clock logic by making sure nodes that are likely to be a clock are not changed during synthesis. The software also maintains (or "protects") multiplexers in clock trees so that the TimeQuest Timing Analyzer has information about which paths are unate, to allow complete and correct analysis of combinational clocks. Multiplexers often occur in clock trees when the design selects between different clocks. To help analysis of clock trees, the software ensures that each multiplexer encountered in a clock tree is broken into 2:1 multiplexers, and each of those 2:1 multiplexers is mapped into one look-up table (independent of the device family). This optimization might result in a slight increase in area, and for some designs a decrease in timing performance. You can turn off this multiplexer protection with the option **Clock MUX Protection** under **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box. This option applies to Arria GX devices, the Stratix and Cyclone series, and MAX II devices.

Preserving Combinational Logic Names

You might want to preserve certain combinational logic node names for verification or debugging, or to ensure that timing assignments are applied correctly.

Use the `keep` attribute to keep a wire name or combinational node name through logic synthesis minimizations and netlist optimizations. Refer to "Keep Combinational Node/Implement as Output of Logic Cell" on page 9–46 for details.

For any internal node in your design clock network, use `keep` to protect the name so that you can apply correct clock settings. Also, set the attribute on combinational logic involved in `cut` assignments and `-through` assignments.

Setting the keep attribute on combinational logic can increase the area utilization and increase the delay of the final mapped logic because it requires the insertion of extra combinational logic. Use the attribute only when necessary.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF form.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

You can specify many of the options described in this section either on an instance, global level, or both.

Use the following Tcl command to make a global assignment:

```
set_global_assignment -name <QSF Variable Name> <Value> ↵
```

Use the following Tcl command to make an instance assignment:

```
set_instance_assignment -name <QSF Variable Name> <Value>\ -to
<Instance Name> ↵
```

Adding an HDL File to a Project and Setting the HDL Version

Use the following Tcl assignments to add an HDL or schematic entry design file to your project:

```
set_global_assignment -name VERILOG_FILE <file name>.<v|sv>
set_global_assignment -name SYSTEMVERILOG_FILE <file name>.sv
set_global_assignment -name VHDL_FILE <file name>.<vhd|vhdl>
set_global_assignment -name AHDL_FILE <file name>.tdf
set_global_assignment -name BDF_FILE <file name>.bdf
```

You can use any file extension for design files, as long as you specify the correct language when adding the design file. For example, you can use **.h** for Verilog HDL header files.

To specify the Verilog HDL or VHDL version, use the following option at the end of the VERILOG_FILE or VHDL_FILE command:

```
-HDL_VERSION <language version>
```

The variable <*language version*> takes one of the following values:

- VERILOG_1995
- VERILOG_2001

- SYSTEMVERILOG_2005
- VHDL87
- VHDL93

For example, to add a Verilog HDL file called **my_file** that is written in Verilog-1995, use the following command:

```
set_global_assignment -name VERILOG_FILE my_file.v -HDL_VERSION
VERILOG_1995
```

Quartus II Synthesis Options

Table 9–9 lists the **.qsf** file variable names and applicable values for the settings discussed in this chapter. The **.qsf** file variable name is used in the Tcl assignment to make the setting along with the appropriate value.

Table 9–9. Quartus II Synthesis Options (Part 1 of 3) *(Note 1)*

Setting Name	Quartus II Settings File Variable	Values
Add Pass-Through Logic to Inferred RAMs	ADD_PASS_THROUGH_LOGIC_TO_INFERRED_ RAMS	On/Off
Allow Any RAM Size for Recognition	ALLOW_ANY_RAM_SIZE_FOR_RECOGNITION	On/Off
Allow Any ROM Size for Recognition	ALLOW_ANY_ROM_SIZE_FOR_RECOGNITION	On/Off
Allow Any Shift Register Size for Recognition	ALLOW_ANY_SHIFT_REGISTER_SIZE_FOR_ RECOGNITION	On/Off
Allow Asynchronous Clear Usage For Shift Register Replacement	ALLOW_ACLR_FOR_SHIFT_REGISTER_ RECOGNITION	On/Off
Allow Synchronous Control Signals	ALLOW_SYNCH_CTRL_USAGE	On/Off
Analysis & Synthesis Message Level	SYNTH_MESSAGE_LEVEL	Low/Medium/High
Auto Carry Chains	AUTO_CARRY_CHAINS	On/Off
Auto Clock Enable Replacement	AUTO_CLOCK_ENABLE_RECOGNITION	On/Off
Auto DSP Block Replacement	AUTO_DSP_RECOGNITION	On/Off
Auto Gated Clock Conversion	SYNTH_GATED_CLOCK_CONVERSION	On/Off
Auto Open-Drain Pins	AUTO_OPEN_DRAIN_PINS	On/Off
Auto RAM Block Balancing	AUTO_RAM_BLOCK_BALANCING	On/Off
Auto RAM to Logic Cell Conversion	AUTO_RAM_TO_LCELL_CONVERSION	On/Off
Auto RAM Replacement	AUTO_RAM_RECOGNITION	On/Off
Auto Resource Sharing	AUTO_RESOURCE_SHARING	On/Off
Auto ROM Replacement	AUTO_ROM_RECOGNITION	On/Off
Auto Shift-Register Replacement	AUTO_SHIFT_REGISTER_RECOGNITION	Always/Auto/Off
Block Design Naming	BLOCK_DESIGN_NAMING	Auto/Max+Plus II/ Quartus II
Carry Chain Length	*<device name>*_CARRY_CHAIN_LENGTH	*<Maximum allowable length of a chain>*

Table 9–9. Quartus II Synthesis Options (Part 2 of 3) *(Note 1)*

Setting Name	Quartus II Settings File Variable	Values
Clock MUX Protection	SYNTH_CLOCK_MUX_PROTECTION	On/Off
Create Debugging Nodes for IP Cores	ENABLE_IP_DEBUG	On/Off
DSP Block Balancing	DSP_BLOCK_BALANCING	Auto/DSP Blocks/ Logic Elements/ Off/Simple 18-bit Multipliers/ Simple Multipliers/Width 18-bit Multipliers
Extract Verilog State Machines	EXTRACT_VERILOG_STATE_MACHINES	On/Off
Extract VHDL State Machines	EXTRACT_VHDL_STATE_MACHINES	On/Off
Force Use of Synchronous Clear Signals	FORCE_SYNCH_CLEAR	On/Off
HDL Message Level	HDL_MESSAGE_LEVEL	Level1/Level2/ Level3
Ignore CARRY Buffers	IGNORE_CARRY_BUFFERS	On/Off
Ignore CASCADE Buffers	IGNORE_CASCADE_BUFFERS	On/Off
Ignore GLOBAL Buffers	IGNORE_GLOBAL_BUFFERS	On/Off
Ignore LCELL Buffers	IGNORE_LCELL_BUFFERS	On/Off
Ignore Maximum Fan-Out Assignments	IGNORE_MAX_FANOUT_ASSIGNMENTS	On/Off
Ignore ROW GLOBAL Buffers	IGNORE_ROW_GLOBAL_BUFFERS	On/Off
Ignore SOFT Buffers	IGNORE_SOFT_BUFFERS	On/Off
Ignore translate_off and synthesis_off directives	IGNORE_TRANSLATE_OFF_AND_SYNTHESIS_OFF	On/Off
Ignore Verilog Initial Constructs	IGNORE_VERILOG_INITIAL_CONSTRUCTS	On/Off
Iteration limit for constant Verilog loops	VERILOG_CONSTANT_LOOP_LIMIT	*<Maximum limit to infinite loops before exhaustion of memory>*
Iteration limit for non-constant Verilog loops	VERILOG_NON_CONSTANT_LOOP_LIMIT	*<Maximum limit to infinite loops before exhaustion of memory>*
Limit AHDL Integers to 32 Bits	LIMIT_AHDL_INTEGERS_TO_32_BITS	On/Off
Maximum DSP Block Usage *(2)*	MAX_BALANCING_DSP_BLOCKS	*<Maximum DSP Block Usage Value>*
Maximum Number of M4K/ M9K Memory Blocks	MAX_RAM_BLOCKS_M4K	*<Maximum memory blocks usage>*
Maximum Number of M512 Memory Blocks	MAX_RAM_BLOCKS_M512	*<Maximum memory blocks usage>*
Maximum Number of M-RAM/ M144K Memory Blocks	MAX_RAM_BLOCKS_MRAM	*<Maximum memory blocks usage>*
NOT Gate Push-Back	NOT_GATE_PUSH_BACK	On/Off
Number of Inverted Registers Reported in Synthesis Report	NUMBER_OF_INVERTED_REGISTERS_REPORTED	*<Maximum number of inverted registers>*
Number of Removed Registers Reported in Synthesis Report	NUMBER_OF_REMOVED_REGISTERS_REPORTED	*<Maximum number of inverted registers>*

Table 9–9. Quartus II Synthesis Options (Part 3 of 3) *(Note 1)*

Setting Name	Quartus II Settings File Variable	Values
Optimization Technique	*<device family>*_OPTIMIZATION_TECHNIQUE	Area/Speed/ Balanced
Parallel Synthesis	PARALLEL_SYNTHESIS	On/Off
Perform WYSIWYG Primitive Resynthesis	ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP	On/Off
PowerPlay Power Optimization	OPTIMIZE_POWER_DURING_SYNTHESIS	Normal compilation/ Extra effort/Off
Power-Up Don't Care *(2)*	ALLOW_POWER_UP_DONT_CARE	On/Off
Remove Duplicate Registers	REMOVE_DUPLICATE_REGISTERS	On/Off
Remove Redundant Logic Cells *(2)*	REMOVE_REDUNDANT_LOGIC_CELLS	On/Off
Restructure Multiplexers	MUX_RESTRUCTURE	On/Off/Auto
Safe State Machine	SAFE_STATE_MACHINE	On/Off
SDC Constraint Protection	SYNTH_PROTECT_SDC_CONSTRAINT	On/Off
Show Parameter Settings Tables in Synthesis Report	SHOW_PARAMETER_SETTINGS_TABLES_IN_ SYNTHESIS_REPORT	On/Off
State Machine Processing	STATE_MACHINE_PROCESSING	Auto/One-Hot/ Gray/Johnson/ Minimal Bits/ Sequential/ User-Encoded
Strict RAM Replacement	STRICT_RAM_RECOGNITION	On/Off
Synthesis Effort *(2)*	SYNTHESIS_EFFORT	Auto/Fast
Timing Driven Synthesis	SYNTH_TIMING_DRIVEN_SYNTHESIS	On/Off
Use LogicLock Constraints during Resource Balancing	USE_LOGICLOCK_CONSTRAINTS_IN_BALANCING	On/Off

Notes to Table 9–9:

(1) These settings are supported as Global and Instance settings, unless specified.

(2) This setting is only a Global setting.

Assigning a Pin

Use the following Tcl command to assign a signal to a pin or device location:

```
set_location_assignment -to <signal name> <location>
```

For example: `set_location_assignment -to data_input Pin_A3`

Valid locations are pin location names. Some device families also support edge and I/O bank locations. Edge locations are EDGE_BOTTOM, EDGE_LEFT, EDGE_TOP, and EDGE_RIGHT. I/O bank locations include IOBANK_1 to IOBANK_n, where n is the number of I/O banks in a particular device.

Creating Design Partitions for Incremental Compilation

To create a partition, use the following command:

```
set_instance_assignment -name PARTITION_HIERARCHY \
<file name> -to <destination> -section_id <partition name>
```

The <*destination*> should be the entity's short hierarchy path. A *short* hierarchy path is the full hierarchy path without the top-level name, for example: `"ram:ram_unit|altsyncram:altsyncram_component"` (with quotation marks). For the top-level partition, you can use the pipe (|) symbol to represent the top-level entity.

For more information about hierarchical naming conventions, refer to "Node-Naming Conventions in Quartus II Integrated Synthesis" on page 9–74.

The <*partition name*> is the user-designated partition name, which must be unique and less than 1024 characters long. The name can consist only of alphanumeric characters, as well as pipe (|), colon (:), and underscore (_) characters. Altera recommends enclosing the name in double quotation marks (" ").

The <*file name*> is the name used for internally generated netlist files during incremental compilation. Netlists are named automatically by the Quartus II software based on the instance name if you create the partition in the GUI. If you are using Tcl to create your partitions, you must assign a custom file name that is unique across all partitions. For the top-level partition, the specified file name is ignored, and you can use any dummy value. To ensure the names are safe and platform independent, file names must be unique regardless of case. For example, if a partition uses the file name `my_file`, no other partition can use the file name `MY_FILE`. For simplicity, Altera recommends that you base each file name on the corresponding instance name for the partition.

The software stores all netlists in the **db** compilation database directory.

Conclusion

The Quartus II software includes complete Verilog HDL and VHDL language support, as well as support for Altera-specific languages, making it an easy-to-use, standalone solution for Altera designs. You can use the synthesis options available in the software to help you improve your synthesis results, giving you more control over the way your design is synthesized. Use Quartus II reports and messages to analyze your compilation results.

Referenced Documents

This chapter references the following documents:

- *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Designing With Low-Level Primitives User Guide*
- *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*
- *Introduction to the Quartus II Software*
- *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*
- *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*
- *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*

- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Reference Manual*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 9–10 shows the revision history for this chapter.

Table 9–10. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Updated Table 9–9. ■ Updated the following sections: → “Partitions for Preserving Hierarchical Boundaries” on page 9–20 → “Analysis & Synthesis Settings Page of the Settings Dialog Box” on page 9–24 → “Timing-Driven Synthesis” on page 9–30 → “Turning Off Add Pass-Through Logic to Inferred RAMs/ no_rw_check Attribute Setting” on page 9–54 ■ Added “Parallel Synthesis” on page 9–21 ■ Chapter 9 was previously Chapter 8 in software version 8.1	Updated for Quartus II software version 9.0 release.

Table 9–10. Document Revision History (Part 2 of 2)

November 2008 v8.1.0	■ Changed page size to 8.5” × 11” ■ Restructured chapter by rearranging sections ■ Updated Figure 8–1 ■ Updated Table 8–9 ■ Added Example 8–23 and Example 8–28 ■ Updated the following sections: →“Setting Default Parameter Values and BDF Instance Parameter Values” →“Incremental Compilation” →“Quartus II Synthesis Options” →“Limiting DSP Block Usage in Partitions” →“Synthesis Effort” →“Using altera_attribute to Set Quartus II Logic Options” →“Quartus II Messages” ■ Added the following sections: →“Quartus II eXported Partition (.qxp) File as Source” →“Auto Gated Clock Conversion” →“Timing-Driven Synthesis” →“SDC Constraint Protection”	Updated for Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Adjusted the items listed in “System Verilog Support” ■ Added the section “VHDL wait Constructs and associated Examples” ■ Added the section “Limiting DSP Block Usage in Partitions” ■ Added the section “Synthesis Effort” ■ Added hyperlinks to referenced documents throughout the chapter ■ Minor editorial updates	Updated for Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

10. Synopsys Synplify Support

QII51009-9.0.0

Introduction

This chapter documents support for the Synopsys Synplify software in the Quartus® II software, as well as key design flows, methodologies, and techniques for achieving good results in Altera® devices. This chapter includes the following topics:

- General design flow with the Synplify and Quartus II software
- Synplify software optimization strategies, including timing-driven compilation settings, optimization options, and Altera-specific attributes
- Exporting designs and constraints to the Quartus II software using NativeLink integration
- Guidelines for Altera megafunctions and library of parameterized module (LPM) functions, instantiating them with the MegaWizard™ Plug-In Manager, and tips for inferring them from hardware description language (HDL) code
- Incremental compilation and block-based design, including the MultiPoint flow in the Synplify Pro and Synplify Premier software

The content in this chapter applies to the Synplify, Synplify Pro, and Synplify Premier software unless otherwise specified. This chapter includes the following sections:

- "Altera Device Family Support"
- "Design Flow" on page 10–2
- "Synplify Optimization Strategies" on page 10–6
- "Exporting Designs to the Quartus II Software Using NativeLink Integration" on page 10–14
- "Guidelines for Altera Megafunctions and Architecture-Specific Features" on page 10–25
- "Incremental Compilation and Block-Based Design" on page 10–37

This chapter assumes that you have set up, licensed, and are familiar with the Synplify software.

Altera Device Family Support

The Synplify software maps synthesis results to Altera device families. The following list shows the Altera device families supported by the Synplify software version C-2009.03, with the Quartus II software version 9.0:

- ACEX® 1K
- APEX™ II, APEX 20K, APEX 20KC, APEX 20KE
- Arria® GX series
- Cyclone® series
- FLEX® 10K, FLEX 6000

- HardCopy® series
- MAX® II
- MAX 7000, MAX 3000
- Stratix® series, Stratix GX series

The Synplify software also supports the Excalibur™ ARM® legacy device that is supported in the Quartus II software only with a specific license requested at www.altera.com/mysupport.

The Synplify software also supports the FLEX 8000 and MAX 9000 legacy devices that are supported only in the Altera MAX+PLUS® II software.

To learn about new device support for a specific Synplify version, refer to the release notes at www.synopsys.com.

Design Flow

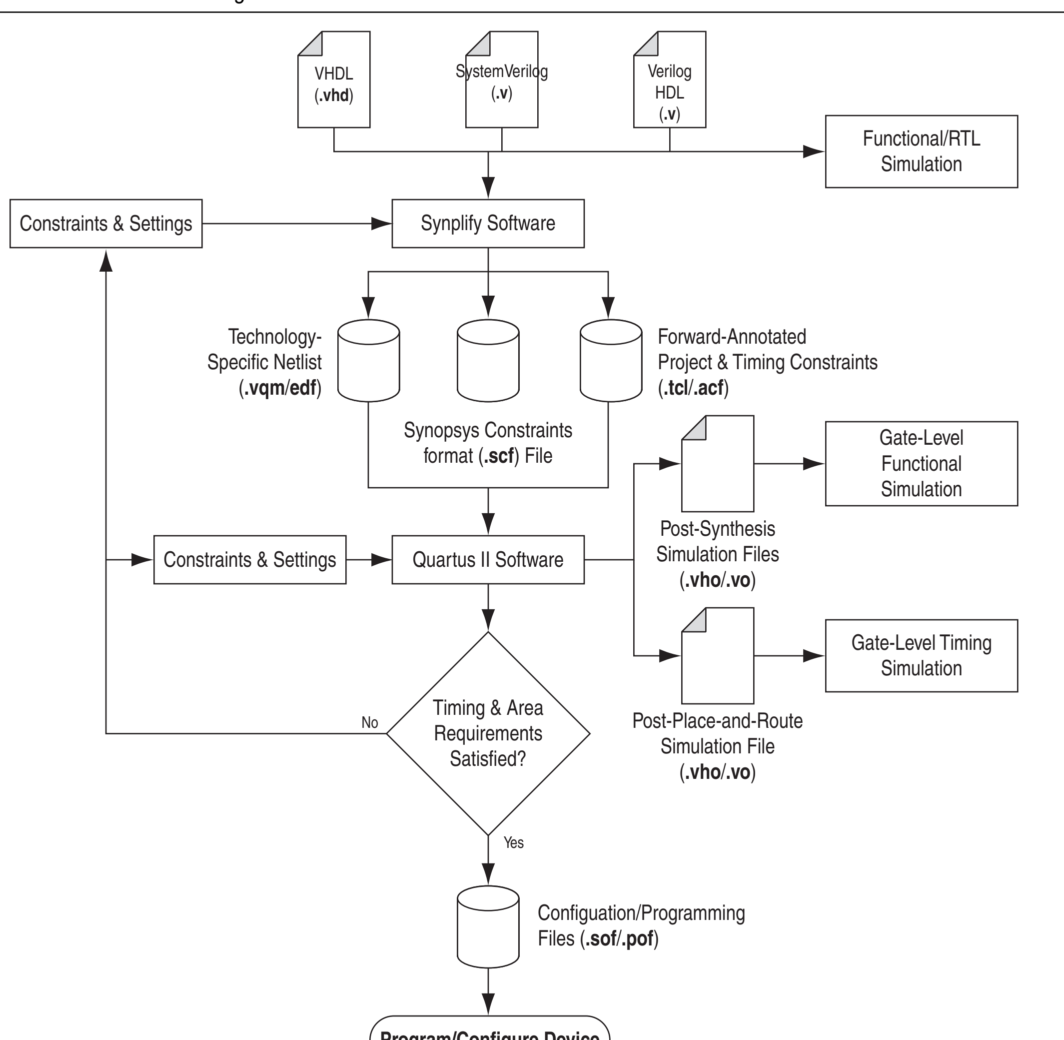
A Quartus II software design flow using the Synplify software consists of the following steps:

1. Create Verilog HDL or VHDL design files.
2. Set up a project in the Synplify software and add the HDL design files for synthesis.
3. Select a target device and add timing constraints and compiler directives in the Synplify software to optimize the design during synthesis.
4. Run synthesis in the Synplify software.
5. Create a Quartus II project and import these files generated by the Synplify software into the Quartus II software. These files are used for placement and routing, and for performance evaluation.
   - The technology-specific Verilog Quartus Mapping File (**.vqm**) netlist or EDIF (**.edf**) netlist for legacy devices also supported in the MAX+PLUS II software
   - The Synopsys Constraints Format (**.scf**) file for TimeQuest timing constraints
   - The tool command language (**.tcl**) constraint file

   Alternatively, you can run the Quartus II software from within the Synplify software. For more detailed information, refer to "Running the Quartus II Software from within the Synplify Software" on page 10–15.
6. After obtaining place-and-route results that meet your requirements, configure or program the Altera device.

Figure 10–1 shows the recommended design flow when using the Synplify and the Quartus II software.

Figure 10–1. Recommended Design Flow



The Synplify software supports VHDL, Verilog HDL, and SystemVerilog source files. However, only the Synplify Pro and Premier software supports mixed synthesis, allowing a combination of VHDL and Verilog HDL or SystemVerilog format source files.

Specify timing constraints and attributes for the design in a Synopsys Constraints File (**.scf**) with the SCOPE window in the Synplify software or directly in the HDL source file. Compiler directives can also be defined in the HDL source file. Many of these constraints are forward-annotated for use by the Quartus II software.

The HDL Analyst that is included in the Synplify software is a graphical tool for generating schematic views of the technology-independent register transfer level (RTL) view netlist (**.srs**) and technology-view netlist (**.srm**) files. You can use the Synplify HDL Analyst to analyze and debug your design visually. The HDL Analyst supports cross probing between the RTL and Technology views, the HDL source code, and the Finite State Machine (FSM) viewer. Synplify HDL Analyst also supports cross-probing between the technology view and the timing report file in the Quartus II software.

A separate license file is required to enable the HDL Analyst in the Synplify software. The Synplify Pro and Premier software include the HDL Analyst.

After synthesis is completed, import the **.vqm** or **.edf** netlist to the Quartus II software for place-and-route. Use the **.tcl** file generated by the Synplify software to forward-annotate your constraints (including device selection) and optionally to set up your project in the Quartus II software.

If you select a Stratix III, Cyclone III, Arria GX, or newer device, the Quartus II software uses the SDC-format timing constraints from the **.scf** file with the TimeQuest Timing Analyzer by default. If you select a Stratix II or Stratix II GX device, you have the option to switch from the Classic Timing Analyzer to the TimeQuest Timing Analyzer by turning on the **Use TimeQuest Timing Analyzer** option in the **Device** tab in the **Implementation Options** dialog box in the Synplify software. For other devices, the Quartus II software uses the Tcl-format timing constraints from the Quartus Setting File (**.qsf**) with the Classic Timing Analyzer. Refer to "Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File" on page 10–16 for information about manually changing from the TimeQuest Timing Analyzer to the Classic Timing Analyzer in the Quartus II software.

If the area and timing requirements are satisfied, use the files generated by the Quartus II software to program or configure the Altera device. As shown in Figure 10–1, if your area or timing requirements are not met, you can change the constraints in the Synplify software or the Quartus II software and repeat the synthesis. Altera recommends that you provide the timing constraints as much as possible at the Synplify software level and the placement constraints at the Quartus II software level. Repeat the process until the area and timing requirements are met.

While you can perform simulation at various points in the process, you can also perform final timing analysis after placement and routing is complete. You can also perform formal verification at various stages of the design process.

For more information about how the Synplify software supports formal verification, refer to *Section III. Formal Verification* in volume 3 of the *Quartus II Handbook*.

You can also use other options and techniques in the Quartus II software to meet area and timing requirements. One such option is called WYSIWYG Primitive Resynthesis, which can perform optimizations on your **.vqm** netlist within the Quartus II software.

For information about netlist optimizations, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

In some cases, you might be required to modify the source code if the area and timing requirements cannot be met using options in the Synplify and Quartus II software.

After synthesis, the Synplify software produces several intermediate and output files. Table 10–1 lists these file types.

Table 10–1. Synplify Intermediate and Output Files (Part 1 of 2)

File Extensions	File Description
.srs	Technology-independent RTL netlist that can be read only by the Synplify software
.srm	Technology view netlist

Table 10–1. Synplify Intermediate and Output Files (Part 2 of 2)

File Extensions	File Description
.srr *(1)*	Synthesis Report file
.edf/.vqm	Technology-specific netlist in **.edf** or **.vqm** file format An **.edf** file is created for ACEX 1K, FLEX 10K, FLEX 10KA, FLEX 10KE, FLEX 6000, FLEX 8000, MAX 7000, MAX 9000, and MAX 3000 devices. A **.vqm** file is created for all other Altera device families.
.acf/.tcl	Forward-annotated constraints file containing constraints and assignments. A **.tcl** file for the Quartus II software is created for all devices. The **.tcl** file contains the appropriate Tcl commands to create and set up a Quartus II project and pass placement constraints. If applicable, the MAX+PLUS II assignments are imported from the **.acf** file.
.scf	Synopsys Constraint Format file containing timing constraints for the TimeQuest Timing Analyzer

Note to Table 10–1:

(1) This report file includes performance estimates that are often based on pre-place-and-route information. Use the f_{MAX} reported by the Quartus II software after place-and-route—it is the only reliable source of timing information. This report file includes post-synthesis device resource utilization statistics that might inaccurately predict resource usage after place-and-route. The Synplify software does not account for black box functions nor for logic usage reduction achieved through register packing performed by the Quartus II software. Register packing combines a single register and look-up table (LUT) into a single logic cell, reducing logic cell utilization below the Synplify software estimate. Use the device utilization reported by the Quartus II software after place-and-route.

Output Netlist File Name and Result Format

Specify the output netlist directory location and name for the Synplify software by performing the following steps:

1. On the Project menu, click **Implementation Options**.
2. Click the **Implementation Results** tab.
3. In the **Results Directory** box, type your output netlist file directory location.
4. In the **Result File Name** box, type your output netlist file name.

By default, the directory and file name are set to the project implementation directory and the top-level design module or entity name.

The **Result Format and Quartus Version** options are also available on the **Implementation Results** tab. The **Result Format** list specifies an **.edf** or **.vqm** netlist, depending on your device family. The software creates an **.edf** output netlist file only for ACEX 1K, FLEX 10K, FLEX 10KA, FLEX 10KE, FLEX 6000, FLEX 8000, MAX 7000, MAX 9000, and MAX 3000 devices. For other Altera devices, the software generates a **.vqm**-formatted netlist.

Select the version of the Quartus II software that you are using in the **Quartus Version** list. This option ensures that the netlist is compatible with the software version and supports the newest features. Altera recommends using the latest version of the Quartus II software whenever possible. If your Quartus II software is newer than the versions available in the **Quartus Version** list, check if there is a newer version of the Synplify software available that supports the current Quartus II software version. Otherwise, choose the latest version in the list for the best compatibility.

The **Quartus Version** list is available only after selecting an Altera device.

To set the Quartus II software version used in the Synplify software, perform the following steps:

1. In the Synplify software, on the Project menu, click **Implementation Options**.
2. Click the **Implementation Results** tab, then click **Quartus Version**.
3. Choose the correct version number in the list.

Alternatively, use the following command from the command line:

```
set_option -quartus_version <version number> ↵
```

Synplify Optimization Strategies

As designs become more complex and require increased performance, using different optimization strategies has become important. Combining Synplify software constraints with VHDL and Verilog HDL coding techniques and Quartus II software options can help you obtain the required results.

For additional design and optimization techniques, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 and the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

The Synplify software offers many constraints and optimization techniques to improve your design's performance. The Synplify Pro and Premier software add additional techniques that are not supported in the basic Synplify software. This section provides an overview of some of the techniques you can use to help improve the quality of your results.

For more information about applying the attributes discussed in this section, refer to the *Altera Constraints, Attributes, and Options* chapter of the *Synopsys FPGA Synthesis Reference Manual*.

Using Synplify Premier to Optimize Your Design

Synplify Premier offers additional physical synthesis optimizations than the other Synplify products. After regular logic synthesis, Synplify Premier places and routes the design and attempts to restructure the netlist based on the physical location of the logic in the Altera device. Synplify Premier forward-annotates the design netlist to the Quartus II software to perform the final placement and routing. In the default flow, Synplify Premier also forward-annotates placement information for the critical path(s) in the design, which can improve the compilation time in the Quartus II software.

The physical location annotation file is called <*design name*>**_plc.tcl**. If you call the Quartus II software from the Synplify Premier user interface, the Quartus II software automatically uses this file for the placement information.

The Physical Analyst allows you to examine the placed netlist from Synplify Premier, similar to the HDL Analyst for a logical netlist. You can use this display to analyze and diagnose possible problems.

Implementations in Synplify Pro or Premier

To create different synthesis results without overwriting the other results, in the Synplify Pro or Premier software, on the Project menu, click **New Implementation**. For each implementation, specify the target device, synthesis options, and constraint files. Each implementation generates its own subdirectory that contains all the resulting files, including **.vqm**/**.edf**, **.scf**, and **.tcl** files, from a compilation of the particular implementation. You can then compare the results of the different implementations to find the optimal set of synthesis options and constraints for a design.

Timing-Driven Synthesis Settings

The Synplify software supports timing-driven synthesis with user-assigned timing constraints to optimize the performance of the design. The Synplify software optimizes the design to attempt to meet these constraints.

The Quartus II NativeLink feature allows timing constraints that are applied in the Synplify software to be forward-annotated for the Quartus II software using either a **.tcl** script file or a **.scf** file for timing-driven place and route. Refer to "Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File" on page 10–16 or "Passing Constraints to the Quartus II Software using Tcl Commands" on page 10–18 for more details about how constraints such as clock frequencies, false paths, and multicycle paths are forward-annotated. This section explains some of the important timing constraints in the Synplify software.

The Synplify Synthesis Report File (**.srr**) contains timing reports of estimated place-and-route delays. The Quartus II software can perform further optimizations on a post-synthesis netlist from third-party synthesis tools. In addition, designs might contain black boxes or intellectual property (IP) functions that have not been optimized by the third-party synthesis software. Actual timing results are obtained only after the design has gone through full placement and routing in the Quartus II software. For these reasons, the Quartus II post place-and-route timing reports provide a more accurate representation of the design. Use the statistics in these reports to evaluate design performance.

Clock Frequencies

For single-clock designs, specify a global frequency when using the push-button flow. While this flow is simple and provides good results, often it does not meet the performance requirements for more advanced designs. You can use timing constraints, compiler directives, and other attributes to help optimize the performance of a design. You can enter these attributes and directives directly in the HDL code. Alternatively, you can enter attributes (not directives) into an **.sdc** file with the SCOPE window in the Synplify software.

Use the SCOPE window to set global frequency requirements for the entire design and individual clock settings. Use the **Clocks** tab in the SCOPE window to specify frequency (or period), rise times, fall times, duty cycle, and other settings. Assigning individual clock settings, rather than over-constraining the global frequency, helps the Quartus II software and the Synplify software achieve the fastest clock frequency for the overall design. The `define_clock` attribute assigns clock constraints.

Multiple Clock Domains

The Synplify software can perform timing analysis on unrelated clock domains. Each clock group is a different clock domain and is treated as unrelated to the clocks in all other clock groups. All the clocks in a single clock group are assumed to be related and the Synplify software automatically calculates the relationship between the clocks. You can assign clocks to a new clock group or put related clocks in the same clock group by using the **Clocks** tab in the SCOPE window or with the `define_clock` attribute.

Input and Output Delays

Specify the input and output delays for the ports of a design in the **Input/Output** tab of the SCOPE window or with the `define_input_delay` and `define_output_delay` attributes. The Synplify software does not allow you to assign the t_{CO} and t_{SU} values directly to inputs and outputs. However, a t_{CO} value can be inferred by setting an external output delay; a t_{SU} value can be inferred by setting an external input delay.

Equation 10–1 illustrates the relationship between t_{CO} and the output delay:

Equation 10–1.

$$t_{CO} = \text{clock period} - \text{external output delay}$$

Equation 10–2 illustrates the relationship between t_{SU} and the input delay:

Equation 10–2.

$$t_{SU} = \text{clock period} - \text{external input delay}$$

When the `syn_forward_io_constraints` attribute is set to 1, the Synplify software passes the external input and output delays to the Quartus II software using NativeLink integration. The Quartus II software then uses the external delays to calculate the maximum system frequency.

Multicycle Paths

Specify any multicycle paths in the design in the **Multi-Cycle Paths** tab of the SCOPE window or with the `define_multicycle_path` attribute. A multicycle path is a path that requires more than one clock cycle to propagate. You must specify which paths are multicycle to avoid having the Quartus II and the Synplify compilers work excessively on a non-critical path. Not specifying these paths can also result in an inaccurate critical path being reported during timing analysis.

False Paths

False paths are paths that should not be considered during timing analysis or which should be assigned low (or no) priority during optimization. Some examples of false paths are slow asynchronous resets and test logic added to the design. Set these paths in the **False Paths** tab of the SCOPE window. Use the `define_false_path` attribute.

FSM Compiler

If the FSM Compiler is turned on, the compiler automatically detects state machines in a design. The compiler can then extract and optimize the state machine. The FSM Compiler analyzes the state machine and decides to implement sequential, gray, or one-hot encoding based on the number of states. It also performs unused-state analysis, optimization of unreachable states, and minimization of transition logic.

If the FSM Compiler is turned off, the compiler does not optimize logic as state machines. The state machines are implemented as coded in the HDL code. Thus, if the coding style for the state machine was sequential, the implementation is also sequential. If the FSM Compiler is turned on, the compiler infers and optimizes the state machines. The implementation is based on the number of states regardless of the coding style in the HDL code.

Use the `syn_state_machine` complier directive to specify or prevent a state machine from being extracted and optimized. To override the default encoding of the FSM Compiler, use the `syn_encoding` directive.

The values for the `syn_encoding` directive are shown in Table 10–2.

Table 10–2. syn_encoding Directive Values

Value	Description
Sequential	Generates state machines with the fewest possible flipflops. Sequential, also called binary, state machines are useful for area-critical designs when timing is not the primary concern.
Gray	Generates state machines where only one flipflop changes during each transition. Gray-encoded state machines tend to be free of glitches.
One-hot	Generates state machines containing one flipflop for each state. One-hot state machines typically provide the best performance and shortest clock-to-output delays. However, one-hot implementations are usually larger than sequential implementations.
Safe	Generates extra control logic to force the state machine to the reset state if an invalid state is reached. You can use the safe value in conjunction with the other three values, which results in the state machine being implemented with the requested encoding scheme and the generation of the reset logic.

Example 10–1 shows sample VHDL code for applying the `syn_encoding` directive.

Example 10–1. Sample VHDL Code for syn_encoding

```
SIGNAL current_state : STD_LOGIC_VECTOR(7 DOWNTO 0);
ATTRIBUTE syn_encoding : STRING;
ATTRIBUTE syn_encoding OF current_state : SIGNAL IS "sequential";
```

The default is to optimize state machine logic for speed and area, but this is potentially undesirable for critical systems. The safe value generates extra control logic to force the state machine to the reset state if an invalid state is reached.

FSM Explorer in Synplify Pro and Premier

The Synplify Pro and Premier software can use FSM Explorer to explore different encoding styles for a state machine automatically, and then implement the best encoding based on the overall design constraints. FSM Explorer uses the FSM Compiler to identify and extract state machines from a design. However, unlike the FSM Compiler that chooses the encoding style based on the number of states, the FSM Explorer tries several different encoding styles before choosing a specific one. The trade-off is that the compilation requires more time to perform the analysis of the state machine, but finds an optimal encoding scheme for the state machine.

Optimization Attributes and Options

The following sections describe other attributes and options that you can modify in the Synplify software to improve your design performance.

Retiming in Synplify Pro and Premier

The Synplify Pro and Premier software can retime a design, which can improve the timing performance of sequential circuits by moving registers (register balancing) across combinational elements. Be aware that retimed registers incur name changes. To retime your design, turn on the **Retiming** option in the **Device** tab in the **Implementation Options** section, or use the `syn_allow_retiming` attribute.

Maximum Fan-Out

When your design has critical path nets with high fan-out, use the `syn_maxfan` attribute to control the fan-out of the net. Setting this attribute for a specific net results in the replication of the driver of the net to reduce overall fan-out. The `syn_maxfan` attribute takes an integer value and applies it to inputs or registers. (The `syn_maxfan` attribute cannot be used to duplicate control signals. The minimum allowed value of the attribute is 4.) Using this attribute might result in increased logic resource utilization, thus putting a strain on routing resources and leading to long compile times and difficult fitting.

If you must duplicate an output register or output enable register, you can create a register for each output pin by using the `syn_useioff` attribute (refer to “Register Packing”).

Preserving Nets

During synthesis, the compiler maintains ports, registers, and instantiated components. However, some nets cannot be maintained to create an optimized circuit. Applying the `syn_keep` directive overrides the optimization of the compiler and preserves the net during synthesis. The `syn_keep` directive takes a Boolean value and can be applied to wires (Verilog HDL) and signals (VHDL). Setting the value to **true** preserves the net through synthesis.

Register Packing

Altera devices allow for the packing of registers into I/O cells. Altera recommends allowing the Quartus II software to make the I/O register assignments. However, you can control register packing with the `syn_useioff` attribute. The `syn_useioff` attribute takes a Boolean value and can be applied to ports or entire modules. Setting the value to **1** instructs the compiler to pack the register into an I/O cell. Setting the value to **0** prevents register packing in both the Synplify and Quartus II software.

Resource Sharing

The Synplify software uses resource sharing techniques during synthesis by default to reduce area. Turning off the **Resource Sharing** option on the **Options** tab of the **Implementation Options** dialog box improves performance results for some designs. You can also turn off the option for a specific module with the `syn_sharing` attribute. If you turn off this option, be sure to check the results to determine if it helps the timing performance. If it does not help, leave **Resource Sharing** turned on.

Preserving Hierarchy

The Synplify software performs cross-boundary optimization by default. This results in the flattening of the design to allow optimization. Use the `syn_hier` attribute to over-ride the default compiler settings. The `syn_hier` attribute takes a string value and applies it to modules, architectures, or both. Setting the value to **hard** maintains the boundaries of a module, architecture, or both, but allows constant propagation. Setting the value to **locked** prevents all cross-boundary optimizations. Use the **locked** setting with the partition setting to create separate design blocks and multiple output netlists for incremental compilation, as described in "Using MultiPoint Synthesis with Incremental Compilation" on page 10–39.

By default, the Synplify software generates a hierarchical **.vqm** file. To flatten the file, set the `syn_netlist_hierarchy` attribute to **0**.

Register Input and Output Delays

The advanced options called `define_reg_input_delay` and `define_reg_output_delay` can speed up paths feeding a register or coming from a register by a specific number of nanoseconds. The Synplify software attempts to meet the global clock frequency goals for a design as well as the individual clock frequency goals (set with `define_clock`). You can use these attributes to add delay to paths feeding into or out of registers to further constrain critical paths. The setting also works with negative numbers, so you can slow down a path that is too highly optimized.

These options are useful to close timing when your design does not meet timing goals because the routing delay after placement and routing exceeds the delay predicted by the Synplify software. Rerun synthesis using these options, specifying the actual routing delay (from place-and-route results) so that the tool can meet the required clock frequency. Synopsys recommends that for best results, do not make these assignments too aggressively. For example, increase the routing delay value but don't use the full routing delay from the last compilation.

In the SCOPE constraint window, use the registers panel with the following entries:

- **Register—**Specifies the name of the register. If you have initialized a compiled design, choose the name from the list.

- **Type**—Specifies whether the delay is an input or output delay.
- **Route**—Shrinks the effective period for the constrained registers by the specified value without affecting the clock period that is forward-annotated to the Quartus II software.

Use the following Tcl command syntax to specify an input or output register delay in nanoseconds.

Example 10–2. Specifying an Input or Output Register Delay Using Tcl Command Syntax

```
define_reg_input_delay {<register>}  -route <delay in ns>
define_reg_output_delay {<register>} -route <delay in ns>
```

syn_direct_enable

This attribute controls the assignment of a clock-enable net to the dedicated enable pin of a register. Using this attribute, you can direct the Synplify mapper to use a particular net as the only clock enable when the design has multiple clock enable candidates.

You can also use this attribute as a compiler directive to infer registers with clock enables. To do so, enter the `syn_direct_enable` directive in your source code, not the SCOPE spreadsheet.

The `syn_direct_enable` data type is Boolean. A value of **1** or **true** enables net assignment to the clock-enable pin. The following is the syntax for Verilog HDL:

```
object /* synthesis syn_direct_enable = 1 */ ;
```

Standard I/O Pad

For certain Altera devices and the equivalent device I/O standard, specify the I/O standard type to use for the I/O pad in the design using the **I/O Standard** panel in the Synplify SCOPE window.

Example 10–3 shows the Synplify SDC syntax for the `define_io_standard` constraint, in which the `delay_type` must be either `input_delay` or `output_delay`.

Example 10–3. Synplify SDC Syntax for the define_io_standard Constraint

```
define_io_standard [-disable|-enable] {<objectName>} -delay_type \
[input_delay|output_delay] <columnTclName>{<value>} [<columnTclName>{<value>}...]
```

For details about supported I/O standards, refer to *Altera I/O Standards* in the *Synopsys FPGA Synthesis Reference Manual.*

Altera-Specific Attributes

The following attributes are for use with specific Altera device features. These attributes are forward-annotated to the Quartus II project and are used during the place-and-route process.

altera_chip_pin_lc

Use this attribute to make pin assignments. This attribute takes a string value and applies it to inputs and outputs. Use the attribute only on the ports of the top-level entity in the design. Do not use this attribute to assign pin locations from entities at lower levels of the design hierarchy.

This attribute is not supported for any of the MAX series devices. In the SCOPE window, select the attribute **altera_chip_pin_lc** and set the value to a pin number or a list of pin numbers.

Example 10–4 shows VHDL code for making location assignments to ACEX 1K and FLEX 10KE devices.

The "@" sign prefix is used to specify pin locations for only ACEX 1K and FLEX 10KE devices. For these devices, pin location assignments are written to the output **.edf** file.

Example 10–4. Making Location Assignments to ACEX 1K and FLEX 10KE Devices, VHDL

```
ENTITY sample (data_in : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
   data_out: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
ATTRIBUTE altera_chip_pin_lc : STRING;
ATTRIBUTE altera_chip_pin_lc OF data_out : SIGNAL IS "@14, @5,@16, @15";
```

Example 10–5 shows VHDL code for making location assignments for other Altera devices. Pin location assignments for these devices are written to the output Tcl script.

Example 10–5. Making Location Assignments to Other Devices, VHDL

```
ENTITY sample (data_in : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
   data_out: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
ATTRIBUTE altera_chip_pin_lc : STRING;
ATTRIBUTE altera_chip_pin_lc OF data_out : SIGNAL IS "14, 5, 16, 15";
```

The `data_out` signal is a 4-bit signal; `data_out[3]` is assigned to pin 14 and `data_out[0]` is assigned to pin 15.

altera_implement_in_esb or altera_implement_in_eab

Use these attributes to implement logic in either embedded system blocks (ESBs) or embedded array blocks (EABs) rather than in logic resources to improve area utilization. The modules selected for such implementation cannot have feedback paths, and either all or none of the I/Os must be registered. This attribute takes a boolean value and can be applied to instances. (This option is applicable for devices with ESBs/EABs only. For example, the Stratix device family is not supported by this option. This attribute is ignored for designs targeting devices that do not have ESBs or EABs.)

altera_io_powerup

Use this attribute to define the power-up value of an I/O register that has no set or reset. This attribute takes a string value (**high** | **low**) and applies it to ports that have I/O registers. By default, the power-up value of the I/O is set to **low**.

altera_io_opendrain

Use this attribute to specify open-drain mode I/O ports. This attribute takes a boolean value and applies it to outputs or bidirectional ports for devices that support open-drain mode.

Exporting Designs to the Quartus II Software Using NativeLink Integration

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools, and allows you to run other EDA design entry or synthesis, simulation, and timing analysis tools automatically from within the Quartus II software. After a design is synthesized in the Synplify software, a **.vqm** or **.edf** netlist file, an **.scf** file for TimeQuest Timing Analyzer timing constraints, and **.tcl** files are used to import the design into the Quartus II software for place-and-route. You can run the Quartus II software from within the Synplify software or as a stand-alone application. After you have imported the design into the Quartus II software, you can specify different options to further optimize the design.

When you are using NativeLink integration, the path to your project must not contain white space. The Synplify software uses Tcl scripts to communicate with the Quartus II software, and the Tcl language does not accept arguments with white space in the path.

Use NativeLink integration to integrate the Synplify software and Quartus II software with a single GUI for both synthesis and place-and-route operations. NativeLink integration allows you to run the Quartus II software from within the Synplify software GUI or to run the Synplify software from within the Quartus II software GUI.

This section explains the different NativeLink flows and provides details about how constraints are passed to the Quartus II software. This section describes the following topics:

- "Running the Quartus II Software from within the Synplify Software" on page 10–15
- "Using the Quartus II Software to Run the Synplify Software" on page 10–15
- "Running the Quartus II Software Manually Using the Synplify-Generated Tcl Script" on page 10–16
- "Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File" on page 10–16
- "Passing Constraints to the Quartus II Software using Tcl Commands" on page 10–18

Running the Quartus II Software from within the Synplify Software

To use the Quartus II software from within the Synplify software, you must first verify that the `QUARTUS_ROOTDIR` environment variable contains the Quartus II software installation directory located at <*Altera Design Suite Installation Directory*>**\quartus**. This environment variable is required to use the Synplify and Quartus II software together.

In the Windows operating system, the `QUARTUS_ROOTDIR` variable is set when you open the Quartus II user interface, so it is automatically set to the most recent version you opened in the user interface. If your software installation is located on another machine, ensure that you set this variable correctly. You can change the variable manually using the Control Panel, System icon.

On UNIX and Linux operating systems, the variable is not set automatically, so you must create an environment variable `QUARTUS_ROOTDIR` that points to the <*Altera Design Suite Installation Directory*>**/quartus** location.

Under each implementation in the Synplify Pro software, create a place-and-route implementation called `pr_`<*number*> `Altera Place and Route`. You can create new place and route implementations using the **New P&R** button in the GUI. To run the Quartus II software in command-line mode after each synthesis run, use the text box to turn on the place-and-route implementation. The results of the place and route are written to a log file in the **pr_**<*number*> directory under the current implementation directory.

You can also use the commands in the Quartus II menu to run the Quartus II software at any time following a successful completion of synthesis. In the Synplify software, on the Options menu, click **Quartus II** and then choose one of the following commands:

- **Launch Quartus**—Opens the Quartus II software GUI and creates a Quartus II project with the synthesized output file, forward-annotated timing constraints, and pin assignments. Use this command to configure options for the project and execute any Quartus II commands.
- **Run Background Compile**—Runs the Quartus II software in command-line mode with the project settings from the synthesis run. The results of the place-and-route are written to a log file.

The <*project_name*>**_cons.tcl** file is used to set up the Quartus II project and calls the <*project_name*>**.tcl** file to pass constraints from the Synplify software to the Quartus II software. By default, the <*project_name*>**.tcl** file contains device, timing, and location assignments. If the project is set up to use the TimeQuest Timing Analyzer, the <*project_name*>**.tcl** file contains the command to use the Synplify-generated **.scf** constraints file with TimeQuest instead of using the Tcl constraints with the Classic Timing Analyzer.

Using the Quartus II Software to Run the Synplify Software

You can set up the Quartus II software to run the Synplify software for synthesis using NativeLink integration. This feature allows you to use the Synplify software to quickly synthesize a design as part of a normal compilation in the Quartus II software. When you use this feature, the Synplify software does not use any timing constraints or assignments such as incremental compilation partitions that you have set in the Quartus II software.

For best results, Synopsys recommends that you set constraints in the Synplify software and use the Tcl script to pass these constraints to the Quartus II software, instead of calling Synplify from within the Quartus II software.

To set up Synplify in the Quartus II software, on the Tools menu, click **Options**. In the **Options** dialog box, click **EDA Tool Options** and specify the path of Synplify or Synplify Pro software.

For detailed information about using NativeLink integration with the Synplify software, refer to the Quartus II Help.

Beginning with the Quartus II software version 7.1, running the Synplify software with NativeLink integration is supported on both floating network and node-locked single-PC licenses. Both types of licenses support batch mode compilation.

Running the Quartus II Software Manually Using the Synplify-Generated Tcl Script

You can also use the Quartus II software separately from the Synplify software. To run the Tcl script generated by the Synplify software to set up your project and set up assignments such as the device selection, perform the following steps:

1. Ensure the **.vqm**/**.edf**, **.scf** (if you are using the TimeQuest Timing Analyzer timing constraints), and **.tcl** files are located in the same directory (they are located in the implementation directory by default).
2. In the Quartus II software, on the View menu, point to **Utility Windows** and click **Tcl Console.** The Quartus II Tcl Console opens.
3. At the Tcl Console command prompt, type the following:

```
source <path>/<project name>_cons.tcl ↵
```

Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File

The TimeQuest Timing Analyzer is a powerful ASIC-style timing analysis tool that validates the timing performance of all logic in your design using an industry standard constraints format, Synopsys Design Constraints (SDC). This section explains how timing constraints set in the Synplify software are passed to the Quartus II software for use with the TimeQuest Timing Analyzer.

The timing constraints you set in the Synplify software are stored in the Synplify Design Constraints (**.sdc**) file. The **.tcl** file always contains all other constraints for the Quartus II software, such as the device specification and any location constraints. The timing constraints are forward-annotated using the **.tcl** file for the Quartus II Classic Timing Analyzer, as described in "Passing Constraints to the Quartus II Software using Tcl Commands" on page 10–18. For the TimeQuest Timing Analyzer, the timing constraints are forward-annotated in the Synopsys Constraints Format (**.scf**) file.

Altera recommends that you use the TimeQuest Timing Analyzer, as specified in the Synplify **.tcl** file that sets up the Quartus II project for the newest devices. However, you can use the Tcl commands for the Classic Timing Analyzer if required. You can manually change from the TimeQuest Timing Analyzer to the Classic Timing Analyzer in the Quartus II software by performing the following steps:

1. From the Assignments menu, click **Settings**.

2. In the **Category** list, select **Timing Analysis Settings**.
3. Under **Timing analysis processing**, select **Use Classic Timing Analyzer during compilation**.
4. Click **OK**.

For additional information about the TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Synopsys recommends that you modify constraints using the SCOPE constraint editor window and not through the generated **.sdc**, **.scf**, or **.tcl** file.

The following list of Synplify constraints are converted to the equivalent Quartus II SDC commands and are forward-annotated to the Quartus II software in the **.scf** file:

- `define_clock`
- `define_input_delay`
- `define_output_delay`
- `define_multicycle_path`
- `define_false_path`

All Synplify constraints described in the following sections use the same Synplify commands as described in "Passing Constraints to the Quartus II Software using Tcl Commands" on page 10–18; however, the constraints are mapped to SDC commands for the TimeQuest Timing Analyzer.

For syntax and arguments for these commands, refer to the applicable subsection or refer to Synplify Help. For a list of corresponding commands in the Quartus II software, refer to the Quartus II Help.

Individual Clocks and Frequencies

Specify clock frequencies for individual clocks in the Synplify software with the command `define_clock`. This command is passed to the Quartus II software with `create_clock`.

Input and Output Delay

Specify input delay and output delay constraints in the Synplify software with the commands `define_input_delay` and `define_output_delay`, respectively. These commands are passed to the Quartus II software with `set_input_delay` and `set_output_delay`.

Multicycle Path

Specify a multicycle path constraint in the Synplify software with the command `define_multicycle_path`. This command is passed to the Quartus II software with `set_multicycle_path`.

False Path

Specify a false path constraint in the Synplify software with the command `define_false_path`. This command is passed to the Quartus II software with `set_false_path`.

Passing Constraints to the Quartus II Software using Tcl Commands

This section describes how Synplify constraints are converted to the equivalent Quartus II assignments and are forward-annotated to the Quartus II software with Tcl commands.

This section also describes timing constraints for the Quartus II Classic Timing Analyzer. If you are using the TimeQuest Timing Analyzer, the Quartus II timing constraints described in this section do not apply. Refer to "Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File" on page 10–16 for information about timing constraints supported by TimeQuest.

Global Signals

The Synplify software automatically promotes clock signals to global routing lines and passes Global Signal assignments to the Quartus II software. The assignments ensure that the same global routing constraints are applied during placement and routing.

The signals promoted to global routing can be different than the ones that the Quartus II software promotes to global routing by default. The Synplify software promotes only clock signals and not other control signals such as reset or enable. By default, without constraints from the Synplify software, the Quartus II software promotes control signals to global routing if they have high fan-out.

Default or Global Clock Frequency

Use the following Synplify command to set the Synplify default or global clock frequency that applies to the entire project:

```
set_option -frequency <frequency>
```

The <*frequency*> is specified in MHz. If a global frequency is not specified, the software uses the default global clock frequency of 1 MHz.

The `set_option` constraint is passed to the Quartus II software with the following command:

```
set_global_assignment -name FMAX_REQUIREMENT <frequency> MHz
```

If a frequency is not specified in the Quartus II software, the software uses the default global clock frequency of 1 GHz.

Individual Clocks and Frequencies

Specify clock frequencies for individual clocks with the following Synplify commands as shown in Example 10–6.

Example 10–6. Specifying Clock Frequencies for Individual Clocks

```
define_clock -name {<clock_name>} -freq <frequency> -clockgroup <clock_group> -rise <rise_time>\
-fall <fall_time>
define_clock -name {<clock_name>} -period <period> -clockgroup <clock_group> -rise <rise_time>\
-fall <fall_time>
```

Table 10–3 shows the command arguments.

 © March 2009 Altera Corporation

Table 10–3. Command Arguments

Argument	Description
`-name`	The <*clock_name*> specifies a design port name or register output signal name and, after synthesis, corresponds to a <*mapped_clock_name*>.
`-freq` *(1)*	The <*frequency*> is specified in MHz.
`-period` *(2)*	The <*period*> is specified in ns.
`-clockgroup`	If the <*clock_group*> is not specified, it defaults to `default_clkgroup`. The Synplify software assumes all clocks belonging to the same clock group are related. If you do not specify a clock group, the clock belongs to the default clock group. Therefore, if you do not specify any clock groups, all the clocks are considered related by default in the software.
`-rise` `-fall`	The <*rise_time*> and <*fall_time*> specify a non-default duty cycle. By default, the Synplify synthesis tool assumes that the clock is a 50% duty cycle clock, with the rising edge at 0 and the falling edge at period/2. If you have another duty clock cycle, you can specify the appropriate **Rise At** and **Fall At** values.

Notes to Table 10–3:

(1) When the <*frequency*> is specified, the Synplify software uses <*fall_time*>and <*frequency*> to calculate the `duty_cycle` with the following formula: `duty_cycle` = (<*fall_time*> – <*rise_time*>) × <*frequency*> / 100.

(2) When the <*period*> is specified, the Synplify software uses <*fall_time*> and <*period*> to calculate the `duty_cycle` with the following formula: `duty_cycle` = 100 × (<*fall_time*> – <*rise_time*>) / <*period*>.

The equivalent Quartus II Classic Timing Analyzer commands depend on how the clock groups are defined. In the Quartus II software, clocks that belong to the same or related clock settings are considered related clocks. Clocks assigned to unrelated clock settings are unrelated clocks. There is a one-to-one correspondence between each Quartus II clock setting and a Synplify clock group.

The following sections describe only the frequency constraints. Use the corresponding constraints for the period.

Virtual Clocks

The Quartus II software supports virtual clocks. If you use the virtual clock setting in the Synplify software, the setting is mapped to a constraint in the Quartus II software.

Route Delay Option

The `-route` option in the Synplify software clock constraints is designed for use in synthesis only if you do not meet timing goals because the routing delay after placement and routing exceeds the delay predicted by the Synplify software. This constraint does not have to be forward-annotated to the Quartus II software.

Multiple Clocks in Different Clock Groups

You can specify clock frequencies for multiple clocks with the Synplify commands shown in Example 10–7.

Example 10–7. Specifying Clock Frequencies for Multiple Clocks

```
define_clock -name {<clock_name1>} -freq <frequency1> -clockgroup <clock_group1> \
-rise <rise_time1> -fall <fall_time1>

define_clock -name {<clock_name2>} -freq <frequency2> -clockgroup <clock_group2> \
-rise <rise_time2> -fall <fall_time2>
```

<clock_group1> and *<clock_group2>* are unique names defined in the Synplify software for base clock settings in the Quartus II Classic Timing Analyzer.

If the clock *<rise_time>* is zero (“0”), multiple separate clocks are passed to the Quartus II software with the commands shown in Example 10–8.

Example 10–8. Quartus II Assignments for Multiple Clocks if the Clock Rise Time is Zero

```
create_base_clock -fmax <frequency1>MHz -duty_cycle <duty_cycle1> \
-target mapped_clock_name1 <base_clock_setting1>

create_base_clock -fmax <frequency2>MHz -duty_cycle <duty_cycle2> \
-target mapped_clock_name2 <base_clock_setting2>
```

If the clock *<rise_time>* is non-zero, multiple separate clocks are passed to the Quartus II software with the following commands shown in Example 10–9.

Example 10–9. Quartus II Assignments for Multiple Clocks if the Clock Rise Time is Not Zero

```
create_base_clock -fmax <frequency1>MHz -duty_cycle <duty cycle1> -no_target <base clock setting1>

create_base_clock -fmax <frequency2>MHz -duty_cycle <duty cycle2> -no_target <base clock setting2>

create_relative_clock -base_clock <base clock setting1> -offset <rise time1>ns \
-duty_cycle <duty cycle1> -multiply <multiply by> -divide <divide by> \
-target <mapped clock name1> <derived clock setting1>

create_relative_clock -base_clock <base clock setting2> -offset <rise time2>ns \
-duty_cycle <duty cycle2> -multiply <multiply by> -divide <divide by> \
-target <mapped clock name2> <derived clock_setting2>
```

Multiple Clocks with Different Frequencies in the Same Clock Group

In the Synplify software, you can specify multiple clocks with relative clock settings in the same clock group with different frequencies, with the commands shown in Example 10–10.

Example 10–10. Specifying Multiple Clocks with Different Frequencies in the Same Clock Group

```
define_clock -name {<clock_name1>} -freq <frequency1> -clockgroup <clock_group1> \
-rise <rise_time1> -fall <fall_time1>

define_clock -name {<clock_name2>} -freq <frequency2> -clockgroup <clock_group2> \
-rise <rise_time2> -fall <fall_time2>
```

When you specify clocks with different frequencies in the same clock group, the software calculates the *<multiply_by>* and the *<divide_by>* factors for relative clock settings from *<frequency1>* and *<frequency2>* in the clock group settings.

If the clock *<rise_time>* is zero, multiple clocks with relative clock settings in the same clock group with different frequencies are passed to the Quartus II software with the commands shown in Example 10–11.

Example 10–11. Quartus II Assignments for Multiple Clocks with Different Frequencies in the Same Clock Group, if the Clock Rise Time is Zero

```
create_base_clock -fmax <frequency1>MHz -duty_cycle <duty_cycle1> \
-target <mapped_clock_name1> <base_clock_setting1>

create_relative_clock -base_clock <base_clock_setting1> -duty_cycle <duty_cycle2> \
-multiply <multiply_by> -divide <divide_by> -target <mapped_clock_name2> <derived_clock_setting2>
```

Inter-Clock Relationships—Delays and False Paths between Clocks

Set a clock-to-clock delay constraint in Synplify with the commands in Example 10–12.

Example 10–12. Specifying Clock-to-Clock Delay Constraints

```
define_clock_delay -fall <clock_name1> -rise <clock_name2> <delay_value>
define_clock_delay -rise <clock_name1> -fall <clock_name2> <delay_value>
define_clock_delay -rise <clock_name1> -rise <clock_name2> <delay_value>
define_clock_delay -fall <clock_name1> -fall <clock_name2> <delay_value>
```

If <*delay_value*> is set to `false`, these constraints in Synplify indicate a false path between the two clocks. If all four rise/fall clock-edge pairs are specified in the Synplify software, the Synplify constraints are mapped to the following constraint in the Quartus II software:

```
set_timing_cut_assignment -from <clock_name1> -to <clock_name2>
```

If all four clock-edge pairs are not specified in Synplify, the constraint cannot be mapped to a constraint for the Quartus II Classic Timing Analyzer.

If <*delay_value*> is set to a value other than `false`, these constraints in Synplify are not mapped to constraints in the Quartus II software. The Quartus II Classic Timing Analyzer does not support clock-edge to clock-edge delay constraints.

False Paths

Specify the false path constraint in the Synplify software with the following command:

```
define_false_path -from <sig_name1> -to <sig_name2>
```

The signals <*sig_name1*> and <*sig_name1*> can be design port names or register instance names.

The **define_false_path** constraint in the Synplify software is mapped to the constraint in the Quartus II software, as shown in the following command:

```
set_timing_cut_assignment -from <sig_name2> -to <sig_name2>
```

The Synplify software can identify pairs of signal sets such that every member of the cross-product of these two sets is a valid false path constraint. Signal groups can be defined in the Quartus II Classic Timing Analyzer with the following commands:

```
timegroup -add_member sig_name1_i <sig_group1>
(for every signal in <sig_group1>)
timegroup -add_member sig_name2_i <sig_group2>
(for every signal in <sig_group2>)
set_timing_cut_assignment -from <sig_group1> -to <sig_group2>
```

If the signals <*sig_name1*> or <*sig_name2*> represent multiple signals such as a wildcard, group, or bus, the constraints can be appropriately expanded for representation in the Quartus II software. The Quartus II software supports wildcard signal names, and signal groups for timing assignments. The Quartus II software does not support bus notation, such as **A[7:4]**.

False Path from a Signal

Specify a false path constraint from a signal in the Synplify software with the following command:

```
define_false_path -from <sig_name>
```

The Quartus II Classic Timing Analyzer does not support "from-only" path specifications. You must also include a "to-path" specification. However, you can specify a wildcard for the `-to` signal. This constraint in Synplify is mapped to the following constraint in the Quartus II software:

```
set_timing_cut_assignment -from <sig_name> -to {*}
```

False Path to a Signal

Specify a false path constraint to a signal in the Synplify software with the following command:

```
define_false_path -to <sig_name>
```

The Quartus II Classic Timing Analyzer does not support "to-only" path specifications. You must include a "from-path" specification. However, you can specify a wildcard for the `-from` signal. This constraint in the Synplify software is mapped to the following constraint in the Quartus II software:

```
set_timing_cut_assignment -from {*} -to <sig_name>
```

False Path through a Signal

Specify a false path constraint through a signal in the Synplify software with the following command:

```
define_false_path -from <sig_name1> -to <sig_name2> \
-through <sig_name3>
```

The Quartus II Classic Timing Analyzer does not support false paths with a "through path" specification. Any constraint in the Synplify software with a `-through` specification is not mapped to a constraint for the Quartus II Classic Timing Analyzer.

Multicycle Paths

Specify a multicycle path constraint in the Synplify software with the following command:

```
define_multicycle_path -from <sig_name1> -to <sig_name2> <clock_cycles>
```

This constraint in the Synplify software is mapped to the following constraint in the Quartus II software:

```
set_multicycle_assignment -from <sig_name1> \
-to <sig_name2> <clock_cycles>
```

If the signals <*sig_name1*> or <*sig_name2*> represent multiple signals such as a wildcard, group, or bus, the constraints can be appropriately expanded for representation in the Quartus II software as described in "False Paths" on page 10–8.

<clock_cycles> is the number of clock cycles for the multicycle path.

Multicycle Path from a Signal

Specify a multicycle path constraint from a signal in the Synplify software with the following command:

```
define_multicycle_path -from <sig_name> <clock_cycles>
```

This constraint is mapped using a wildcard for the `-to` value in the Quartus II Classic Timing Analyzer, similar to the false path constraints:

```
set_multicycle_assignment -from <sig_name> -to {*} <clock_cycles>
```

Multicycle Path to a Signal

Specify a multicycle path constraint to a signal in the Synplify software with the following command:

```
define_multicycle_path -to <sig_name> <clock_cycles>
```

This constraint is mapped using a wildcard for the `-from` value in the Quartus II Classic Timing Analyzer, similar to the false path constraints:

```
set_multicycle_assignment -from {*} -to <sig_name> <clock_cycles>
```

Multicycle Path through a Signal

Specify a multicycle path constraint through a signal in the Synplify software using the following command:

```
define_multicycle_path -from <sig_name1> -to <sig_name2> \
-through <sig_name3> <clock_cycles>
```

The Quartus II Classic Timing Analyzer does not support multicycle paths with a "through path" specification. Any constraint in the Synplify software with a `-through` specification is not mapped to a constraint for the Quartus II Classic Timing Analyzer.

Maximum Path Delays

Specify the maximum path delay relationships between signals in the Synplify software with the following command:

```
define_path_delay -from <sig_name1> -to <sig_name2> -max <delay_value>
```

This constraint in the Synplify software is mapped to the following constraint in the Quartus II software:

```
set_instance_assignment -from <sig_name1> \
-to <sig_name2> -name SETUP_RELATIONSHIP <delay_value>ns
```

The Quartus II Classic Timing Analyzer does not support signal groups or bus notation. It supports only register names for this constraint.

Maximum Path Delay from a Signal

Specify the maximum path delay constraint from a signal in the Synplify software with the following command:

```
define_path_delay -from <sig_name> -max <delay_value>
```

This constraint is mapped using a wildcard for the `-to` value in the Quartus II Classic Timing Analyzer, similar to false path constraints:

```
set_instance_assignment -from <sig_name> -to {*} \
-name SETUP_RELATIONSHIP <delay_value>ns
```

Maximum Path Delay to a Signal

Specify the maximum path delay constraint to a signal in the Synplify software with the following command:

```
define_path_delay -to <sig_name> -max <delay_value>
```

This constraint is mapped using a wildcard for the `-from` value in the Quartus II Classic Timing Analyzer, similar to the false path constraints.

```
set_instance_assignment -from {*}<sig_name> \
-to <sig_name> -name SETUP_RELATIONSHIP <delay_value>ns
```

Maximum Path Delay through a Signal

Specify the maximum path delay constraint through a signal in the Synplify software with the following command:

```
define_path_delay -from <sig_name1> -to <sig_name2> \
-through <sig_name3> -max <delay_value>
```

The Quartus II Classic Timing Analyzer does not support maximum path delay constraints with a "through path" specification. Any constraint in Synplify with a `-through` specification is not mapped to a constraint for the Quartus II Classic Timing Analyzer.

Register Input and Output Delays

These register input delay and register output delay constraints in the Synplify software are for use in synthesis only, and therefore are not forward-annotated to the Quartus II software.

Default External Input Delay

Specify the default input delay constraint in the Synplify software with the following command:

```
define_input_delay -default <delay_value>
```

This constraint is mapped to the following constraint in the Quartus II software:

```
set_input_delay -clock {*} <delay_value> {*}
```

Port-Specific External Input Delay

Specify a port-specific input delay constraint in the Synplify software with the following command:

```
define_input_delay <input_port_name> <delay_value> \
-ref <clock_name>:<clock_edge>
```

The *<clock_edge>* can be set to **`r`** (rising edge) or **`f`** (falling edge).

When the clock edge is **`r`** (rising edge), this constraint is mapped to the following constraint in the Quartus II software:

```
set_input_delay -clock <clock_name> <delay_value> <input_port_name>
```

When the <*clock_edge*> is **f** (falling edge), this constraint is not mapped to a constraint in the Quartus II software. The Quartus II Classic Timing Analyzer does not support the specification of input delays with respect to the falling edge of the clock.

Default External Output Delay

Specify the default output delay constraint in the Synplify software with the following command:

```
define_output_delay -default <delay_value>
```

This constraint is mapped to the following constraint in the Quartus II software:

```
set_output_delay -clock {*} <delay_value> {*}
```

Port-Specific External Output Delay

Specify a port-specific input delay constraint in the Synplify software with the following command:

```
define_output_delay <output_port_name> <delay_value> \
-ref <clock_name>:<clock_edge>
```

The <*clock_edge*> can be set to **r** (rising edge) or **f** (falling edge). When the clock edge is **r** (rising edge), this constraint is mapped to the following constraint in the Quartus II software:

```
set_output_delay -clock <clock_name> <delay_value> <output_port_name>
```

When the clock_edge is **f** (falling edge), this constraint is not mapped to a constraint in the Quartus II software. The Quartus II Classic Timing Analyzer does not support the specification of output delays with respect to the falling edge of the clock.

Guidelines for Altera Megafunctions and Architecture-Specific Features

Altera provides parameterizable megafunctions including LPMs, device-specific Altera megafunctions, IP available as Altera MegaCore® functions, and IP available through the Altera Megafunction Partners Program (AMPP℠). You can use megafunctions and IP functions by instantiating them in your HDL code, or you can infer certain megafunctions from generic HDL code.

If you want to instantiate a megafunction in your HDL code, you can do so with the MegaWizard Plug-In Manager to parameterize the function or by instantiating the function using the port and parameter definition. The MegaWizard Plug-In Manager provides a graphical interface within the Quartus II software for customizing and parameterizing any available megafunction for the design. For more information about the MegaWizard Plug-In Manager flow with the Synplify software, refer to "Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager" on page 10–26 and "Instantiating Intellectual Property Using the MegaWizard Plug-In Manager and IP Toolbench" on page 10–28.

For more information about specific Altera megafunctions, refer to the Quartus II Help. For more information about IP functions, refer to the appropriate IP documentation.

The Synplify software also automatically recognizes certain types of HDL code and infers the appropriate megafunction when a megafunction provides optimal results. The Synplify software provides options to control inference of certain types of megafunctions, as described in "Inferring Altera Megafunctions from HDL Code" on page 10–31.

For a detailed discussion about instantiating versus inferring megafunctions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. This chapter also provides details about using the MegaWizard Plug-In Manager in the Quartus II software and explains the files generated by the wizard, as well as providing coding style recommendations and HDL examples for inferring megafunctions in Altera devices.

Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager

This section describes how to instantiate Altera megafunctions using the MegaWizard Plug-In Manager.

When you use the MegaWizard Plug-In Manager to set up and parameterize a megafunction, the MegaWizard Plug-In Manager creates a VHDL or Verilog HDL wrapper file <*output file*>**.v|vhd** that instantiates the megafunction.

The Synplify software makes use of the Quartus II timing and resource estimation netlist feature to report more accurate resource utilization and timing performance estimates, and take better advantage of timing-driven optimization than treating the megafunction as a "black box". Include the MegaWizard-generated megafunction variation wrapper file in your Synplify project so the Synplify software has all the information about the megafunction.

There is an option in the MegaWizard Plug-In Manager to generate a netlist for resource and timing estimation. This option is not recommended for the Synplify software because the software generates this information in the background without a separate netlist. If you do create a separate netlist <*output file*>**_syn.v** and use that file in your synthesis project, you must also include the <*output file*>**.v|vhd** file in your Quartus II project.

Make sure to set the correct Quartus II version in the Synplify software before compiling the MegaWizard-generated file so the software uses the correct library definitions for the megafunction. The **Quartus Version** setting must match the version of the Quartus II software used to generate the customized megafunction in the MegaWizard Plug-In Manager.

For details about how to set the Quartus II version in the Synplify software, refer to "Output Netlist File Name and Result Format" on page 10–5.

In addition, ensure that the `QUARTUS_ROOTDIR` environment variable is set to the installation directory location of the correct Quartus II version. The Synplify software uses this information to launch the Quartus II software in the background. The environment variable setting must match the version of the Quartus II software used to generate the customized megafunction in the MegaWizard Plug-In Manager. Refer to "Using the Quartus II Software to Run the Synplify Software" on page 10–15 for details.

Using MegaWizard Plug-In Manager-Generated Verilog HDL Files for Megafunction Instantiation

If you check the <*output file*>**_inst.v** option on the last page of the wizard, the MegaWizard Plug-In Manager generates a Verilog HDL instantiation template file for use in your Synplify design. The instantiation template file, <*output file*>**_inst.v**, helps to instantiate the megafunction variation wrapper file, <*output file*>**.v**, in your top-level design. Include the megafunction variation wrapper file <*output file*>**.v** in your Synplify project. The Synplify software includes the megafunction information in the output **.vqm** netlist file. There is no need to include the MegaWizard-generated megafunction variation wrapper file in your Quartus II project.

Using MegaWizard Plug-In Manager-Generated VHDL Files for Megafunction Instantiation

If you check the <*output file*>**.cmp** and <*output file*>**_inst.vhd** options on the last page of the wizard, the MegaWizard Plug-In Manager generates a VHDL component declaration file and a VHDL instantiation template file for use in your Synplify design. These files can help you instantiate the megafunction variation wrapper file, <*output file*>**.vhd**, in your top-level design. Include <*output file*>**.vhd** in your Synplify project. The Synplify software includes the megafunction information in the output **.vqm** netlist file. There is no need to include the MegaWizard-generated megafunction variation wrapper file in your Quartus II project.

Changing Synplify's Default Behavior for Instantiated Altera Megafunctions

By default, the Synplify software automatically calls the Quartus II software in the background to generate a resource and timing estimation netlist for megafunctions, as described in the previous sections.

You might want to change this behavior to reduce run times in the Synplify software (because generating the netlist files can take several minutes for large designs), or if the Synplify software cannot access your Quartus II software installation to generate the files. Changing this behavior might speed up the compilation time in the Synplify software, but the Quality of Results (QoR) might be reduced.

The Synplify software calls the Quartus II software to generate information in two ways:

- Some megafunctions provide a "clear box" model—Synplify software can fully synthesize this model and include the device architecture-specific primitives in the output **.vqm** netlist file.
- Other megafunctions provide a "grey box" model—Synplify can read the resource information but the netlist does not contain all the logic functionality.

For these functions, the Synplify software uses the logic information for resource and timing estimation and optimization, and then instantiates the megafunction in the output **.vqm** netlist file so the Quartus II software can implement the appropriate device primitives. By default, the Synplify software uses the clear box model when available, and otherwise uses the grey box model. To change this behavior, click **Implementation Options**, and on the **Device** tab, change the Altera Models setting. The default is **on**. To enable clear box models but not grey box, select **clearbox_only**, or to turn off the feature entirely, choose **off**.

Instantiating Intellectual Property Using the MegaWizard Plug-In Manager and IP Toolbench

Many Altera IP functions include a resource and timing estimation netlist that the Synplify software uses to report more accurate resource utilization and timing performance estimates, and take better advantage of timing-driven optimization than a black box function.

To create this netlist file, first select the IP function in the MegaWizard Plug-In Manager and click **Next** to open the IP Toolbench. Click **Step 2: Set Up Simulation**, which sets up all the EDA options. Enable the **Generate netlist** option to generate a netlist for resource and timing estimation. The netlist file is generated when you click **Step 3: Generate**.

The Quartus II software generates a file <*output file*>**_syn.v**. This netlist contains the "grey box" information for resource and timing estimation, but does not contain the actual implementation. Include this netlist file in your Synplify project. Next, include the megafunction variation wrapper file <*output file*>**.v|vhd** in the Quartus II project along with your Synplify **.vqm** output netlist.

If your IP function does not include a resource and timing estimation netlist, the Synplify software must treat the IP function as a black box. In this case, refer to the following subsections for details about creating black boxes.

For information about including Quartus II-specific files in your Synplify project so they are automatically passed to the Quartus II software along with the output **.vqm** file, refer to "Including Files for Quartus II Placement and Routing Only" on page 10–30.

Using Generated Verilog HDL Files for Black Box IP Function Instantiation

Use the `syn_black_box` compiler directive to declare a module as a black box. The top-level design files must contain the IP port mapping and a hollow-body module declaration. Apply the `syn_black_box` directive to the module declaration in the top-level file or a separate file included in the project to instruct the Synplify software that this is a black box. The software compiles successfully without this directive, but reports an additional warning message. Using this directive allows you to add other directives, as discussed in "Other Synplify Software Attributes for Creating Black Boxes" on page 10–29.

Example 10–13 shows a sample top-level file that instantiates **my_verilogIP.v**, which is a simplified customized variation generated by the MegaWizard Plug-In Manager and IP Toolbench.

Example 10–13. Sample Top-Level Verilog HDL Code with Black Box Instantiation of IP

```
module top (clk, count);
   input clk;
   output[7:0]  count;
   my_verilogIP verilogIP_inst (.clock (clk), .q (count));
endmodule
// Module declaration
// The following attribute is added to create a
// black box for this module.
module my_verilogIP (clock, q) /* synthesis syn_black_box */;
   input clock;
   output[7:0]  q;
endmodule
```

Using Generated VHDL Files for Black Box IP Function Instantiation

Use the syn_black_box compiler directive to declare a component as a black box. The top-level design files must contain the megafunction variation component declaration and port mapping. Apply the syn_black_box directive to the component declaration in the top-level file. The software compiles successfully without this directive, but reports an additional warning message. Using this directive allows you to add other directives, such as the ones in the "Other Synplify Software Attributes for Creating Black Boxes" section.

Example 10–14 shows a sample top-level file that instantiates **my_vhdlIP.vhd**, which is a simplified customized variation generated by the MegaWizard Plug-In Manager and IP Toolbench.

Example 10–14. Sample Top-Level VHDL Code with Black Box Instantiation of IP

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY top IS
    PORT (
        clk: IN STD_LOGIC ;
        count: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
    );
END top;

ARCHITECTURE rtl OF top IS
COMPONENT my_vhdlIP
    PORT (
        clock: IN STD_LOGIC ;
        q: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
    );
end COMPONENT;
attribute syn_black_box : boolean;
attribute syn_black_box of my_vhdlIP: component is true;
BEGIN
    vhdlIP_inst : my_vhdlIP PORT MAP (
        clock => clk,
        q => count
    );
END rtl;
```

Other Synplify Software Attributes for Creating Black Boxes

Instantiating a function as a black box methodology does not provide the synthesis tool any visibility into the function module. Thus, it does not take full advantage of the synthesis tool's timing-driven optimization. For better timing optimization, especially if the black box does not have registered inputs and outputs, add timing models to black boxes. This can be done by adding the syn_tpd, syn_tsu, and syn_tco attributes. Refer to Example 10–15 for a Verilog HDL example.

Example 10–15. Adding Timing Models to Black Boxes in Verilog HDL

```
module ram32x4(z,d,addr,we,clk);
    /* synthesis syn_black_box syn_tco1="clk->z[3:0]=4.0"
        syn_tpd1="addr[3:0]->z[3:0]=8.0"
        syn_tsu1="addr[3:0]->clk=2.0"
        syn_tsu2="we->clk=3.0" */
    output[3:0]z;
    input[3:0]d;
    input[3:0]addr;
    input we
    input clk
endmodule
```

The following additional attributes are supported by the Synplify software to communicate details about the characteristics of the black box module within the HDL code:

- `syn_resources`—Specifies the resources used in a particular black box
- `black_box_pad_pin`—Prevents mapping to I/O cells
- `black_box_tri_pin`—Indicates a tri-stated signal

For more information about applying these attributes, refer to the *Altera Constraints, Attributes, and Options* chapter of the *Synopsys FPGA Synthesis Reference Manual*.

Including Files for Quartus II Placement and Routing Only

In the Synplify software, you can add files to your project that are used only during placement and routing in the Quartus II software. This can be useful if you have grey boxes or black boxes for Synplify synthesis that require the full design files to be compiled in the Quartus II software.

Add the files to the Synplify project like other source files. Then right-click on the file and click **File options**. Enable the **Use for Place and Route Only** option. You can also set the option in a script using the `-job_owner par` option.

For example, the commands in Example 10–16 define files for a Synplify project that includes a top-level design file, a grey box netlist file, an IP wrapper file, and an encrypted IP file. With these files, the Synplify software writes an empty instantiation of "core" in the **.vqm** file and uses the grey box netlist for resource and timing estimation. The files **core.v** and **core_enc8b10b.v** are not compiled by Synplify and are copied into the place-and-route directory. The Quartus II software compiles these files to implement the "core" IP block.

Example 10–16. Commands to Define Files for a Synplify Project

```
add_file -verilog -job_owner par "core_enc8b10b.v"
add_file -verilog -job_owner par "core.v"
add_file -verilog "core_gb.v"
add_file -verilog "top.v"
```

Inferring Altera Megafunctions from HDL Code

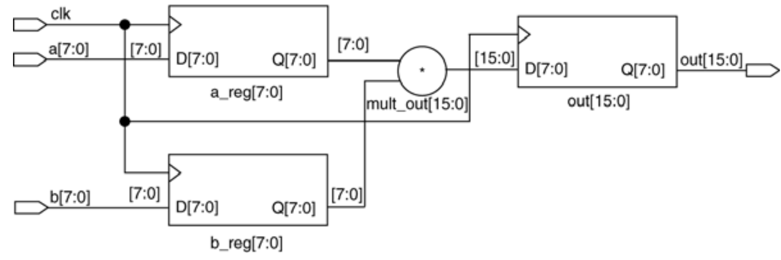
The Synplify software uses Behavior Extraction Synthesis Technology (BEST) algorithms to infer high-level structures such as RAMs, ROMs, operators, FSMs, and DSP multiplication operations. It then keeps the structures abstract for as long as possible in the synthesis process. This allows the use of technology-specific resources to implement these structures by inferring the appropriate Altera megafunction when a megafunction provides optimal results. The following sections outline some of the Synplify-specific details when inferring Altera megafunctions. The Synplify software provides options to control inference of certain types of megafunctions, which is also described in the following sections.

For coding style recommendations and examples for inferring megafunctions in Altera devices, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Inferring Multipliers

Figure 10–2 shows the HDL Analyst view of an unsigned 8 × 8 multiplier with two pipeline stages after synthesis as seen in HDL Analyst in the Synplify software. This multiplier is converted into an ALTMULT_ADD or ALTMULT_ACCUM megafunction. For devices with DSP blocks, the software might implement the function in a DSP block instead of regular logic, depending on device utilization. For certain devices, the software maps directly to DSP block device primitives instead of instantiating a megafunction in the **.vqm** file.

Figure 10–2. HDL Analyst View of LPM_MULT Megafunction (Unsigned 8 × 8 Multiplier with Pipeline=2)



Resource Balancing

While mapping multipliers to DSP blocks, the Synplify software performs resource balancing for optimum performance.

Altera devices have a fixed number of DSP blocks, which include a fixed number of embedded multipliers. If the design uses more multipliers than are available, the Synplify software automatically maps the extra multipliers to logic elements (LEs), or adaptive logic modules (ALMs).

If a design uses more multipliers than are available in the DSP blocks, the Synplify software maps the multipliers in the critical paths to DSP blocks. Next, any wide multipliers, which might or might not be in the critical paths, are mapped to DSP blocks. Smaller multipliers and multipliers that are not in the critical paths might then be implemented in the logic (LEs or ALMs). This ensures that the design fits successfully in the device.

Controlling the Inferring of DSP Blocks

You can implement multipliers in DSP blocks or in logic in Altera devices that contain DSP blocks. You can control this implementation through attribute settings in the Synplify software.

Signal Level Attribute

You can control the implementation of individual multipliers by using the `syn_multstyle` attribute as shown in the following Verilog HDL code:

```
<signal_name> /* synthesis syn_multstyle = "logic" */;
```

where *signal_name* is the name of the signal.

This setting applies to wires only; it cannot be applied to registers.

Table 10–4 shows the values for the signal level attribute in the Synplify software that controls the implementation of the multipliers in the DSP blocks or LEs.

Table 10–4. Attribute Settings for DSP Blocks in the Synplify Software

Attribute Name	Value	Description
`syn_multstyle`	`lpm_mult`	LPM function inferred and multipliers implemented in DSP blocks
`syn_multstyle`	`logic`	LPM function not inferred and multipliers implemented LEs by the Synplify software
`syn_multstyle`	`block_mult`	DSP megafunction is inferred and multipliers are mapped directly to DSP block device primitives (for supported devices)

Example 10–17 and Example 10–18 show simple Verilog HDL and VHDL code using the `syn_multstyle` attribute.

Example 10–17. Signal Attributes for Controlling DSP Block Inference in Verilog HDL Code

```
module mult(a,b,c,r,en);
    input [7:0] a,b;
    output [15:0] r;
    input [15:0] c;
    input en;
    wire [15:0] temp /* synthesis syn_multstyle="logic" */;

    assign temp = a*b;
    assign r = en ? temp : c;
endmodule
```

Example 10–18. Signal Attributes for Controlling DSP Block Inference in VHDL Code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity onereg is port (
    r  : out std_logic_vector(15 downto 0);
    en : in std_logic;
    a  : in std_logic_vector(7 downto 0);
    b  : in std_logic_vector(7 downto 0);
    c  : in std_logic_vector(15 downto 0)
    );
end onereg;

architecture beh of onereg is
signal temp : std_logic_vector(15 downto 0);
attribute syn_multstyle : string;
attribute syn_multstyle of temp : signal is "logic";

begin
    temp <= a * b;
    r <= temp when en='1' else c;
end beh;
```

Inferring RAM

When a RAM block is inferred from an HDL design, the software uses an Altera megafunction to target the device memory architecture. For certain devices, the software maps directly to memory block device primitives instead of instantiating a megafunction in the **.vqm** file.

Follow these guidelines for the Synplify software to successfully infer RAM in a design:

- The address line must be at least two bits wide.
- Resets on the memory are not supported. Refer to the device family documentation for information about whether read and write ports must be synchronous.
- Some Verilog HDL statements with blocking assignments might not be mapped to RAM blocks, so avoid blocking statements when modeling RAMs in Verilog HDL.

For certain device families, the `syn_ramstyle` attribute specifies the implementation to use for an inferred RAM. You can apply `syn_ramstyle` globally, to a module, or to a RAM instance, to specify `registers` or `block_ram` values. To turn off RAM inference, set the attribute value to `registers`.

When inferring RAM for certain Altera device families, the Synplify software generates additional bypass logic. This logic is generated to resolve a half-cycle read/write behavior difference between the RTL and post-synthesis simulations. The RTL simulation shows the memory being updated on the positive edge of the clock; the post-synthesis simulation shows the memory being updated on the negative edge of the clock. To eliminate bypass logic, the output of the RAM must be registered. By adding this register, the output of the RAM is seen after a full clock cycle, by which time the update has occurred, thus eliminating the need for bypass logic.

For devices with TriMatrix memory blocks, disable the creation of glue logic by setting the `syn_ramstyle` value to `no_rw_check`. Use `syn_ramstyle` with a value of `no_rw_check` to disable the creation of glue logic in dual-port mode.

Example 10–19 shows sample VHDL code for inferring dual-port RAM.

Example 10–19. VHDL Code for Inferred Dual-Port RAM

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY dualport_ram IS
PORT ( data_out: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    data_in: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    wr_addr, rd_addr: IN STD_LOGIC_VECTOR (6 DOWNTO 0);
    we: IN STD_LOGIC;
    clk: IN STD_LOGIC);
END dualport_ram;

ARCHITECTURE ram_infer OF dualport_ram IS
TYPE Mem_Type IS ARRAY (127 DOWNTO 0) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL mem: Mem_Type;
SIGNAL addr_reg: STD_LOGIC_VECTOR (6 DOWNTO 0);

BEGIN
    data_out <= mem (CONV_INTEGER(rd_addr));
    PROCESS (clk, we, data_in) BEGIN
        IF (clk='1' AND clk'EVENT) THEN
            IF (we='1') THEN
                mem(CONV_INTEGER(wr_addr)) <= data_in;
            END IF;
        END IF;
    END PROCESS;
END ram_infer;
```

Example 10–20 shows an example of the VHDL code preventing bypass logic for inferring dual-port RAM. The extra latency behavior stems from the inferring methodology and is not required when instantiating a megafunction.

Example 10–20. VHDL Code for Inferred Dual-Port RAM Preventing Bypass Logic

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY dualport_ram IS
PORT ( data_out: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    data_in : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    wr_addr, rd_addr : IN STD_LOGIC_VECTOR (6 DOWNTO 0);
    we : IN STD_LOGIC;
    clk : IN STD_LOGIC);
END dualport_ram;

ARCHITECTURE ram_infer OF dualport_ram IS
TYPE Mem_Type IS ARRAY (127 DOWNTO 0) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL mem : Mem_Type;
SIGNAL addr_reg : STD_LOGIC_VECTOR (6 DOWNTO 0);
SIGNAL tmp_out : STD_LOGIC_VECTOR(7 DOWNTO 0); --output register

BEGIN
    tmp_out <= mem (CONV_INTEGER(rd_addr));
    PROCESS (clk, we, data_in) BEGIN
        IF (clk='1' AND clk'EVENT) THEN
            IF (we='1') THEN
                mem(CONV_INTEGER(wr_addr)) <= data_in;
            END IF;
            data_out <= tmp_out; --registers output preventing
                                 -- bypass logic generation.
        END IF;
    END PROCESS;
END ram_infer;
```

RAM Initialization

Use Verilog HDL system tasks `$readmemb` or `$readmemh` in your HDL code to initialize RAM memories. The Synplify compiler forward-annotates the initialization values in the **.srs** (technology-independent RTL netlist) file and the mapper generates a corresponding hexadecimal memory initialization (**.hex**) file. One **.hex** file is created for each of the `altsyncram` megafunctions that are inferred in the design. The **.hex** file is associated with the `altsyncram` instance in the **.vqm** file using the `init_file` attribute.

Example 10–21 and Example 10–22 illustrate how RAM memories can be initialized through HDL code and how the corresponding **.hex** file is generated using Verilog HDL.

Example 10–21. Using $readmemb System Task to Initialize an Inferred RAM in Verilog HDL Code

```
initial
begin
    $readmemb("mem.ini", mem);
end

always @(posedge clk)
begin
    raddr_reg <= raddr;
    if(we)
        mem[waddr] <= data;
end
```

Example 10–22. Sample of .vqm Instance Containing Memory Initialization File from Example 10–21

```
altsyncram mem_hex( .wren_a(we), .wren_b(GND),...);

defparam mem_hex.lpm_type = "altsyncram";
defparam mem_hex.operation_mode = "Dual_Port";
...
defparam mem_hex.init_file = "mem_hex.hex";
```

Inferring ROM

When a ROM block is inferred from an HDL design, the software uses an Altera megafunction to target the device memory architecture. For certain devices, the software maps directly to memory block device atoms instead of instantiating a megafunction in the **.vqm** file. Follow these guidelines for the Synplify software to successfully infer ROM in a design:

- The address line must be at least two bits wide.
- The ROM must be at least half full.
- A CASE or IF statement must make 16 or more assignments using constant values of the same width.

Inferring Shift Registers

The software infers shift registers for sequential shift components so that they can be placed in dedicated memory blocks in supported device architectures using the ALTSHIFT_TAPS megafunction.

If required, set the implementation style with the `syn_srlstyle` attribute. If you do not want the components automatically mapped to shift registers, set the value to `registers`. You can set the value globally or on individual modules or registers.

For some designs, turning off shift register inference improves the design performance.

Incremental Compilation and Block-Based Design

As designs become more complex and designers work in teams, a block-based incremental design flow is often an effective design approach. In an incremental compilation flow, you can make changes to part of the design while maintaining the placement and performance of unchanged parts of the design. Design iterations are made dramatically faster by focusing new compilations on particular design partitions and merging results with previous compilation results of other partitions. In a bottom-up or team-based approach, you can perform optimization on individual subblocks and then preserve the results before you integrate the blocks into a final design and optimize it at the top level.

MultiPoint synthesis, which is available for certain device technologies in the Synplify Pro and Premier software, provides an automated block-based incremental synthesis flow. The MultiPoint feature manages a design hierarchy to let you design incrementally and synthesize designs that take too long for top-down synthesis of the entire project. MultiPoint synthesis allows different netlist files to be created for different sections of a design hierarchy and supports the Quartus II incremental compilation methodology. It also ensures that only those sections of a design that have been updated are resynthesized when the design is compiled, reducing synthesis run time and preserving the results for the unchanged blocks. You can change and resynthesize one section of a design without affecting other sections of the design.

You can also partition your design and create different netlist files manually with the Synplify software by creating a separate project for the logic in each partition of the design. Creating different netlist files for each partition of the design also means that each partition can be independent of the others.

Hierarchical design methodologies can improve the efficiency of your design process, providing better design reuse opportunities and fewer integration problems when working in a team environment. When you use these incremental synthesis methodologies, you can take advantage of incremental compilation in the Quartus II software. You can perform placement and routing on only the changed partitions of the design, reducing place-and-route time and preserving your fitting results. Follow the guidelines in this section to help you achieve good results with these methodologies.

The following list shows the general top-down compilation flow when using these features of the Quartus II software:

1. Create Verilog HDL or VHDL design files as in the regular design flow.
2. Determine which hierarchical blocks are to be treated as separate partitions in your design.
3. Set up your design using the MultiPoint feature or separate projects so that a separate netlist file is created for each partition of the design.
4. If using separate projects, disable I/O pad insertion in the implementations for lower-level partitions.
5. Compile and map each partition in the Synplify software, making constraints as you would in the regular design flow.
6. Import the **.vqm** netlist and **.tcl** file for each partition into the Quartus II software and set up the Quartus II project(s) to use incremental compilation.

7. Compile your design in the Quartus II software and preserve the compilation results using the post-fit netlist in incremental compilation.
8. When you make design or synthesis optimization changes to part of your design, resynthesize only the changed partition to generate a new netlist and **.tcl** file. Do not regenerate netlist files for the unchanged partitions.
9. Import the new netlist and **.tcl** file into the Quartus II software and recompile the design in the Quartus II software using incremental compilation.

For more information about creating partitions and using the incremental compilation in the Quartus II software, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Creating a Design with Separate Netlist Files for Incremental Compilation

The first stage of a hierarchical or incremental design flow is to ensure that different parts of your design do not affect each other. Ensure that you have separate netlists for each partition in your design so you can take advantage of incremental compilation in the Quartus II software. If the entire design is in one netlist file, changes in one partition might affect other partitions because of possible node name changes when you resynthesize the design.

To ensure the proper functioning of the synthesis flow, create separate netlist files only for modules and entities. In addition, each module or entity requires its own design file. If two different modules are in the same design file but are defined as being part of different partitions, you cannot maintain incremental compilation since both partitions would have to be recompiled when you change one of the modules.

Altera recommends that you register all inputs and outputs of each partition. This makes logic synchronous and avoids any delay penalty on signals that cross partition boundaries.

If you use boundary tri-states in a lower-level block, the Synplify software pushes (or "bubbles") the tri-states through the hierarchy to the top level to make use of the tri-state drivers on output pins of Altera devices. Because bubbling tri-states requires optimizing through hierarchies, lower-level tri-states are not supported with a block-based compilation methodology. Use tri-state drivers only at the external output pins of the device and in the top-level block in the hierarchy.

For more detailed recommendations about designing your hierarchy and creating partitions, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.

You can generate multiple **.vqm** netlist files with the MultiPoint synthesis flow in the Synplify Pro and Premier software, or by manually creating separate Synplify projects and creating a black box for each block that you want to be considered as a separate design partition.

In the MultiPoint synthesis flow (Synplify Pro and Premier only), you create multiple **.vqm** netlist files from one easy-to-manage, top-level synthesis project. By using the manual black box method, you have multiple synthesis projects, which might be required for certain team-based or bottom-up designs where a single top-level project is not desired.

After you have created multiple **.vqm** files using one of these two methods, you must create the appropriate Quartus II projects to place-and-route the design.

Using MultiPoint Synthesis with Incremental Compilation

This section describes how to generate multiple **.vqm** files using the Synplify Pro and Premier MultiPoint synthesis flow. You must first set up your constraint file and Synplify options, then apply the appropriate Compile Point settings to write multiple **.vqm** files and create design partition assignments for incremental compilation.

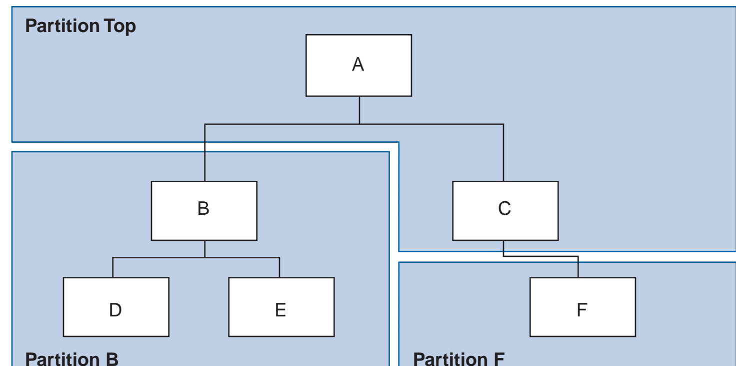
Set Compile Points and Create Constraint Files

The MultiPoint flow lets you segment a design into smaller synthesis units, called "Compile Points." The synthesis software treats each Compile Point as a partition for incremental mapping, which allows you to isolate and work on individual Compile Point modules as independent segments of the larger design without impacting other design modules. A design can have any number of Compile Points, and Compile Points can be nested. The top-level module is always treated as a Compile Point.

Compile Points are optimized in isolation from their parent, which can be another Compile Point or a top-level design. Each block created with a Compile Point is unaffected by critical paths or constraints on its parent or other blocks. A Compile Point is independent, with its own individual constraints. During synthesis, any Compile Points that have not yet been synthesized are synthesized before the top level. Nested Compile Points are synthesized before the parent Compile Points in which they are contained. When you apply the appropriate setting for the Compile Point, a separate netlist is created for that Compile Point, isolating that logic from any other logic in the design.

Figure 10–3 shows an example of a design hierarchy that is split into multiple partitions. The top-level block of each partition can be synthesized as a separate Compile Point.

Figure 10–3. Partitions in a Hierarchical Design



In this case, modules A, B, and F are Compile Points. The top-level Compile Point consists of the top-level block in the design (that is, block A in this example), including the logic that is not defined under another Compile Point. In this example, the design for top-level Compile Point A also includes the logic in one of its subblocks, C. Because block F is defined as its own Compile Point, it is not treated as part of the top-level Compile Point A. Another separate Compile Point B contains the logic in blocks B, D, and E. One netlist is created for the top-level module A and submodule C, another netlist is created for B and its submodules D and E, while a third netlist is created for F.

Apply Compile Points to the module or architecture in the Synplify Pro SCOPE spreadsheet or in the **.sdc** file. You cannot set a Compile Point in the Verilog HDL or VHDL source code. You can set the constraints manually using Tcl or by editing the **.sdc** file, or you can use one of two methods in the GUI, as described in the following subsections.

Defining Compile Points Using .tcl or .sdc Files

To set Compile Points using a **.tcl** or **.sdc** file, use the `define_compile_point` command, as shown in Example 10–23.

Example 10–23. The define_compile_point Command

```
define_compile_point [-disable] {<objname>} -type {locked, partition}
```

In Example 10–23, *objname* represents any module in the design. The Compile Point type `{locked, partition}` indicates that the Compile Point represents a partition for the Quartus II incremental compilation flow.

Each Compile Point has a set of constraint files that begin with the `define_current_design` command to set up the SCOPE environment, as follows:

```
define_current_design {<my_module>}
```

Defining Compile Points in the Top-Level SCOPE Window

The following method requires you to separately create constraint files for the top-level and lower-level Compile Points:

1. In the top-level SCOPE window, select the **Compile Points** tab.
2. Select the modules that you want to define as Compile Points and set **Type** to **locked, partition**.
3. Manually create a constraint file for each module to set constraints for each Compile Point.

Defining Compile Points by Creating a New SCOPE File

When you use the following process, the lower-level constraint file is created automatically:

1. On the File menu, click **New** and choose to create a new **Constraint File**. Or, click the **SCOPE** icon in the tool bar.
2. From the **Select File Type** tab of the **Create a New SCOPE File** dialog box, select **Compile Point**.

3. Select the module you want to designate as a Compile Point and click **OK**. The software automatically sets the Compile Points in the top-level constraint file and creates a lower-level constraint file for each Compile Point.

Additional Considerations for Compile Points

To ensure that changes to a Compile Point do not affect the top-level parent module, disable the **Update Compile Point Timing Data** option in the **Implementation Options** dialog box. If this option is enabled, updates to a child module can impact the top-level module.

You can apply the `syn_allowed_resources` attribute to any Compile Point view to restrict the number of resources for a particular module.

When using Compile Points with incremental compilation, keep the following restrictions in mind:

- To use Compile Points effectively, you must provide timing constraints (timing budgeting) for each Compile Point; the more accurate the constraints, the better your results are. Constraints are not automatically budgeted, so manual time budgeting is essential. Altera recommends that you register all inputs and outputs of each partition. This avoids any logic delay penalty on signals that cross partition boundaries.
- When using the Synplify attribute `syn_useioff` to pack registers in the I/O Elements (IOEs) of Altera devices, these registers must be in the top-level module, not a lower level. Otherwise, you must allow the Quartus II software to perform I/O register packing instead of the `syn_useioff` attribute. You can use the **Fast Input Register** or **Fast Output Register** options, or set I/O timing constraints and turn on **Optimize I/O cell register placement for timing** on the Fitter Settings page of the **Settings** dialog box in the Quartus II software.
- There is no incremental synthesis support for top-level logic; any logic in the top-level is resynthesized during every compilation in the Synplify software.

For more information about using Compile Points and setting Synplify attributes and constraints for both top-level and lower-level Compile Points, refer to the *Synopsys FPGA Synthesis User Guide* and the *Synopsys FPGA Synthesis Reference Manual* in the Synplify software.

Creating a Quartus II Project for Compile Points and Multiple .vqm Files

During compilation, the Synplify Pro and Premier software creates a <*top-level project*>**.tcl** file that provides the Quartus II software with the appropriate constraints and design partition assignments, creating a partition for each **.vqm** file along with the information to set up a Quartus II project. For details about using the Tcl script generated by the Synplify software to set up your Quartus II project and pass your constraints, refer to “Running the Quartus II Software Manually Using the Synplify-Generated Tcl Script” on page 10–16.

Depending on your design methodology, you can create one Quartus II project for all netlists (a top-down placement and routing flow) or a separate Quartus II project for each netlist (a bottom-up placement and routing flow). In a top-down incremental compilation design flow, you create design partition assignments and optionally LogicLock™ floorplan location assignments for each partition in the design within a single Quartus II project. This methodology allows for the best quality of results and performance preservation during incremental changes to your design.

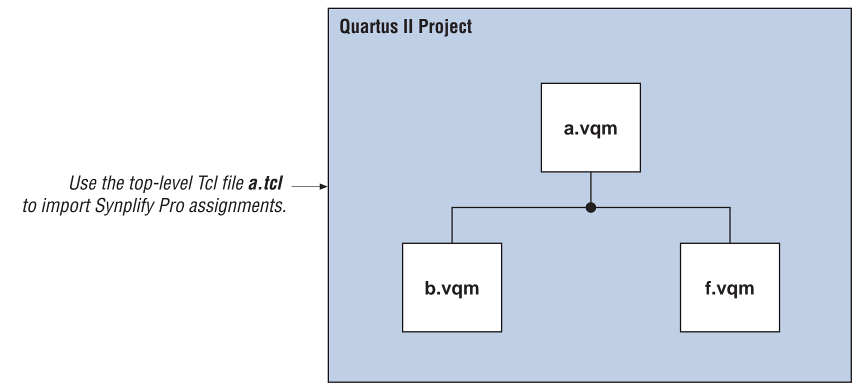
You might require a bottom-up design flow if each partition must be optimized separately, such as in certain team-based design flows. If you use this flow, Altera recommends you create a design floorplan to avoid placement conflicts between each partition. To perform a bottom-up compilation in the Quartus II software, create separate Quartus II projects and import each design partition into a top-level design using the incremental compilation export and import features to maintain placement results.

The following sections describe how to create the Quartus II projects for these two design flows.

Creating a Single Quartus II Project for a Top-Down Incremental Compilation Flow

Use the *<top-level project>***.tcl** file that contains the Synplify assignments for all partitions within the project. This method allows you to import all the partitions into one Quartus II project and optimize all modules within the project at once, taking advantage of the performance preservation and compilation-time reduction incremental compilation offers. Figure 10–4 shows a visual representation of the design flow for the example design in Figure 10–3 on page 10–39.

Figure 10–4. Design Flow Using Multiple .vqm Files with One Quartus II Project



Creating Multiple Quartus II Projects for a Bottom-Up Incremental Compilation Flow

Use the *<lower-level compile point>***.tcl** files that contain the Synplify assignments for each Compile Point. Generate multiple Quartus II projects, one for each partition and netlist in the design. The designers in the project can optimize their own partitions separately within the Quartus II software and export the results for their own partitions. Figure 10–5 shows a visual representation of the design flow for the example design in Figure 10–3 on page 10–39. You can export the optimized sub-designs and then import them into one top-level Quartus II project using incremental compilation to complete the design.

Figure 10–5. Design Flow Using Multiple .vqm Files with Multiple Quartus II Projects

Quartus II Project
a.vqm
*Use the top-level Tcl file **a.tcl** to Import Synplify Pro Assignments*

Quartus II Project
b.vqm
*Use the lower-level Tcl file **b.tcl** to Import Synplify Pro Assignments*

Quartus II Project
f.vqm
*Use the lower-level Tcl file **f.tcl** to Import Synplify Pro Assignments*

Creating Multiple .vqm Files for Incremental Compilation Using Separate Synplify Projects

This section describes how to manually generate multiple **.vqm** files for incremental compilation using black boxes and separate Synplify projects for each design partition. This manual flow is supported in versions of the Synplify software that do not include the MultiPoint Synthesis feature.

Manually Creating Multiple .vqm Files Using Black Boxes

To create multiple **.vqm** files manually in the Synplify software, create a separate project for each low-level module and top-level design that you want to maintain as a separate **.vqm** file for an incremental compilation partition. Implement black box instantiations of lower-level partitions in your top-level project.

When synthesizing the projects for the lower-level modules, perform the following steps:

1. In the **Implementation Options** dialog box, turn on **Disable I/O Insertion** for the target technology.
2. Read the HDL files for the modules.

Modules might include black box instantiations of lower-level modules that are also maintained as separate **.vqm** files.

3. Add constraints with the SCOPE constraint window.
4. Enter the clock frequency to ensure that the sub-design is correctly optimized.
5. In the **Attributes** tab, set **syn_netlist_hierarchy** to **0**.

When synthesizing the top-level design project, perform the following steps:

1. Turn off **Disable I/O Insertion** for the target technology.
2. Read the HDL files for top-level designs.
3. Create black boxes using lower-level modules in the top-level design.

4. Add constraints with the SCOPE constraint window.
5. Enter the clock frequency to ensure that the design is correctly optimized.
6. In the **Attributes** tab, set **syn_netlist_hierarchy** to **0**.

The following sections describe an example of black box implementation to create separate **.vqm** files. Figure 10–3 on page 10–39 shows an example of a design hierarchy that is split into multiple partitions.

The partition top contains the top-level block in the design (block A) and the logic that is not defined as part of another partition. In this example, the partition for top-level block A also includes the logic in one of its sub-blocks, C. Because block F is contained in its own partition, it is not treated as part of the top-level partition A. Another separate partition, B, contains the logic in blocks B, D, and E. In a team-based design, different engineers can work on the logic in different partitions. One netlist is created for the top-level module A and its submodule C, another netlist is created for B and its submodules D and E, while a third netlist is created for F.

To create multiple **.vqm** files for this design, follow these steps:

1. Generate a **.vqm** file for module B. Use **B.v/.vhd**, **D.v/.vhd**, and **E.v/.vhd** as the source files.
2. Generate a **.vqm** file for module F. Use **F.v/.vhd** as the source files.
3. Generate a top-level **.vqm** file for module A. Use **A.v/.vhd** and **C.v/.vhd** as the source files. Ensure that you use black box modules B and F, which were optimized separately in the previous steps.

Creating Black Boxes in Verilog HDL

Any design block that is not defined in the project or included in the list of files to be read for a project are treated as a black box by the software. Use the `syn_black_box` attribute to indicate that you intend to create a black box for the given module. In Verilog HDL, you must provide an empty module declaration for the module that is treated as a black box.

Example 10–24 shows an example of the **A.v** top-level file. Follow the same procedure for lower-level files that also contain a black box for any module beneath the current level hierarchy.

Example 10–24. Verilog HDL Black Box for Top-Level File A.v

```
module A (data_in, clk, e, ld, data_out);
    input data_in, clk, e, ld;
    output [15:0] data_out;

    wire [15:0] cnt_out;

    B U1 (.data_in (data_in),.clk(clk), .ld (ld),.data_out(cnt_out));
    F U2 (.d(cnt_out), .clk(clk), .e(e), .q(data_out));

    // Any other code in A.v goes here.
endmodule

// Empty Module Declarations of Sub-Blocks B and F follow here.
// These module declarations (including ports) are required for black
// boxes.

module B (data_in, clk, ld, data_out) /* synthesis syn_black_box */ ;
    input data_in, clk, ld;
    output [15:0] data_out;
endmodule

module F (d, clk, e, q) /* synthesis syn_black_box */ ;
    input [15:0] d;
    input clk, e;
    output [15:0] q;
endmodule
```

Creating Black Boxes in VHDL

Any design block that is not defined in the project or included in the list of files to be read for a project is treated as a black box by the software. Use the `syn_black_box` attribute to indicate that you intend to treat the given component as a black box. In VHDL, you must have a component declaration for the black box just like any other block in the design.

Although VHDL is not case-sensitive, a **.vqm** (a subset of Verilog HDL) file is case-sensitive. Entity names and their port declarations are forwarded to the **.vqm** file. Black box names and port declarations are also passed to the **.vqm** file. To prevent case-based mismatches, use the same capitalization for black box and entity declarations in VHDL designs.

Example 10–25 shows an example of the **A.vhd** top-level file. Follow this same procedure for any lower-level files that contain a black box for any block beneath the current level of hierarchy.

Example 10–25. VHDL Black Box for Top-Level File A.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY synplify;
USE synplify.attributes.all;

ENTITY A IS
PORT ( data_in : IN INTEGER RANGE 0 TO 15;
    clk, e, ld : IN STD_LOGIC;
    data_out : OUT INTEGER RANGE 0 TO 15 );
END A;

ARCHITECTURE a_arch OF A IS

COMPONENT B PORT(
    data_in : IN INTEGER RANGE 0 TO 15;
    clk, ld : IN STD_LOGIC;
    d_out : OUT INTEGER RANGE 0 TO 15 );
END COMPONENT;

COMPONENT F PORT(
    d : IN INTEGER RANGE 0 TO 15;
    clk, e: IN STD_LOGIC;
    q : OUT INTEGER RANGE 0 TO 15 );
END COMPONENT;

attribute syn_black_box of B: component is true;
attribute syn_black_box of F: component is true;

-- Other component declarations in A.vhd go here
signal cnt_out : INTEGER RANGE 0 TO 15;

BEGIN

U1 : B
PORT MAP (
    data_in => data_in,
    clk => clk,
    ld => ld,
    d_out => cnt_out );

U2 : F
PORT MAP (
    d => cnt_out,
    clk => clk,
    e => e,
    q => data_out );

-- Any other code in A.vhd goes here

END a_arch;
```

After you have completed the steps described in this section, you have a netlist file for each partition of the design. These files are ready for use with incremental compilation in the Quartus II software.

Creating a Quartus II Project for Multiple .vqm Files

The Synplify software creates a **.tcl** file for each **.vqm** file that provides the Quartus II software with the appropriate constraints and information to set up a project. For details about using the Tcl script generated by the Synplify software to set up your Quartus II project and pass your constraints, refer to "Running the Quartus II Software Manually Using the Synplify-Generated Tcl Script" on page 10–16.

Depending on your design methodology, you can create one Quartus II project for all netlists (a top-down placement and routing flow) or a separate Quartus II project for each netlist (a bottom-up placement and routing flow). In a top-down incremental compilation design flow, you create design partition assignments and optional LogicLock floorplan location assignments for each partition in the design within a single Quartus II project. This methodology allows for the best quality of results and performance preservation during incremental changes to your design. You might require a bottom-up design flow where each partition must be optimized separately, such as in certain team-based design flows.

To perform a bottom-up compilation in the Quartus II software, create separate Quartus II projects and import each design partition into a top-level design using the incremental compilation export and import features to maintain the results.

The following sections describe how to create the Quartus II projects for these two design flows.

Creating a Single Quartus II Project for a Top-Down Incremental Compilation Flow

Use the <*top-level project*>**.tcl** file that contains the Synplify assignments for the top-level design. This method allows you to import all of the partitions into one Quartus II project and optimize all modules within the project at once, taking advantage of the performance preservation and compilation time reduction offered by incremental compilation. Figure 10–6 shows a visual representation of the design flow for the example design in Figure 10–3 on page 10–39.

All of the constraints from the top-level project are passed to the Quartus II software in the top-level **.tcl** file, but any constraints made in the lower-level projects within the Synplify software is not forward-annotated. Enter these constraints manually in your Quartus II project.

Figure 10–6. Design Flow Using Multiple .vqm Files with One Quartus II Project

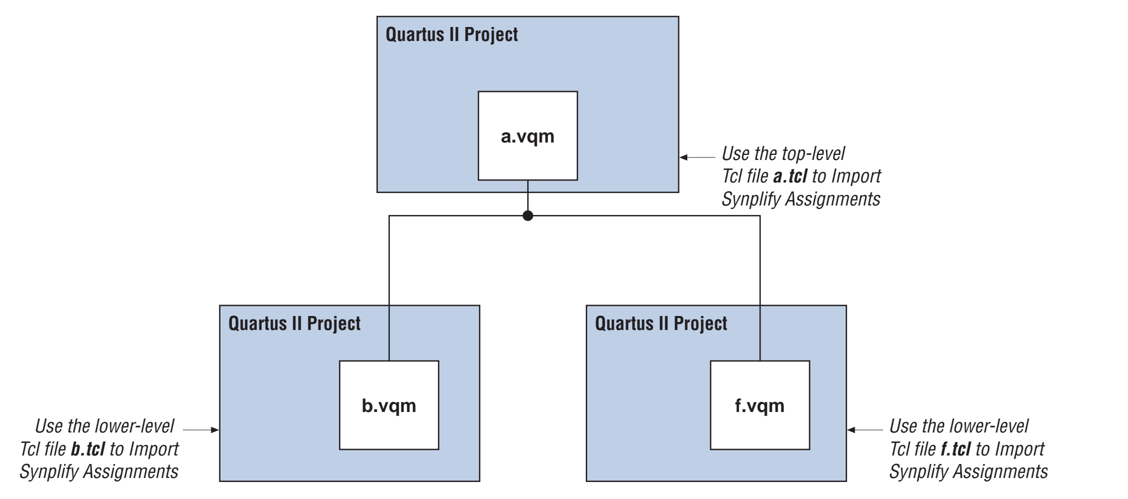
Quartus II Project
a.vqm
b.vqm
f.vqm
*Use **a.tcl** to import top-level Synplify Pro assignments. Enter any lower-level assignments manually.*

Creating Multiple Quartus II Projects for a Bottom-Up Incremental Compilation Flow

Use the **.tcl** file that is created for each **.vqm** file by the Synplify software for each Synplify project. This method generates multiple Quartus II projects, one for each block in the design. The designers in the project can optimize their own blocks separately within the Quartus II software and export the placement of their own blocks. Figure 10–7 shows a visual representation of the design flow for the example in Figure 10–3 on page 10–39.

Designers should create a LogicLock region to create a design floorplan for each block to avoid conflicts between partitions. The top-level designer then imports all the blocks and assignments into the top-level project. This method allows each block in the design to be optimized separately and then imported into one top-level project.

Figure 10–7. Design Flow Using Multiple Synplify Projects and Multiple Quartus II Projects



Performing Incremental Compilation in the Quartus II Software

In a top-down design flow using Multipoint Synthesis, the Synplify software uses the Quartus II top-level **.tcl** file to ensure that the two tools databases stay synchronized. The Tcl creates, changes, or deletes partition assignments in the Quartus II software for Compile Points that you create, change, or delete in Synplify. However, if you create, change, or delete a partition in the Quartus II software, the Synplify software does not change your Compile Point settings. Make any corresponding change in your Synplify project so that you create the correct **.vqm** files.

If you use the NativeLink integration feature described in "Using the Quartus II Software to Run the Synplify Software" on page 10–15, the Synplify software does not use any information about design partition assignments that you have set in the Quartus II software.

If you are creating netlist files using multiple Synplify projects, or if you don't use the Synplify Pro or Premier-generated **.tcl** files to update constraints in your Quartus II project, you must ensure that your Synplify **.vqm** netlists align with your Quartus II partition settings.

After you have set up your Quartus II project with **.vqm** netlist files as separate design partitions, set the appropriate Quartus II options to preserve your compilation results. On the Assignments menu, click **Design Partitions Window**. Change the Netlist Type to **Post-Fit** to preserve the previous compilation's post-fit placement results. To preserve routing results as well, set the **Fitter Preservation Level** to **Placement and Routing**. If you do not make these settings, the Quartus II software does not reuse the placement or routing results from the previous compilation.

You can take advantage of incremental compilation with your Synplify design to reduce compilation time in the Quartus II software and preserve the results for unchanged design blocks.

For more information about using Quartus II incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Conclusion

Advanced synthesis is an important part of the design flow. Taking advantage of the Synopsys Synplify and Altera Quartus II design flows allow you to control how your design files are prepared for the Quartus II place-and-route process, as well as improve performance and optimize a design for use with Altera devices. The methodologies outlined in this chapter can help optimize a design to achieve performance goals and save design time.

Referenced Documents

This chapter references the following documents:

- *Altera Constraints, Attributes, and Options* chapter in the *Synopsys FPGA Synthesis Reference Manual*
- *Altera I/O Standards* in the *Synopsys FPGA Synthesis Reference Manual*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*
- *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*
- *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Section III. Formal Verification* in volume 3 of the *Quartus II Handbook*

Document Revision History

Table 10–5. Document Revision History

Date and Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Added new section "Exporting Designs to the Quartus II Software Using NativeLink Integration" on page 10–14 ■ Minor updates for the Quartus II software version 9.0 release ■ Chapter 10 was previously Chapter 9 in software version 8.1	Updated for the Quartus II software version 9.0 release
November 2008 v8.1.0	■ Changed to 8-1/2 x 11 page size ■ Changed the chapter title from "Synplicity Synplify & Synplify Pro Support" to "Synopsys Synplify Support" ■ Replaced references to Synplicity with references to Synopsys ■ Added information about Synplify Premier ■ Updated supported device list ■ Added SystemVerilog information to Figure 10–1	Updated for the Quartus II software version 8.1 release and the Synplify software version 9.6.2 release.
May 2008 v8.0.0	■ Updated supported device list ■ Updated constraint annotation information for the TimeQuest Timing Analyzer ■ Updated RAM and MAC constraint limitations ■ Revised Table 9–1 ■ Added new section "Changing Synplify's Default Behavior for Instantiated Altera Megafunctions" ■ Added new section "Instantiating Intellectual Property Using the MegaWizard Plug-In Manager and IP Toolbench" ■ Added new section "Including Files for Quartus II Placement and Routing Only" ■ Added new section "Additional Considerations for Compile Points" ■ Removed section "Apply the LogicLock Attributes" ■ Modified Figure 9–4, 9–43, 9–47. and 9–48 ■ Added new section "Performing Incremental Compilation in the Quartus II Software" ■ Numerous text changes and additions throughout the chapter ■ Renamed several sections ■ Updated "Referenced Documents" section	Updated for Quartus II software release, version 8.0, and the Synplify software release, version 9.4.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

11. Mentor Graphics Precision Synthesis Support

QII51011-9.0.0

Introduction

This chapter documents support for the Mentor Graphics Precision RTL Synthesis and Precision RTL Plus Synthesis software in the Quartus II software design flow, as well as key design methodologies and techniques for improving your results for Altera devices.

As programmable logic device (PLD) designs become more complex and require increased performance, advanced synthesis has become an important part of the design flow. When integrated into the Quartus® II design flow, Mentor Graphics® Precision Synthesis can be used to improve performance results for Altera® devices.

The topics discussed in this chapter include:

- "Design Flow" on page 11–2
- "Creating and Compiling a Project in the Precision Synthesis Software" on page 11–6
- "Mapping the Precision Synthesis Design" on page 11–6
- "Synthesizing the Design and Evaluating the Results" on page 11–11
- "Exporting Designs to the Quartus II Software Using NativeLink Integration" on page 11–11
- "Guidelines for Altera Megafunctions and Architecture-Specific Features" on page 11–19
- "Incremental Compilation and Block-Based Design" on page 11–28

This chapter assumes that you have installed and licensed the Precision Synthesis software and the Quartus II software. You must install and license the Precision RTL Plus Synthesis software if you want to use the incremental synthesis feature for incremental compilation and block-based design.

To obtain and license the Precision Synthesis software, refer to the Mentor Graphics website at www.mentor.com. To install and run the Precision Synthesis software and to set up your work environment, refer to the *Precision Synthesis Installation Guide* in the Precision Manuals Bookcase. To access the Manuals Bookcase, in the Precision Synthesis software, click **Help** and select **Open Manuals Bookcase**.

Device Family Support

The following list shows the Altera device families supported by the Mentor Graphics Precision Synthesis software version 2009a when used with the Quartus II software version 9.0:

- ACEX® 1K
- APEX™ II, APEX 20K, APEX 20KC, APEX 20KE
- Arria® GX
- Cyclone® series
- FLEX® series
- HardCopy® series
- MAX® series, MAX II
- Stratix® series, Stratix GX series

The Precision Synthesis software also supports the Excalibur™ ARM® legacy device that is supported in the Quartus II software only with a specific license requested at www.altera.com/mysupport).

The Precision Synthesis software also supports the FLEX 8000 and MAX 9000 legacy devices that are supported only in the Altera MAX+PLUS® II software.

For information about future device support, such as the Arria II GX device family introduced with the Quartus II software version 9.0, contact your Mentor Graphics representative or refer to the Mentor Graphics website at www.mentor.com.

Design Flow

The basic steps in a Quartus II design flow using the Precision Synthesis software include:

1. Create Verilog HDL or VHDL design files.
2. Create a project in the Precision Synthesis software that contains the HDL files for your design, select your target device, and set global constraints. Refer to "Creating and Compiling a Project in the Precision Synthesis Software" on page 11–6 for details about how to create a project in the Precision Synthesis software.
3. Compile the project in the Precision Synthesis software.
4. Add specific timing constraints, optimization attributes, and compiler directives to optimize the design during synthesis.

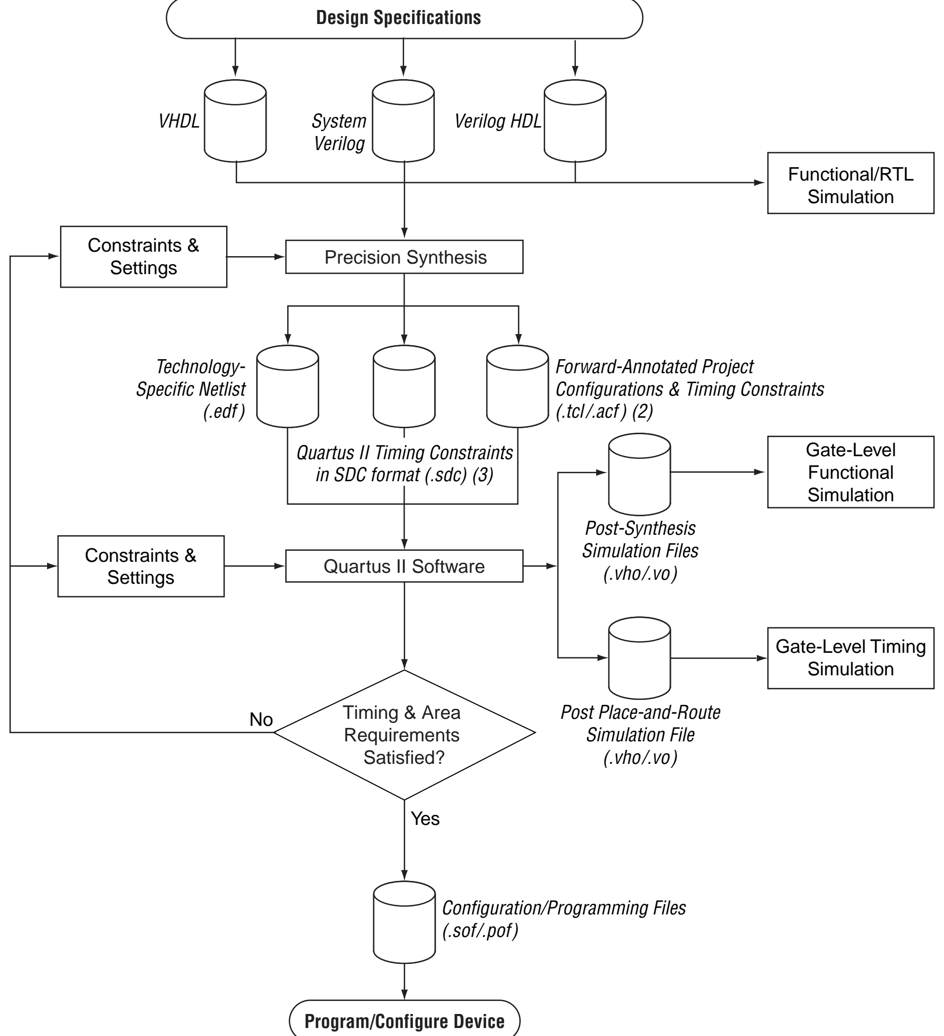
For best results, Mentor Graphics recommends specifying constraints that are as close as possible to actual operating requirements. Properly setting clock and I/O constraints, assigning clock domains, and indicating false and multicycle paths guide the synthesis algorithms more accurately toward a suitable solution in the shortest synthesis time.

5. Synthesize the project in the Precision Synthesis software. With the design analysis capabilities and cross-probing of the Precision Synthesis software, you can identify and improve circuit area and performance issues using pre-layout timing estimates.
6. Create a Quartus II project and import the following files generated by the Precision Synthesis software into the Quartus II project:
   - Technology-specific EDIF (**.edf**) netlist
   - Synopsys Design Constraints (**.sdc**) file for the TimeQuest Timing Analyzer
   - Tool command language (**.tcl**) files to set up your Quartus II project and pass constraints

   You can run the Quartus II software from within the Precision Synthesis software, or launch the Precision Synthesis software using the Quartus II software. Refer to "Running the Quartus II Software from within the Precision Synthesis Software" on page 11–12 and "Using Quartus II Software to Launch the Precision Synthesis Software" on page 11–14 for more detailed information.
7. After obtaining place-and-route results that meet your requirements, configure or program the Altera device.

Figure 11–1 shows the Quartus II design flow using the Precision Synthesis software as described in these steps. The steps are further described in detail in this chapter.

Figure 11–1. Design Flow Using the Precision Synthesis Software and Quartus II Software *(Note 1)*



Notes to Figure 11–1:

(1) Refer to Table 11–1 for details about the files generated by the Precision Synthesis software for the Quartus II design flow.

(2) Some of the constraints from the Precision Synthesis software are forward-annotated to the Quartus II software. For all devices, one Tcl file (**.tcl**) acts as a Quartus II Project Configuration file. Another **.tcl** file can be generated that contains timing constraints for the Quartus II Classic Timing Analyzer. The Assignment and Configuration File (**.acf**) stores the assignments and configuration settings for a MAX+PLUS II project.

(3) This file forward-annotates timing constraints in synopsys design constraint (**.sdc**) format for the TimeQuest Timing Analyzer.

If your area or timing requirements are not met, you can change the constraints and resynthesize the design in the Precision Synthesis software, or you can change constraints to optimize the design during place-and-route in the Quartus II software. Repeat the process until the area and timing requirements are met (Table 11–1).

You can use other options and techniques in the Quartus II software to meet area and timing requirements. One such option is the **WYSIWYG Primitive Resynthesis** option, which can perform optimizations on your EDIF netlist in the Quartus II software.

For information about netlist optimizations, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*. For more recommendations about how to optimize your design, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook.*

While simulation and analysis can be performed at various points in the design process, final timing analysis should be performed after placement and routing is complete.

During the synthesis process, the Precision Synthesis software produces several intermediate and output files. Table 11–1 lists these files and provides a brief description of each file type.

Table 11–1. Precision Synthesis Software Intermediate and Output Files

File Extension	File Description
.psp	Precision Synthesis Project File
.xdb	Mentor Graphics Design Database File
.rep *(1)*	Synthesis Area and Timing Report File
.edf	Technology-specific netlist in electronic design interchange format (EDIF)
.acf	Assignment and Configurations file for backward compatibility with the MAX+PLUS II software. An **.acf** file is created only for ACEX 1K, FLEX 10K, FLEX 10KA, FLEX 6000, FLEX 8000, MAX 7000, MAX 9000, and MAX 3000 devices.
.tcl	Forward-annotated Tcl assignments and constraints file. The <*project name*>**.tcl** file is generated for all devices. It acts as the Quartus II Project Configuration file and is used to make basic project and placement assignments, and to create and compile a Quartus II project for your EDIF netlist. If the project is set up to use the TimeQuest Timing Analyzer, this file contains the command required to use the TimeQuest Timing Analyzer instead of the Classic Timing Analyzer. The <*project name*>**_pnr_constraints.tcl** file is generated automatically for devices that use the Classic Timing Analyzer by default in the Quartus II software, and contains timing constraints for the Classic Timing Analyzer.
.sdc	Quartus II timing constraints file in Synopsys Design Constraints format This file is generated automatically if the device uses the TimeQuest Timing Analyzer by default in the Quartus II software, and has the naming convention <*project name*>**_pnr_constraints.sdc**. For more information about generating a TimeQuest constraint file, refer to "Exporting Designs to the Quartus II Software Using NativeLink Integration" on page 11–11.

Note to Table 11–1:

(1) The timing report file includes performance estimates that are based on pre-place-and-route information. Use the f_{MAX} reported by the Quartus II software after place-and-route for accurate post-place-and-route timing information. The area report file includes post-synthesis device resource utilization statistics that can differ from the resource usage after place-and-route due to black boxes or further optimizations performed during placement and routing. Use the device utilization reported by the Quartus II software after place-and-route for final resource utilization results. See "Synthesizing the Design and Evaluating the Results" on page 11–11 for details.

Creating and Compiling a Project in the Precision Synthesis Software

After creating your design files, create a project in the Precision Synthesis software that contains the basic settings for compiling the design.

Creating a Project

Set up your design files as follows:

1. In the Precision Synthesis software, click the **New Project** icon in the **Design Bar** on the left side of the GUI.
2. Set the **Project Name** and the **Project Folder**. The implementation name of the design corresponds to this project name.
3. Add input files to the project with the **Add Input Files** icon in the **Design Bar**. Precision Synthesis software automatically detects the top-level module/entity of the design. It uses the top-level module/entity to name the current implementation directory, logs, reports, and netlist files.
4. In the **Design Bar**, click the **Setup Design** icon.
5. To specify a target device family, expand the Altera entry and choose the target device and speed grade.
6. If desired, set a global design frequency and/or default input and output delays. This constrains all clock paths and all I/O pins in your design. Modify the settings for individual paths or pins that do not require such a setting.

To generate additional netlist files (for example, an HDL netlist for simulation), on the Tools menu, point to **Set Options** and **Output** and select the desired output format. The Precision Synthesis software generates a separate file for each selected type of file: EDIF, Verilog HDL, and VHDL.

Compiling the Design

To compile the design into a technology-independent implementation, in the **Design Bar**, click the **Compile** icon.

Mapping the Precision Synthesis Design

In the next steps, you set constraints and map the design to technology-specific cells. The Precision Synthesis software maps the design by default to the fastest possible implementation that meets your timing constraints. To accomplish this, you must specify timing requirements for the automatically determined clock sources. With this information, the Precision Synthesis software performs static timing analysis to determine the location of the critical timing paths. The Precision Synthesis software achieves the best results for your design when you set as many realistic constraints as possible. Be sure to set constraints for timing, mapping, false paths, multicycle paths, and other factors that control the structure of the implemented design.

Mentor Graphics recommends creating an **.sdc** file and adding this file to the **Constraint Files** section of the **Project Files** list. You can create this file with a text editor, by issuing command line constraint parameters, or using the Precision Synthesis software to generate one automatically for you on the first synthesis run. To create a constraint file with the user interface, set constraints on design objects (such

as clocks, design blocks, or pins) in the Design Hierarchy browser. By default, the Precision Synthesis software saves all timing constraints and attributes in two files: **precision_rtl.sdc** and **precision_tech.sdc**. The **precision_rtl.sdc** file contains constraints set on the RTL-level database (after compilation) and the **precision_tech.sdc** file contains constraints set on the gate-level database (after synthesis) located in the current implementation directory.

You can also enter constraints at the command line. After adding constraints at the command line, update the **.sdc** file with the **update constraint file** command. You can add constraints that change infrequently directly to the HDL source files with HDL attributes or pragmas.

The Precision SDC constraints file contains all the constraints for the Precision Synthesis project. For the Quartus II software, placement constraints are written in a **.tcl** file along with timing constraints for the Classic Timing Analyzer. The Quartus II **.sdc** file contains only timing constraints for the TimeQuest Timing Analyzer.

For details about the syntax of Synopsys Design Constraint commands, refer to the *Precision RTL Synthesis User's Manual* and the *Precision Synthesis Reference Manual*. For more details and examples of attributes, refer to the *Attributes* chapter in the *Precision Synthesis Reference Manual.* To access these manuals, in the Precision Synthesis software, click **Help** and select **Open Manuals Bookcase**.

Setting Timing Constraints

Timing constraints, based on the industry-standard **.sdc** file format, help the Precision Synthesis software to deliver optimal results. Missing timing constraints can result in incomplete timing analysis and might prevent timing errors from being detected. Precision Synthesis software provides constraint analysis prior to synthesis to ensure that designs are fully and accurately constrained. If the selected device uses the Classic Timing Analyzer by default in the Quartus II software, all timing constraints are forward-annotated to the Quartus II software using Tcl scripts for the Quartus II Classic Timing Analyzer. If the selected device uses the TimeQuest Timing Analyzer by default in the Quartus II software, <*project name*>**_pnr_constraints.sdc** is generated that contains timing constraints in SDC format.

Because the **.sdc** file format requires that timing constraints must be set relative to defined clocks, you must specify your clock constraints before applying any other timing constraints.

You also can use multicycle path and false path assignments to relax requirements or exclude nodes from timing requirements. Doing so can improve area utilization and allow the software optimizations to focus on the most critical parts of the design.

For details about the syntax of Synopsys Design Constraint commands, refer to the *Precision RTL Synthesis User's Manual* and the *Precision Synthesis Reference Manual*. To access these manuals, in the Precision Synthesis software, click **Help** and select **Open Manuals Bookcase**.

Setting Mapping Constraints

Mapping constraints affect how your design is mapped into the target Altera device. You can set mapping constraints in the user interface, in HDL code, or with the **set_attribute** command in the constraint file.

Assigning Pin Numbers and I/O Settings

The Precision Synthesis software supports assigning device pin numbers, I/O standards, drive strengths, and slew-rate settings to top-level ports of the design. You can set these timing constraints with the **set_attribute** command, the GUI, or by specifying synthesis attributes in your HDL code. These constraints are forward-annotated in the <*project name*>**.tcl** file that is read by the Quartus II software during place-and-route and do not affect synthesis.

You can use the **set_attribute** command in Precision's **.sdc** file format to specify pin number constraints, I/O standards, drive strengths, and slow slew-rate settings. Table 11–2 outlines the format to use for entries in the Precision constraint file.

Table 11–2. Constraint File Settings

Constraint	Entry Format for Precision Constraint File
Pin number	`set_attribute -name PIN_NUMBER -value "<pin number>" -port <port name>`
I/O standard	`set_attribute -name IOSTANDARD -value "<I/O Standard>" -port <port name>`
Drive strength	`set_attribute -name DRIVE -value "<drive strength in mA>" -port <port name>`
Slew rate	`set_attribute -name SLEW -value "TRUE | FALSE" -port <port name>`

You can also specify these options in the GUI. To specify a pin number or other I/O setting in the Precision Synthesis GUI, follow these steps:

1. After compiling the design, expand the **Ports** entry in the Design Hierarchy Browser.
2. Under **Ports**, expand the **Inputs** or **Outputs** entry.

You also can assign I/O settings by right-clicking the pin in the Schematic Viewer.

3. Right-click the desired pin name and select the **Set Input Constraints** option under **Inputs** or **Set Output Constraints** option under **Outputs**.
4. Enter the desired pin number on the Altera device in the **Pin Number** box (**Port Constraints** dialog box).
5. Select the I/O standard from the **IO_STANDARD** list.
6. For output pins, you can also select a drive strength setting and slew rate setting using the **DRIVE** and **SLOW SLEW** lists.

You also can use synthesis attributes or pragmas in your HDL code to make these assignments. Example 11–1 and Example 11–2 show code samples that make a pin assignment in your HDL code.

Example 11–1. Verilog HDL Pin Assignment

```
//pragma attribute clk pin_number P10;
```

Example 11–2. VHDL Pin Assignment

```
attribute pin_number : string
attribute pin_number of clk : signal is “P10”;
```

You can use the same syntax to assign the I/O standard using the attribute `IOSTANDARD`, drive strength using the attribute `DRIVE`, and slew rate using the attribute `SLEW`.

For more details about attributes and how to set these attributes in your HDL code, refer to the *Precision Synthesis Reference Manual.* To access this manual, in the Precision Synthesis software, click **Help** and select **Open Manuals Bookcase**.

Assigning I/O Registers

The Precision Synthesis software performs timing-driven I/O register mapping by default. It moves registers into an I/O element (IOE) when doing so does not negatively impact the register-to-register performance of your design, based on the timing constraints.

You can force a register to the device's IOE using the Complex I/O constraint. This option does not apply if you turn off **I/O pad insertion.** Refer to "Disabling I/O Pad Insertion" for more information.

To force an I/O register into the device's IOE using the GUI, follow these steps:

1. After compiling the design, expand the **Ports** entry in the Design Hierarchy browser.
2. Under **Ports**, expand the **Inputs** or **Outputs** entry, as desired.
3. Under **Inputs** or **Outputs**, right-click the desired pin name, point to **Map Input Register to IO** or **Map Output Register to IO** for input or output respectively, and click **True**.

You also can make the assignment by right-clicking on the pin in the Schematic Viewer.

For the Stratix and Cyclone series, and MAX II device families, the Precision Synthesis software can move an internal register to an I/O register without any restrictions on design hierarchy.

For more mature devices, the Precision Synthesis software can move an internal register to an I/O register only when the register exists in the top level of the hierarchy. If the register is buried in the hierarchy, you must flatten the hierarchy so that the buried registers are moved to the top level of the design.

Disabling I/O Pad Insertion

The Precision Synthesis software assigns I/O pad atoms (device primitives used to represent the I/O pins and I/O registers) to all ports in the top level of a design by default. In certain situations, you might not want the software to add I/O pads to all I/O pins in the design. The Quartus II software can compile a design without I/O pads; however, including I/O pads provides the Precision Synthesis software with the most information about the top-level pins in the design.

Preventing the Precision Synthesis Software from Adding I/O Pads

If you are compiling a subdesign as a separate project, I/O pins cannot be primary inputs or outputs of the device and therefore should not have an I/O pad associated with them. To prevent the Precision Synthesis software from adding I/O pads, perform the following steps:

1. On the Tools menu, click **Set Options**. The **Options** dialog box appears.
2. On the **Optimization** page, turn off **Add IO Pads**.
3. Click **Apply**.

These steps add the following command to the project file:

```
setup_design -addio=false
```

Preventing the Precision Synthesis Software from Adding an I/O Pad on an Individual Pin

To prevent I/O pad insertion on an individual pin when you are using a black box, such as DDR or a phase-locked loop (PLL), at the external ports of the design, follow these steps:

1. After compiling the design, in the Design Hierarchy browser, expand the **Ports** entry by clicking the "+" icon.
2. Under **Ports**, expand the **Inputs** or **Outputs** entry.
3. Under **Inputs** or **Outputs**, right-click the desired pin name and click **Set Input Constraints**.
4. In the **Port Constraints** dialog box for the selected pin name, turn off **Insert Pad**.

You also can make the assignment by right-clicking on the pin in the Schematic Viewer or by attaching the `nopad` attribute to the port in the HDL source code.

Controlling Fan-Out on Data Nets

Fan-out is defined as the number of nodes driven by an instance or top-level port. High fan-out nets can have significant delays that result in an unroutable net. On a critical path, high fan-out nets can cause larger delays in a single net segment that result in the timing constraints not being met. To prevent this behavior, each device family has a global fan-out value set in the Precision Synthesis software library. In addition, the Quartus II software automatically routes high fan-out signals on global routing lines in the Altera device whenever possible.

To eliminate routability and timing issues associated with high fan-out nets, the Precision Synthesis software also allows you to override the library default value on a global or individual net basis. You can override the library value by setting a `max_fanout` attribute on the net.

Synthesizing the Design and Evaluating the Results

To synthesize the design for the target device, click on the **Synthesize** icon in the **Precision Synthesis Design Bar**. During synthesis, the Precision Synthesis software optimizes the compiled design, then writes out netlists and reports to the implementation subdirectory of your working directory after the implementation is saved, using the naming convention:

<*project name*>`_impl_`<*number*>

After synthesis is complete, you can evaluate the results in terms of area and timing. The *Precision RTL Synthesis User's Manual* on the Mentor Graphics website describes different results that can be evaluated in the software.

There are several schematic viewers available in the Precision Synthesis software: RTL schematic, Technology-mapped schematic, and Critical Path schematic. These analysis tools allow you to quickly and easily isolate the source of timing or area issues, and to make additional constraint or code changes to optimize the design.

Obtaining Accurate Logic Utilization and Timing Analysis Reports

Historically, designers have relied on post-synthesis logic utilization and timing reports to determine how much logic their design requires, how big a device they require, and how fast the design runs. However, today's FPGA devices provide a wide variety of advanced features in addition to basic registers and look-up tables (LUTs). The Quartus II software has advanced algorithms to take advantage of these features, as well as optimization techniques to increase performance and reduce the amount of logic required for a given design. In addition, designs can contain black boxes and functions that take advantage of specific device features. Because of these advances, synthesis tool reports provide post-synthesis area and timing estimates, but the place-and-route software should be used to obtain final logic utilization and timing reports.

Exporting Designs to the Quartus II Software Using NativeLink Integration

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools, which allows you to run other EDA design entry/synthesis, simulation, and timing analysis tools automatically from within the Quartus II software.

After a design is synthesized in the Precision Synthesis software, the technology-mapped design is written to the current implementation directory as an EDIF netlist file, along with a Quartus II Project Configuration File and a place-and-route constraints file. You can use the Project Configuration script, <*project name*>**.tcl**, to create and compile a Quartus II project for your EDIF netlist. This script makes basic project assignments, such as assigning the target device

specified in the Precision Synthesis software. For the Quartus II Classic Timing Analyzer, the Project Configuration script calls the place-and-route constraints script, <*project name*>**_pnr_constraints.tcl**, to make your timing constraints. If you select an Arria GX, Stratix III, Cyclone III, or newer device, the constraints are written in SDC format to the <*project name*>**_pnr_constraints.sdc** file by default and is used by the Fitter and the TimeQuest Timing Analyzer in the Quartus II software.

If you want to use the Quartus II TimeQuest Timing Analyzer, use the following Precision command before compilation:

```
setup_design -timequest_sdc
```

With this command, a file named <*project name*>**_pnr_constraints.sdc** is generated after the synthesize command.

Running the Quartus II Software from within the Precision Synthesis Software

Precision Synthesis software also has a built-in place-and-route environment that allows you to run the Quartus II Fitter and view the results in the Precision Synthesis GUI. This feature is useful when performing an initial compilation of your design to view post-place-and-route timing and device utilization results, but not all the advanced Quartus II options that control the compilation process are available.

After you specify an Altera device as the target, set the options for the Quartus II software. On the Tools menu, click **Set Options**. On the **Integrated Place and Route** page (under **Quartus II Modular**), specify the path to the Quartus II executables in the **Path to Quartus II installation tree** box.

To automate the place-and-route process, click the **Run Quartus II** icon in the **Quartus II Modular** window of the Precision Synthesis toolbar. The Quartus II software uses the current implementation directory as the Quartus II project directory and runs a full compilation in the background (that is, the user interface does not appear).

Two primary Precision Synthesis software commands control the place-and-route process. Use the `setup_place_and_route` command to set the place-and-route options. The process is started with the `place_and_route` command.

Precision Synthesis software versions 2004a and later support using individual Quartus II executables, such as analysis and synthesis (`quartus_map`), Fitter (`quartus_fit`), and the Classic Timing Analyzer (`quartus_tan`) or the TimeQuest Timing Analyzer (`quartus_sta`) (only for software version 2006a and later), for improved runtime and memory utilization during place and route. This flow is referred to as the **Quartus II Modular** flow option in Precision Synthesis software and is compatible with Quartus II software versions beginning with version 4.0. By default, the Precision Synthesis software generates a Quartus II Project Configuration File (**.tcl** file) for Arria GX, Stratix series, MAX II, and Cyclone series device families. When you use this flow, all timing constraints that you set during synthesis are exported to the Quartus II place-and-route constraints file <*project name*>**_pnr_constraints.tcl**, or <*project name*>**_pnr_constraints.sdc**, depending on which Quartus II timing analyzer the Precision Synthesis software is targeting.

For other device families, the Precision Synthesis software uses the **Quartus II** flow option, which enables the Quartus II compilation flow that existed in Precision Synthesis software versions earlier than 2004a. The Quartus II Project Configuration File (**.tcl** file) is written when using the **Quartus II** flow option that includes supported timing constraints that you specified during synthesis. This **.tcl** file is compatible with all versions of the Quartus II software; however, the format and timing constraints do not take full advantage of the features in the Quartus II software introduced with version 4.0.

To force the use of a particular flow when it is not the default for a certain device family, use the following command to set up the integrated place-and-route flow:

```
setup_place_and_route -flow "<Altera Place-and-Route flow>"
```

Depending on the device family, you can use one of the following flow options in the preceding command:

- `Quartus II Modular`
- `Quartus II`
- `MAX+PLUS II`

For example, for the Stratix II or MAX II device families (which were not supported in Quartus II software versions earlier than 4.0), you can use only the **Quartus II Modular** flow. For the Stratix device family, you can use either the **Quartus II Modular** or **Quartus II** flows. The FLEX 8000 device family, which is not supported in the Quartus II software, is supported only by the **MAX+PLUS II** flow.

After the design is compiled in the Quartus II software from within the Precision Synthesis software, you can invoke the Quartus II GUI manually and then open the project using the generated Quartus II project file. You can view reports, run analysis tools, specify options, and run the various processing flows available in the Quartus II software.

Running the Quartus II Software Manually Using the Precision Synthesis-Generated Tcl Script

You can use the Quartus II software separately from the Precision Synthesis software. To run the Tcl script generated by the Precision Synthesis software to set up your project and start a full compilation, perform the following steps:

1. Ensure the **.edf**, **.tcl** files, and **.sdc** file (if using the TimeQuest Timing Analyzer) are located in the same directory (by default, the files should be located in the implementation directory).
2. In the Quartus II software, on the View menu, point to **Utility Windows** and click **Tcl Console**.
3. At the Tcl Console command prompt, type the command:

   ```
   source <path>/<project name>.tcl ↵
   ```
4. On the File menu, click **Open Project**. Browse to the project name, and click **Open**.
5. Compile the project in the Quartus II software.

Using Quartus II Software to Launch the Precision Synthesis Software

With NativeLink integration, you can set up the Quartus II software to run the Precision Synthesis software. This feature allows you to use the Precision Synthesis software to synthesize a design as part of a normal compilation.

For detailed information about using NativeLink integration with the Precision Synthesis software, go to *Specifying EDA Tool Settings* in the Quartus II Help index.

Passing Constraints to the Quartus II Software

The place-and-route constraints script forward-annotates timing constraints that you made in the Precision Synthesis software. This integration allows you to enter these constraints once in the Precision Synthesis software, and then pass them automatically to the Quartus II software.

All of the constraints you set in the Precision Synthesis software are mapped to the Quartus II software. For some constraints you set in the Precision Synthesis software, there might be a different command mapped to the Quartus II software, depending on whether you are using the TimeQuest Timing Analyzer or the Classic Timing Analyzer.

Refer to the introductory text in the section "Exporting Designs to the Quartus II Software Using NativeLink Integration" on page 11–11 for information on how to ensure the Precision Synthesis software targets the TimeQuest Timing Analyzer.

The following constraints are translated by the Precision Synthesis software and are applicable to the Classic Timing Analyzer and the TimeQuest Timing Analyzer:

- create_clock
- set_input_delay
- set_output_delay
- set_max_delay
- set_min_delay
- set_false_path
- set_multicycle_path

create_clock

You can specify a clock in the Precision Synthesis software, as shown in Example 11–3.

Example 11–3. Specifying a Clock using create_clock

```
create_clock -name <clock_name> -period <period in ns> -waveform {<edge_list>} -domain \
<ClockDomain> <pin>
```

The period is specified in units of nanoseconds (ns). If no clock domain is specified, the clock belongs to a default clock domain `main`. All clocks in the same clock domain are treated as synchronous (related) clocks. If no <*clock_name*> is provided, the default name `virtual_default` is used. The <*edge_list*> sets the rise and fall edges of the clock signal over an entire clock period. The first value in the list is a rising transition, typically the first rising transition after time zero. The waveform can contain any even number of alternating edges, and the edges listed should alternate between rising and falling. The position of any edge can be equal to or greater than zero but must be equal to or less than the clock period.

If `-waveform` <*edge_list*> is not specified, and `-period` <*period in ns*> is specified, the default waveform has a rising edge of 0.0 and a falling edge of <*period_value*>/2.

The Precision Synthesis software passes the clock definitions to the Quartus II software with the `create_base_clock` command in the place-and-route constraints file for the Classic Timing Analyzer. For the TimeQuest Timing Analyzer, the clock constraint is mapped to the TimeQuest `create_clock` setting in the Quartus II software.

The following list describes some differences in the clock properties supported by the Precision Synthesis software and the Quartus II software:

- The Quartus II software supports only clock waveforms with two edges in a clock cycle. If the Precision Synthesis software finds a multi-edge clock, it issues an error message when you synthesize your design in the Precision Synthesis software. This applies to both the Quartus II TimeQuest Timing Analyzer and the Quartus II Classic Timing Analyzer.
- Clocks in the same clock domain are annotated with the `create_relative_clock` command to create related clocks for the Quartus II Classic Timing Analyzer.
- The Quartus II Classic Timing Analyzer assumes the first clock edge to be at time zero (0.0). If the Precision Synthesis software waveform has a first transition at a time different than time zero, the Precision Synthesis software creates a base clock without any target, then uses this to create a relative clock with an offset set to the first clock edge.

set_input_delay

This port-specific input delay constraint is specified in the Precision Synthesis software, as shown in Example 11–4.

Example 11–4. Specifying set_input_delay

```
set_input_delay {<delay_value> <port_pin_list>} -clock <clock_name> -rise -fall -add_delay
```

This constraint is mapped to the `set_input_delay` setting in the Quartus II software.

When the reference clock <*clock_name*> is not specified, all clocks are assumed to be the reference clocks for this assignment. The input pin name for the assignment can be an input pin name of a time group. The software can use the option `clock_fall` to specify delay relative to the falling edge of the clock.

Although the Precision Synthesis software allows you to set input delays on pins inside the design, these constraints are not sent to the Quartus II software, and a message is displayed.

set_output_delay

This port-specific output delay constraint is specified in the Precision Synthesis software, as shown in Example 11–5.

Example 11–5. Using the set_output_delay Constraint

```
set_output_delay {<delay_value> <port_pin_list>} -clock <clock_name> -rise -fall -add_delay
```

This constraint is mapped to the `set_output_delay` setting in the Quartus II software.

When the reference clock <*clock_name*> is not specified, all clocks are assumed to be the reference clocks for this assignment. The output pin name for the assignment can be an output pin name of a time group.

Although the Precision Synthesis software allows you to set output delays on pins inside the design, these constraints are not sent to the Quartus II software.

set_max_delay

The total delay for a point-to-point timing path constraint is specified in the Precision Synthesis software, as shown in Example 11–6.

Example 11–6. Using the set_max_delay Constraint

```
set_max_delay -from {<from_node_list>} -to {<to_node_list>} <delay_value>
```

This command specifies that the maximum required delay for any start point in <*from_node_list*> to any endpoint in <*to_node_list*> must be less than <*delay_value*>. Typically, this command is used to override the default setup constraint for any path with a specific maximum time value for the path.

The node lists can contain a collection of clocks, registers, ports, pins, or cells. The `-from` and `-to` parameters specify the source (start point) and the destination (endpoint) of the timing path, respectively. The source list (<*from_node_list*>) cannot include output ports, and the destination list (<*to_node_list*>) cannot include input ports. If you include more than one node on a list, you must enclose the nodes in quotes or in '{ }' braces.

If you specify a clock in the source list, you must specify a clock in the destination list. Applying `set_max_delay` between clocks applies the exception from all registers or ports driven by the source clock to all registers or ports driven by the destination clock. Applying exceptions between clocks is more efficient than applying them for specific node to node, or node to clock paths. If you want to specify pin names in the list, the source must be a clock pin, and the destination must be any non-clock input pin to a register. Assignments from clock pins, or to and from cells, apply to all registers in the cell or for those driven by the clock pin.

set_min_delay

The minimum delay for a point-to-point timing path constraint is specified in the Precision Synthesis software, as shown in Example 11–7.

Example 11–7. Using the set_min_delay Constraint

```
set_min_delay -from {<from_node_list>} -to {<to_node_list>} <delay_value>
```

This command specifies that the minimum required delay for any start point in *<from_node_list>* to any endpoint in *<to_node_list>* must be greater than *<delay_value>*. Typically, you use this command to override the default setup constraint for any path with a specific minimum time value for the path.

The node lists can contain a collection of clocks, registers, ports, pins, or cells. The `-from` and `-to` parameters specify the source (start point) and the destination (endpoint) of the timing path, respectively. The source list (*<from_node_list>*) cannot include output ports, and the destination list (*<to_node_list>*) cannot include input ports. If you include more than one node to a list, you must enclose the nodes in quotes or in '{ }' braces.

If you specify a clock in the source list, you must specify a clock in the destination list. Applying `set_min_delay` between clocks applies the exception from all registers or ports driven by the source clock to all registers or ports driven by the destination clock. Applying exceptions between clocks is more efficient than applying them for specific node to node, or node to clock paths. If you want to specify pin names in the list, the source must be a clock pin, and the destination must be any non-clock input pin to a register. Assignments from clock pins, or to and from cells, apply to all registers in the cell or for those driven by the clock pin.

set_false_path

The false path constraint is specified in the Precision Synthesis software, as shown in Example 11–8.

Example 11–8. Using the set_false_path Constraint

```
set_false_path -to <to_node_list> -from <from_node_list> -reset_path
```

The node lists can be a list of clocks, ports, instances, and pins. Multiple elements in the list can be represented using wildcards such as "*" and "?".

In place-and-route Tcl constraints file, this setting in the Precision Synthesis software is mapped to a `set_timing_cut_assignment` setting for the Classic Timing Analyzer. For the TimeQuest Timing Analyzer, this constraint is mapped to the `set_false_path` setting.

The node lists for this assignment represents top-level ports and/or nets connected to instances (end points of timing assignments).

The Quartus II software supports `setup`, `hold`, `rise`, or `fall` options for this assignment only if you are using the TimeQuest Timing Analyzer.

The Quartus II Classic Timing Analyzer does not support false paths with the `through` path specification. Any setting in the Precision Synthesis software with a `through` specification can be mapped to a setting in the Quartus II software only if you use the TimeQuest Timing Analyzer.

For the Classic Timing Analyzer, if you use the `-from` or `-to` option without using both options, the Precision Synthesis command is converted to a Quartus II command using wildcards. Table 11–3 lists these `set_false_path` constraints in the Precision Synthesis software and the Quartus II software equivalent when the Classic Timing Analyzer is used.

Table 11–3. set_false_path Constraints with the Classic Timing Analyzer

Precision Synthesis Assignment	Quartus II Equivalent
set_false_path -from <*from_node_list*>	set_timing_cut_assignment -to {*} -from \ <*node_list*>
set_false_path -to <*to_node_list*>	set_timing_cut_assignmet -to <*node_list*> \ -from {*}

set_multicycle_path

This multi-cycle path constraint is specified in the Precision Synthesis software, as shown in Example 11–9.

Example 11–9. Using the set_multicycle_path Constraint

```
set_multicycle_path <multiplier_value> [-start] [-end] -to <to_node_list> -from <from_node_list> \
-reset_path
```

The node list can contain clocks, ports, instances, and pins. Multiple elements in the list can be represented using wildcards such as "*" and "?". Paths without multicycle path definitions are identical to paths with multipliers of 1. To add one additional cycle to the datapath, use a multiplier value of 2. The option `start` indicates that source clock cycles should be considered for the multiplier. The option `end` indicates that destination clock cycles should be considered for the multiplier. The default is to reference the end clock.

In the place-and-route Tcl constraints file, this setting in the Precision Synthesis software is mapped to a `set_multicycle_assignment` setting for the Classic Timing Analyzer. For TimeQuest Timing Analyzer, this constraint is mapped to the `set_multicycle_path` setting.

The node lists represent top-level ports and/or nets connected to instances (end points of timing assignments). The node lists can contain wildcards (such as "*"); the Quartus II software automatically expands all wildcards.

For the Classic Timing Analyzer, if you use the `-from` or `-to` option without using both options, the Precision Synthesis command is converted to a Quartus II command using wildcards. Table 11–4 lists the `set_multicycle_path` constraints in the Precision Synthesis software and the Quartus II software equivalent, when the Classic Timing Analyzer is used.

Table 11–4. set_multicycle_path Constraints for the Classic Timing Analyzer

Precision Synthesis Assignment	Quartus II Equivalent
set_multicycle_path -from \ <*from_node_list*> <*value*>	set_multicycle_assignment -to {*} \ -from <*node_list*> <*value*>
set_multicycle_path -to \ <*to_node_list*> <*value*>	set_multicycle_assignmet -to \ <*node_list*> -from {*} <*value*>

The Quartus II software supports the `rise` or `fall` options on this assignment only if you use the TimeQuest Timing Analyzer.

The Quartus II Classic Timing Analyzer does not support multicycle path with a `through` path specification. Any setting in Precision Synthesis software with a `-through` specification can be mapped to a setting in the Quartus II software only if you use the TimeQuest Timing Analyzer.

Guidelines for Altera Megafunctions and Architecture-Specific Features

Altera provides parameterizable megafunctions, including the LPMs, device-specific Altera megafunctions, IP available as Altera MegaCore functions, and IP available through the Altera Megafunction Partners Program (AMPP℠). You can use megafunctions and IP functions by instantiating them in your HDL code or you can infer certain megafunctions from generic HDL code.

If you want to instantiate a megafunction such as a PLL in your HDL code, you can do so with the MegaWizard™ Plug-In Manager to parameterize the function or instantiating the function using the port and parameter definition. The MegaWizard Plug-In Manager provides a graphical interface within the Quartus II software for customizing and parameterizing any available megafunction for the design. "Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager" and "Instantiating Intellectual Property Using the MegaWizard Plug-In Manager and IP Toolbench" on page 11–20 describe the MegaWizard Plug-In Manager flow with the Precision Synthesis software.

For more information about specific Altera megafunctions, refer to the Quartus II Help. For more information about IP functions, refer to the appropriate IP documentation.

The Precision software automatically recognizes certain types of HDL code and infers the appropriate megafunction. The Precision Synthesis software provides options to control inference of certain types of megafunctions, as described in "Inferring Altera Megafunctions from HDL Code" on page 11–22.

For a detailed discussion about instantiating versus inferring megafunctions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. This chapter also provides details on using the MegaWizard Plug-In Manager in the Quartus II software and explains the files generated by the wizard, as well as providing coding style recommendations and HDL examples for inferring megafunctions in Altera devices.

Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager

This section describes how to instantiate Altera megafunctions using the MegaWizard Plug-In Manager, and how to generate the files that are included in the Precision Synthesis project for synthesis.

You can run the stand-alone version of the MegaWizard Plug-In Manager by typing the following command at a command prompt:

```
qmegawiz ↵
```

Using MegaWizard Plug-In Manager-Generated Verilog HDL Files for Megafunction Instantiation

The MegaWizard Plug-In Manager generates a Verilog HDL instantiation template file <*output file*>**_inst.v** and a hollow-body black box module declaration <*output file*>**_bb.v** for use in your Precision Synthesis design. Incorporate the instantiation template file, <*output file*>**_inst.v**, into your top-level design to instantiate the megafunction wrapper file, <*output file*>**.v**.

Include the hollow-body black box module declaration <*output file*>**_bb.v** in your Precision Synthesis project to describe the port connections of the black box. Adding the megafunction wrapper file <*output file*>**.v** in your Precision Synthesis project is optional, but you must add it to your Quartus II project along with the Precision Synthesis-generated EDIF netlist.

Alternatively, you can include the megafunction wrapper file <*output file*>**.v** in your Precision Synthesis project and then right-click on the file in the input file list, and select **Properties**. In the **Input file properties** dialog box, turn on **Exclude file from Compile Phase** and click **OK**. When this option is on, the Precision Synthesis software excludes the file from compilation and makes a copy of the file in the appropriate directory so that the Quartus II software can use it during place and route.

Using MegaWizard Plug-In Manager-Generated VHDL Files for Megafunction Instantiation

The MegaWizard Plug-In Manager generates a VHDL component declaration file <*output file*>**.cmp** and a VHDL instantiation template file <*output file*>**_inst.vhd** for use in your Precision Synthesis design. Incorporate the component declaration and instantiation template into your top-level design to instantiate the megafunction wrapper file, <*output file*>**.vhd**.

Adding the megafunction wrapper file <*output file*>**.vhd** in your Precision Synthesis project is optional, but you must add it to your Quartus II project along with the Precision Synthesis-generated EDIF netlist.

Alternatively, you can include the megafunction wrapper file <*output file*>**.vhd** in your Precision Synthesis project and then right-click on the file in the input file list, and select **Properties**. In the **Input file properties** dialog box, turn on **Exclude file from Compile Phase** and click **OK**. When this option is on, the Precision Synthesis software excludes the file from compilation and makes a copy of the file in the appropriate directory so that the Quartus II software can use it during place and route.

Instantiating Intellectual Property Using the MegaWizard Plug-In Manager and IP Toolbench

Many Altera IP functions include a resource and timing estimation netlist that the Precision Synthesis software can use to synthesize and optimize logic around the IP efficiently. As a result, the Precision Synthesis software provides better timing correlation, area estimates, and Quality of Results (QoR) than a black-box approach.

To create this netlist file, perform the following steps:

1. Select the IP function in the MegaWizard Plug-In Manager and click **Next** to open the IP Toolbench.
2. Click **Set Up Simulation**, which sets up all the EDA options.

3. Enable the **Generate netlist** option to generate a netlist for resource and timing estimation and click **OK**.
4. Click **Generate** to generate the netlist file.

The Quartus II software generates a file <*output file*>**_syn.v**. This netlist contains the "grey box" information for resource and timing estimation, but does not contain the actual implementation. Include this netlist file into your Precision Synthesis project as an input file. Then include the megafunction wrapper file <*output file*>**.v | vhd** into the Quartus II project along with your EDIF output netlist.

The generated "grey box" netlist file, <*output file*>**_syn.v**, is always in Verilog HDL format, even if you select VHDL as the output file format.

There is currently no grey box support for SOPC Builder systems in the MegaWizard Plug-In Manager. For information about creating a grey box netlist file from the command line, search Altera's Knowledge Database. Alternatively, you can use a black box approach as described in "Using Generated Verilog HDL Files for Black Box IP Function Instantiation".

Using Generated Verilog HDL Files for Black Box IP Function Instantiation

You can use the `syn_black_box` or `black_box` compiler directive to declare a module as a black box. The top-level design files must contain the IP port mapping and a hollow-body module declaration. You can apply the directive to the module declaration in the top-level file or a separate file included in the project to instruct the Precision Synthesis software that this is a black box.

The `syn_black_box` and `black_box` directives are supported only on module or entity definition.

Example 11–10 shows a sample top-level file that instantiates **my_verilogIP.v**, which is a simplified customized variation generated by the MegaWizard Plug-In Manager and IP Toolbench.

Example 11–10. Top-Level Verilog HDL Code with Black Box Instantiation of IP

```
module top (clk, count);
   input clk;
   output[7:0] count;

   my_verilogIP verilogIP_inst (.clock (clk), .q (count));
endmodule

// Module declaration
// The following attribute is added to create a
// black box for this module.
module my_verilogIP (clock, q) /* synthesis syn_black_box */;
   input clock;
   output[7:0] q;
endmodule
```

Using Generated VHDL Files for Black Box IP Function Instantiation

You can use the `syn_black_box` or `black_box` compiler directive to declare a component as a black box. The top-level design files must contain the megafunction variation component declaration and port mapping. Apply the directive to the component declaration in the top-level file.

The `syn_black_box` and `black_box` directives are supported only on module or entity definition.

Example 11–11 shows a sample top-level file that instantiates **my_vhdlIP.vhd**, which is a simplified customized variation generated by the MegaWizard Plug-In Manager and IP Toolbench.

Example 11–11. Top-Level VHDL Code with Black Box Instantiation of IP

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY top IS
  PORT (
    clk: IN STD_LOGIC ;
    count: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
  );
END top;

ARCHITECTURE rtl OF top IS
  COMPONENT my_vhdlIP
  PORT (
    clock: IN STD_LOGIC ;
    q: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
  );
  end COMPONENT;
  attribute syn_black_box : boolean;
  attribute syn_black_box of my_vhdlIP: component is true;
  BEGIN
    vhdlIP_inst : my_vhdlIP PORT MAP (
       clock => clk,
       q => count
    );
END rtl;
```

Inferring Altera Megafunctions from HDL Code

The Precision Synthesis software automatically recognizes certain types of HDL code and maps arithmetic and relational operators, and memory (RAM and ROM), to efficient technology-specific implementations. This allows for the use of technology-specific resources to implement these structures by inferring the appropriate Altera megafunction to provide optimal results. In some cases, the Precision Synthesis software has options that you can use to disable or control inference.

For coding style recommendations and examples for inferring megafunctions in Altera devices, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*, and the *Precision Synthesis Style Guide* in the Precision Manuals Bookcase. To access these manuals, in the Precision Synthesis software, click **Help** and select **Open Manuals Bookcase**.

Multipliers

The Precision Synthesis software detects multipliers in HDL code and maps them directly to device atoms to implement the multiplier in the appropriate type of logic. The Precision Synthesis software also allows you to control the device resources that are used to implement individual multipliers, as described in the following section.

Controlling DSP Block Inference for Multipliers

By default, the Precision Synthesis software uses DSP blocks available in the Stratix series of devices to implement multipliers. The default setting is **AUTO**, to allow Precision Synthesis software the flexibility to choose between logic look-up tables (LUTs) and DSP blocks, depending on the size of the multiplier. You can use the Precision Synthesis GUI or HDL attributes to direct the mapping to only logic elements or to only DSP blocks. The options for multiplier mapping in the Precision Synthesis software are shown in Table 11–5.

Table 11–5. Options for dedicated_mult Parameter to Control Multiplier Implementation in Precision Synthesis

Value	Description
ON	Use only DSP blocks to implement multipliers, regardless of the size of the multiplier.
OFF	Use only logic (LUTs) to implement multipliers.
AUTO	Use logic (LUTs) and DSP blocks to implement multipliers depending on the size of the multipliers.

Using the GUI

To set the `Use Dedicated Multiplier` option in the Precision Synthesis GUI, perform the following steps:

1. Compile the design.
2. In the Design Hierarchy browser, right-click the operator for the desired multiplier and click **Use Dedicated Multiplier**.

Using Attributes

To control the implementation of a multiplier in your HDL code, use the `dedicated_mult` attribute with the appropriate value from Table 11–5, as shown in Example 11–12 and Example 11–13.

Example 11–12. Setting the dedicated_mult Attribute in Verilog HDL

```
//synthesis attribute <signal name> dedicated_mult <value>
```

Example 11–13. Setting the dedicated_mult Attribute in VHDL

```
ATTRIBUTE dedicated_mult: STRING;
ATTRIBUTE dedicated_mult OF <signal name>: SIGNAL IS <value>;
```

The `dedicated_mult` attribute can be applied to signals and wires; it does not work when applied to a register. This attribute can be applied only to simple multiplier code, such as `a = b * c`.

Some signals for which the `dedicated_mult` attribute is set can be synthesized away by the Precision Synthesis software because of design optimization. In such cases, if you want to force the implementation, you should preserve the signal by setting the `preserve_signal` attribute to `TRUE`, as shown in Example 11–14 and Example 11–15.

Example 11–14. Setting the preserve_signal Attribute in Verilog HDL

```
//synthesis attribute <signal name> preserve_signal TRUE
```

Example 11–15. Setting the preserve_signal Attribute in VHDL

```
ATTRIBUTE preserve_signal: BOOLEAN;
ATTRIBUTE preserve_signal OF <signal name>: SIGNAL IS TRUE;
```

Example 11–16 and Example 11–17 are examples in Verilog HDL and VHDL of using the `dedicated_mult` attribute to implement the given multiplier in regular logic in the Quartus II software.

Example 11–16. Verilog HDL Multiplier Implemented in Logic

```
module unsigned_mult (result, a, b);
   output [15:0] result;
   input [7:0] a;
   input [7:0] b;
   assign result = a * b;
   //synthesis attribute result dedicated_mult OFF
endmodule
```

Example 11–17. VHDL Multiplier Implemented in Logic

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY unsigned_mult IS
   PORT(
      a: IN std_logic_vector (7 DOWNTO 0);
      b: IN std_logic_vector (7 DOWNTO 0);
      result: OUT std_logic_vector (15 DOWNTO 0));
ATTRIBUTE dedicated_mult: STRING;
END unsigned_mult;

ARCHITECTURE rtl OF unsigned_mult IS
   SIGNAL a_int, b_int: UNSIGNED (7 downto 0);
   SIGNAL pdt_int:  UNSIGNED (15 downto 0);
ATTRIBUTE dedicated_mult OF pdt_int: SIGNAL IS "OFF";
BEGIN
   a_int <= UNSIGNED (a);
   b_int <= UNSIGNED (b);
   pdt_int <= a_int * b_int;
   result <= std_logic_vector(pdt_int);
END rtl;
```

Multiplier-Accumulators and Multiplier-Adders

The Precision Synthesis software detects multiply-accumulators or multiply-adders in HDL code and infers an ALTMULT_ACCUM or ALTMULT_ADD megafunction so that the logic can be placed in DSP blocks, or maps directly to device atoms to implement the multiplier in the appropriate type of logic.

The Precision Synthesis software supports inference for these functions only if the target device family has dedicated DSP blocks.

The Precision Synthesis software also allows you to control the device resources used to implement multiply-accumulators or multiply-adders in your project or in a particular module. Refer to "Controlling DSP Block Inference" for more information.

For more information about DSP blocks in Altera devices, refer to the appropriate Altera device family handbook and device-specific documentation. For details about which functions a given DSP block can implement, refer to the DSP Solutions Center on the Altera website at www.altera.com.

For more information about inferring Multiply-Accumulator and Multiply-Adder megafunctions in HDL code, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*, and the *Precision Synthesis Style Guide* in the Precision Synthesis Manuals Bookcase.

Controlling DSP Block Inference

By default, the Precision Synthesis software infers the ALTMULT_ADD or ALTMULT_ACCUM megafunction as appropriate for your design. These megafunctions allow the Quartus II software the flexibility to choose regular logic or DSP blocks depending on the device utilization and the size of the function.

You can use the `extract_mac` attribute to prevent the inference of an ALTMULT_ADD or ALTMULT_ACCUM megafunction in a certain module or entity. The options for this attribute are shown in Table 11–6.

Table 11–6. Options for extract_mac Attribute Controlling DSP Implementation

Value	Description
TRUE	The ALTMULT_ADD or ALTMULT_ACCUM megafunction is inferred
FALSE	The ALTMULT_ADD or ALTMULT_ACCUM megafunction is not inferred

To control inference, use the `extract_mac` attribute with the appropriate value from Table 11–6 in your HDL code, as shown in Example 11–18 and Example 11–19.

Example 11–18. Setting the extract_mac Attribute in Verilog HDL

```
//synthesis attribute <module name> extract_mac <value>
```

Example 11–19. Setting the extract_mac Attribute in VHDL

```
ATTRIBUTE extract_mac: BOOLEAN;
ATTRIBUTE extract_mac OF <entity name>: ENTITY IS <value>;
```

To control the implementation of the multiplier portion of a multiply-accumulator or multiply-adder, you must use the `dedicated_mult` attribute.

Example 11–20 and Example 11–21 use the `extract_mac`, `dedicated_mult`, and `preserve_signal` attributes (in Verilog HDL and VHDL) to implement the given DSP function in logic in the Quartus II software.

Example 11–20. Using extract_mac, dedicated_mult and preserve_signal in Verilog HDL

```
module unsig_altmult_accum1 (dataout, dataa, datab, clk, aclr, clken);
   input [7:0] dataa, datab;
   input clk, aclr, clken;
   output    [31:0] dataout;

   reg       [31:0] dataout;
   wire      [15:0] multa;
   wire      [31:0] adder_out;

   assign multa = dataa * datab;

   //synthesis attribute multa preserve_signal TRUE
   //synthesis attribute multa dedicated_mult OFF
   assign adder_out = multa + dataout;

   always @ (posedge clk or posedge aclr)
   begin
      if (aclr)
      dataout <= 0;
      else if (clken)
      dataout <= adder_out;
   end

   //synthesis attribute unsig_altmult_accum1 extract_mac FALSE
endmodule
```

Example 11–21. Using extract_mac, dedicated_mult, and preserve_signal in VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;
ENTITY signedmult_add IS
   PORT(
      a, b, c, d: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
      result: OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
   ATTRIBUTE preserve_signal: BOOLEAN;
   ATTRIBUTE dedicated_mult: STRING;
   ATTRIBUTE extract_mac: BOOLEAN;
   ATTRIBUTE extract_mac OF signedmult_add: ENTITY IS FALSE;
END signedmult_add;
ARCHITECTURE rtl OF signedmult_add IS
   SIGNAL a_int, b_int, c_int, d_int : signed (7 DOWNTO 0);
   SIGNAL pdt_int, pdt2_int : signed (15 DOWNTO 0);
   SIGNAL result_int: signed (15 DOWNTO 0);
   ATTRIBUTE preserve_signal OF pdt_int: SIGNAL IS TRUE;
   ATTRIBUTE dedicated_mult OF pdt_int: SIGNAL IS "OFF";
   ATTRIBUTE preserve_signal OF pdt2_int: SIGNAL IS TRUE;
   ATTRIBUTE dedicated_mult OF pdt2_int: SIGNAL IS "OFF";
BEGIN
   a_int <= signed (a);
   b_int <= signed (b);
   c_int <= signed (c);
   d_int <= signed (d);
   pdt_int <= a_int * b_int;
   pdt2_int <= c_int * d_int;
   result_int <= pdt_int + pdt2_int;
   result <= STD_LOGIC_VECTOR(result_int);
END rtl;
```

RAM and ROM

The Precision Synthesis software detects memory structures in HDL code and converts them to an operator that infers an ALTSYNCRAM or LPM_RAM_DP megafunction, depending on the device family. The software then places these functions in memory blocks.

The software supports inference for these functions only if the target device family has dedicated memory blocks.

For more information about inferring RAM and ROM megafunctions in HDL code, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*, and the *Precision Synthesis Style Guide* in the Precision Synthesis Manuals Bookcase. To access these manuals, in the Precision Synthesis software, click **Help** and select **Open Manuals Bookcase**.

Incremental Compilation and Block-Based Design

As designs become more complex and designers work in teams, a block-based hierarchical or incremental design flow is often an effective design approach. In an incremental compilation flow, you can make changes to a part of the design while maintaining the placement and performance of unchanged parts of the design. Design iterations can be made dramatically faster by focusing new compilations on particular design partitions and merging results with the results of previous compilations of other partitions. In a bottom-up or team-based approach, you can perform optimization on individual blocks and then integrate them into a final design and optimize it at the top level.

The first step in a hierarchical or incremental design flow is to make sure that different parts of your design do not affect each other. Ensure that you have separate netlists for each partition in your design so that you can take advantage of the incremental compilation design flow in the Quartus II software. If the whole design is in one netlist file, changes in one partition affect other partitions because of possible node name changes when you resynthesize the design.

You can create different implementations for each partition in your Precision Synthesis project, which allows you to switch between partitions without leaving the current project file. You can also create a separate project for each partition if you require separate projects for a bottom-up or team-based design flow. Alternatively, you can use the incremental synthesis capability in the Precision RTL Plus software.

For more information about creating partitions and using incremental compilation in the Quartus II software, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Creating a Design with Precision RTL Plus Incremental Synthesis

The Precision RTL Plus incremental synthesis flow for Quartus II incremental compilation uses a partition-based approach to achieve faster design iterations cycle time in one Precision implementation without sacrificing design performance.

Using the incremental synthesis feature, you can create different netlist files for different partitions of a design hierarchy within one partition implementation. This makes each partition independent of the others in either a top-down or a bottom-up incremental compilation flow. In either case, only the portions of a design that have been updated must be recompiled during design iterations. You can make changes and resynthesize one partition in a design to create a new netlist without affecting the synthesis results or fitting of other partitions.

The following steps show a general flow for partition-based incremental synthesis with Quartus II incremental compilation.

1. Create Verilog HDL or VHDL design files as you do in the regular design flow.
2. Determine which hierarchical blocks you want to treat as separate partitions in your design and indicate the partitions using the `incr_partition` attribute. For the syntax to create partitions, refer to “Creating Partitions with the incr_partition Attribute” on page 11–29.
3. Create a project in the Precision RTL Plus Synthesis software and add the HDL design files to the project.

4. Enable incremental synthesis in the Precision RTL Plus Synthesis software using one of these methods:
   - On the Tools menu, click **Set Options**. On the **Optimization** page, turn on **Enable Incremental Synthesis**.
   - Run the following command in the Transcript Window:

     ```
     setup_design -enable_incr_synth ↵
     ```
5. Run the basic Precision Synthesis flow of compile, synthesis, and place-and-route on your design. In subsequent runs, the Precision RTL Plus Synthesis software processes only the parts of the design that have changed, resulting in a shorter iteration than the initial run. The performance of the unchanged partitions is preserved.

   The Precision RTL Plus Synthesis software sets the netlist types of the unchanged partitions to **Post-Fit**, and the changed partitions to **Post-Synthesis**. You can change the netlist type during timing closure in the Quartus II software to get the best QoR.
6. Import the EDIF netlist for each partition and the top-level **.tcl** file into the Quartus II software and set up the Quartus II project to use incremental compilation.
7. Compile your Quartus II project.

   To change the Quartus II incremental compilation netlist type for a partition, on the Assignments menu, click **Design Partitions Window**.

   - To preserve the previous post-fit placement results, change the **Netlist Type** of the partition to **Post-Fit**.
   - To preserve the previous routing results, set the **Fitter Preservation Level** of the partition to **Placement and Routing**.

Creating Partitions with the incr_partition Attribute

Partitions are set using the HDL `incr_partition` attribute. The Precision Synthesis software creates or deletes partitions by looking at this attribute during compile iterations. The attribute can be attached to either the design unit definition, or an instance. Example 11–22 and Example 11–23 show how to use the attribute to create partitions. Simply remove the attribute or set the attribute value to false to delete the partitions.

The Precision Synthesis software ignores partitions set in a black box.

Example 11–22. Using incr_partition Attribute to Create Partition in Verilog HDL

Design unit partition:

```
module my_block(
  input clk;
  output reg [31:0] data_out) /* synthesis incr_partition */ ;
```

Instance partition:

```
my_block my_block_inst(.clk(clk), .data_out(data_out));
// synthesis attribute my_block_inst incr_partition true
```

Example 11–23. Using incr_partition Attribute to Create Partition in VHDL

Design unit partition:

```
entity my_block is
  port(
    clk : in std_logic;
    data_out : out std_logic_vector(31 downto 0)
  );
  attribute incr_partition : boolean;
  attribute incr_partition of my_block : entity is true;
end entity my_block;
```

Instance partition:

```
component my_block is
  port(
    clk : in std_logic;
    data_out : out std_logic_vector(31 downto 0)
  );
end component;

attribute incr_partition : boolean;
attribute incr_partition of my_block_inst : label is true;

my_block_inst my_block
  port map(clk, data_out);
```

Creating Multiple EDIF Netlist Files Using Separate Precision Projects or Implementations

This section describes how to manually generate multiple EDIF netlist files for incremental compilation using black boxes and separate Precision projects or implementations for each design partition. This manual flow is supported in versions of the Precision software that do not include the incremental synthesis feature. You might also use this feature if you perform synthesis in a team-based environment where there is no one top-level synthesis project that includes all of the lower-level design blocks.

In the Precision Synthesis software, create a separate implementation or a separate project for each lower-level module and for the top-level design that you want to maintain as a separate EDIF netlist file. Implement black box instantiations of lower-level modules in your top-level implementation or project.

For more information about managing implementations and projects, refer to the *Precision RTL Synthesis User's Manual*. To access this manual, in the Precision Synthesis software, click **Help** and select **Open Manuals Bookcase**.

When synthesizing the implementations for lower-level modules, perform these steps in the Precision Synthesis software:

1. On the Tools menu, turn off **Add IO Pads** on the **Optimization** page under **Set Options**.

You must turn off the **Add IO Pads** option while synthesizing the lower-level modules individually. Enable the **Add IO Pads** option only while synthesizing the top-level module.

2. Read the HDL files for the modules.

Modules can include black box instantiations of lower-level modules that are also maintained as separate EDIF files.

3. Add constraints for all partitions in the design.

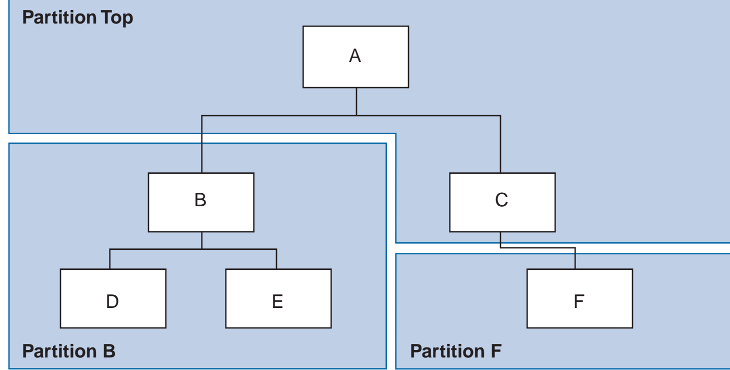
When synthesizing the top-level design implementation, perform these steps:

1. Read the HDL files for top-level designs.
2. On the Tools menu, click **Set Options**. On the **Optimization** page, turn on **Add IO Pads**.
3. Create black boxes for lower-level modules in the top-level design.
4. Add constraints.

In a top-down Quartus II incremental compilation flow, Precision constraints made on lower-level modules are not passed to the Quartus II software. Ensure that appropriate constraints are made in the top-level Precision Synthesis project, or in the Quartus II project.

The following sections describe an example of implementing black boxes to create separate EDIF netlists. Figure 11–2 shows an example of a design hierarchy separated into various partitions.

Figure 11–2. Partitions in a Hierarchical Design



In Figure 11–2, the top-level partition contains the top-level block in the design (block A) and the logic that is not defined as part of another partition. In this example, the partition for top-level block A also includes the logic in the C sub-block. Because block F is contained in its own partition, it is not treated as part of the top-level partition A. Another separate partition, B, contains the logic in blocks B, D, and E. In a team-based design, different engineers may work on the logic in different partitions. One netlist is created for the top-level module A and its submodule C, another netlist is created for module B and its submodules D and E, while a third netlist is created for module F. To create multiple EDIF netlist files for this design, follow these steps:

1. Generate an EDIF file for module B. Use **B.v/.vhd**, **D.v/.vhd**, and **E.v/.vhd** as the source files.
2. Generate an EDIF file for module F. Use **F.v/.vhd** as the source file.
3. Generate a top-level EDIF file for module A. Use **A.v/.vhd** and **C.v/.vhd** as the source files. Ensure that you create black boxes for modules B and F, which were optimized separately in the previous steps.

The goal is to individually synthesize and generate an EDIF netlist file for each lower-level module and then instantiate these modules as black boxes in the top-level file. You can then synthesize the top-level file to generate the EDIF netlist file for the top-level design. Finally, both the lower-level and top-level EDIF netlist files are provided to your Quartus II project.

When you make design or synthesis optimization changes to part of your design, resynthesize only the changed partition to generate the new EDIF netlist file. Do not resynthesize the implementations or projects for the unchanged partitions.

Creating Black Boxes in Verilog HDL

Any design block that is not defined in the project or included in the list of files to be read for a project is treated as a black box by the software. In Verilog HDL, you must provide an empty module declaration for any module that is treated as a black box.

A black box for the top-level file **A.v** is shown in the following example. Use this same procedure for any lower-level files, which also contain a black box for any module beneath the current level of hierarchy.

Example 11–24. Verilog HDL Black Box for Top-Level File A.v

```
module A (data_in, clk, e, ld, data_out);
   input data_in, clk, e, ld;
   output [15:0] data_out;
   wire [15:0] cnt_out;
   B U1 (.data_in (data_in),.clk(clk), .ld (ld),.data_out(cnt_out));
   F U2 (.d(cnt_out), .clk(clk), .e(e), .q(data_out));
   // Any other code in A.v goes here.
endmodule
// Empty Module Declarations of Sub-Blocks B and F follow here.
// These module declarations (including ports) are required for black
// boxes.
module B (data_in, clk, ld, data_out);
   input data_in, clk, ld;
   output [15:0] data_out;
endmodule
module F (d, clk, e, q);
   input [15:0] d;
   input clk, e;
   output [15:0] q;
endmodule
```

Creating Black Boxes in VHDL

Any design block that is not defined in the project or included in the list of files to be read for a project is treated as a black box by the software. In VHDL, you must have a component declaration for the black box just like any other block in the design.

A black box for the top-level file **A.vhd** is shown in Example 11–25. Follow this same procedure for any lower-level files that also contain a black box or for any block beneath the current level of hierarchy.

Example 11–25. VHDL Black Box for Top-Level File A.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY A IS
   PORT ( data_in : IN INTEGER RANGE 0 TO 15;
      clk, e, ld : IN STD_LOGIC;
      data_out : OUT INTEGER RANGE 0 TO 15);
END A;
ARCHITECTURE a_arch OF A IS
COMPONENT B PORT(
   data_in : IN INTEGER RANGE 0 TO 15;
   clk, ld : IN STD_LOGIC;
   d_out : OUT INTEGER RANGE 0 TO 15);
END COMPONENT;
COMPONENT F PORT(
   d : IN INTEGER RANGE 0 TO 15;
   clk, e: IN STD_LOGIC;
   q : OUT INTEGER RANGE 0 TO 15);
END COMPONENT;
-- Other component declarations in A.vhd go here
signal cnt_out : INTEGER RANGE 0 TO 15;
BEGIN
   U1 : B
   PORT MAP (
      data_in => data_in,
      clk => clk,
      ld => ld,
      d_out => cnt_out);
   U2 : F
   PORT MAP (
      d => cnt_out,
      clk => clk,
      e => e,
      q => data_out);
   -- Any other code in A.vhd goes here
END a_arch;
```

After you complete the steps outlined in this section, you have different EDIF netlist files for each partition of the design. These files are ready for use with incremental compilation in the Quartus II software.

Creating Quartus II Projects for Multiple EDIF Files

The Precision Synthesis software creates a **.tcl** file for each implementation, and provides the Quartus II software with the appropriate constraints and information to set up a project. When using incremental synthesis, the Precision RTL Plus Synthesis software creates only a single **.tcl** file, <*project name*>**_incr_partitions.tcl**, to pass the partition information to the Quartus II software. For details about using the **.tcl** script generated by the Precision Synthesis software to set up your Quartus II project and to pass your top-level constraints, refer to "Running the Quartus II Software Manually Using the Precision Synthesis-Generated Tcl Script" on page 11–13.

 © March 2009 Altera Corporation

Depending on your design methodology, you can create one Quartus II project for all EDIF netlists (a top-down flow), or a separate Quartus II project for each EDIF netlist (a bottom-up flow). In a top-down compilation design flow, you create design partition assignments for each partition in the design within a single Quartus II project. This methodology provides the best QoR and performance preservation during incremental changes to your design. You might have to use a bottom-up design flow when each partition must be optimized separately, such as in certain team-based design flows.

To perform a bottom-up compilation in the Quartus II software, create separate Quartus II projects and import each design partition into a top-level design using the incremental compilation export and import features to maintain placement results.

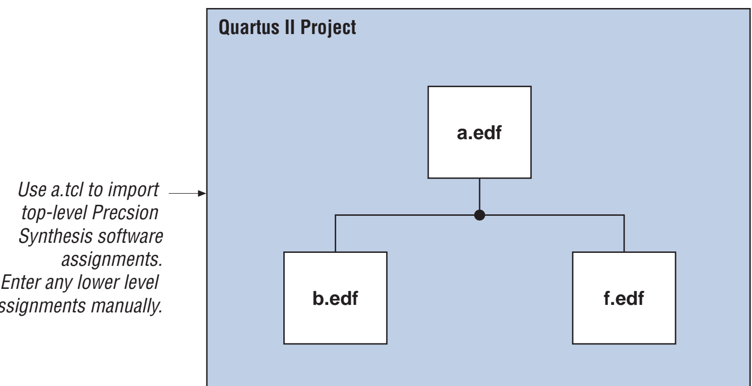
The following sections describe how to create the Quartus II projects for these two design flows.

Creating a Single Quartus II Project for a Top-Down Incremental Compilation Flow

Use the <*top-level project*>**.tcl** file generated for the top-level partition to create your Quartus II project and import all the netlists into this one Quartus II project for an incremental compilation flow. You can optimize all partitions within the single Quartus II project and take advantage of the performance preservation and compilation time reduction that incremental compilation provides. Figure 11–3 shows the design flow for the example design in Figure 11–2 on page 11–31.

All the constraints from the top-level implementation are passed to the Quartus II software in the top-level **.tcl** file, but any constraints made only in the lower-level implementations within the Precision Synthesis software are not forward-annotated. Enter these constraints manually in your Quartus II project.

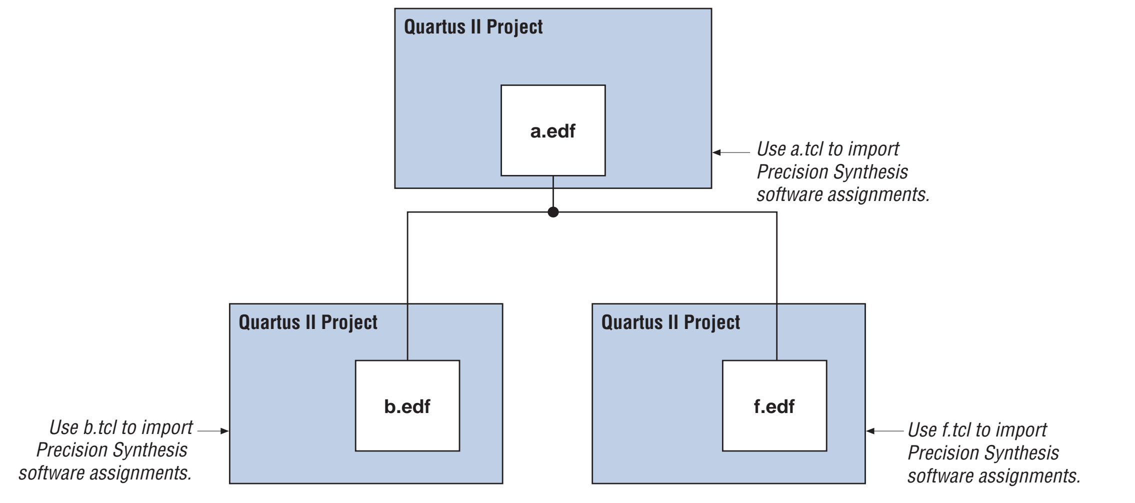
Figure 11–3. Design Flow Using Multiple EDIF Files with One Quartus II Project



Creating Multiple Quartus II Projects for a Bottom-Up Flow

Use the **.tcl** files generated by the Precision Synthesis software for each Precision Synthesis software implementation or project to generate multiple Quartus II projects, one for each partition in the design. Each designer in the project can optimize their block separately in the Quartus II software and export the placement of their blocks using incremental compilation. Designers should create a LogicLock region to provide a floorplan location assignment for each block; the top-level designer should then import all the blocks and assignments into the top-level project. Figure 11–4 shows the design flow for the example design in Figure 11–2 on page 11–31.

Figure 11–4. Design Flow: Using Multiple EDIF Files with Multiple Quartus II Projects



Hierarchy and Design Considerations

To ensure the proper functioning of the synthesis flow, you can create separate partitions only for modules, entities, or existing netlist files. In addition, each module or entity must have its own design file. If two different modules are in the same design file but are defined as being part of different partitions, you cannot maintain incremental synthesis because both regions must be recompiled when you change one of the modules.

Altera recommends that you register all inputs and outputs of each partition. This makes logic synchronous and avoids any delay penalty on signals that cross partition boundaries.

If you use boundary tri-states in a lower-level block, the Precision Synthesis software pushes the tri-states through the hierarchy to the top level to make use of the tri-state drivers on output pins of Altera devices. Because pushing tri-states requires optimizing through hierarchies, lower-level tri-states are not supported with a block-based compilation methodology. You should use tri-state drivers only at the external output pins of the device and in the top-level block in the hierarchy.

For more tips on design partitioning, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.

Conclusion

Advanced synthesis is an important part of the design flow. The Mentor Graphics Precision Synthesis software and Quartus II design flow allow you to control how to prepare your design files for the Quartus II place-and-route process. This allows you to improve performance and optimize your design for use with Altera devices. Several of the methodologies outlined in this chapter can help you optimize your design to achieve performance goals and decrease design time.

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*
- *Precision RTL Synthesis User's Manual* in the Precision Manuals Bookcase
- *Precision Synthesis Style Guide* in the Precision Manuals Bookcase
- *Precision Synthesis Reference Manual* in the Precision Manuals Bookcase
- *Specifying EDA Tool Settings* in the Quartus II Help index
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*

Document Revision History

Table 11–7 shows the revision history for this chapter.

Table 11–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Updated list of supported devices for the Quartus II software version 9.0 release ■ Chapter 11 was previously Chapter 10 in software version 8.1	Updated chapter for the Quartus II software version 9.0 release.
November 2008 v8.1.0	■ Changed to 8-1/2 x 11 page size ■ Title changed to *Mentor Graphics Precision Synthesis Support* ■ Updated list of supported devices ■ Added information about the Precision RTL Plus incremental synthesis flow ■ Updated Figure 10-1 to include SystemVerilog ■ Updated "Guidelines for Altera Megafunctions and Architecture-Specific Features" on page 10–19 ■ Updated "Incremental Compilation and Block-Based Design" on page 10–28 ■ Added section "Creating Partitions with the incr_partition Attribute" on page 10–29	Updated chapter for the Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Removed Mercury from the list of supported devices ■ Changed Precision version to 2007a update 3 ■ Added note for Stratix IV support ■ Renamed "Creating a Project and Compiling the Design" section to "Creating and Compiling a Project in the Precision RTL Synthesis Software" ■ Added information about constraints in the Tcl file ■ Updated document based on the Quartus II software version 8.0	Updated chapter based on the Quartus II software version 8.0

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

12. Mentor Graphics LeonardoSpectrum Support

QII51010-9.0.0

Introduction

This chapter documents key design methodologies and techniques for Altera® devices using the LeonardoSpectrum and Quartus II design flow. Combining HDL coding techniques, Mentor Graphics LeonardoSpectrum™ software constraints, and Quartus® II options provide the performance increase required for today's system-on-a-programmable-chip (SOPC) designs.

The LeonardoSpectrum software is a mature synthesis tool supporting legacy devices and many current devices. The LeonardoSpectrum software version 2008b supports the Arria® GX, Stratix® IV, Stratix III, Stratix II, Stratix, Stratix GX, Cyclone® III, Cyclone II, Cyclone, MAX® II, MAX series, APEX™ series, FLEX® series, and ACEX® series device families. Altera recommends using the advanced Precision Synthesis software for new designs in new device families.

For more information about Precision RTL Synthesis, refer to the *Mentor Graphics Precision RTL Synthesis Support* chapter in volume 1 of the *Quartus II Handbook*.

This chapter assumes that you have set up, licensed, and are familiar with the LeonardoSpectrum software.

To obtain and license the LeonardoSpectrum software, refer to the Mentor Graphics website at www.mentor.com. For information about installing the LeonardoSpectrum software and setting up your working environment, refer to the *LeonardoSpectrum Installation Guide* and the *LeonardoSpectrum User's Manual*.

Design Flow

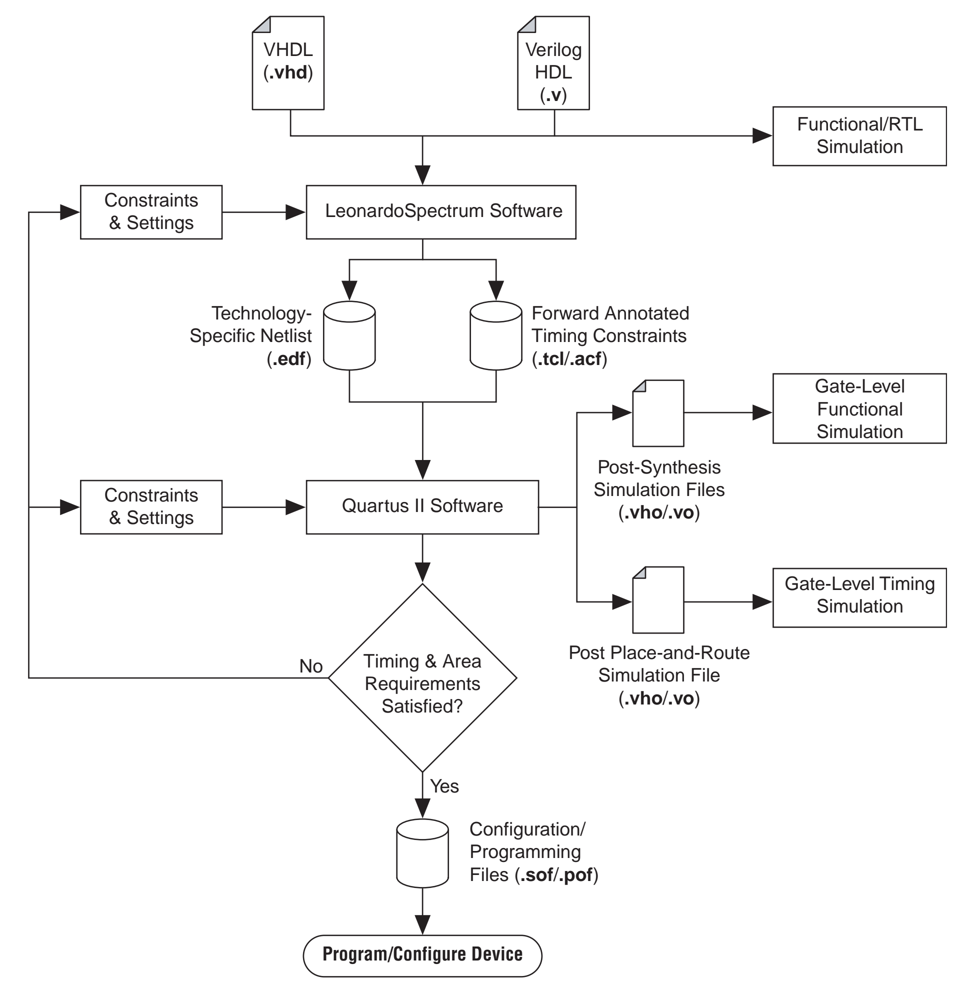
The following are basic steps in a LeonardoSpectrum-Quartus II design flow:

1. Create Verilog HDL or VHDL design files in the LeonardoSpectrum software or a text editor.
2. Import the Verilog HDL or VHDL design files into the LeonardoSpectrum software for synthesis.
3. Select a target device and add timing constraints and compiler directives to help optimize the design during synthesis.
4. Synthesize the project in the LeonardoSpectrum software.
5. Create a Quartus II project and import the technology-specific EDIF Input File (**.edf**) netlist and the Tcl Script File (**.tcl**) generated by the LeonardoSpectrum software into the Quartus II software for placement and routing, and for performance evaluation.
6. After obtaining place-and-route results that meet your requirements, configure or program the Altera device.

Figure 12–1 shows the recommended design flow using the LeonardoSpectrum and Quartus II software.

If your area and timing requirements are satisfied, use the programming files generated by the Quartus II software to program or configure the Altera device. As shown in Figure 12–1, if the area or timing requirements are not met, change the constraints in the LeonardoSpectrum software and re-run the synthesis. Repeat the process until the area and timing requirements are met. You can also use other Quartus II software options and techniques to meet the area and timing requirements.

Figure 12–1. Recommended Design Flow Using LeonardoSpectrum and Quartus II Software



The LeonardoSpectrum software supports both VHDL and Verilog HDL source files. With the appropriate license, it also supports mixed synthesis, allowing a combination of VHDL and Verilog HDL source files. After synthesis, the LeonardoSpectrum software produces several intermediate and output files. Table 12–1 lists these file extensions with a short description of each file.

Table 12–1. LeonardoSpectrum Intermediate and Output Files

File Extension(s)	File Description
.xdb	Technology-independent register transfer level (RTL) netlist file that can only be read by the LeonardoSpectrum software.
.edf	Technology-specific output netlist in electronic design interchange format (EDIF).

Table 12–1. LeonardoSpectrum Intermediate and Output Files

File Extension(s)	File Description
.acf/**.tcl** *(1)*	Forward-annotated constraint file containing constraints and assignments.

Note to Table 12–1:

(1) An assignment and configuration (**.acf**) file is created only for ACEX 1K, FLEX series, and MAX series devices. The assignment and configuration file is generated for backward compatibility with the MAX+PLUS® II software. A **.tcl** file is generated for the Quartus II software that also contains Tcl commands to create a Quartus II project.

Altera recommends that you do not use project directory names that include spaces. Some file operations in the LeonardoSpectrum software do not work correctly if the path name contains spaces.

Specify timing constraints and compiler directives for the design in the LeonardoSpectrum software, or in a constraint file (**.ctr**). Many of these constraints are forward-annotated in the **.tcl** file for use by the Quartus II software.

The LeonardoInsight™ Schematic Viewer is an add-on graphical tool for schematic views of the technology-independent RTL netlist (**.xdb**) and the technology-specific gate-level results. You can use the Schematic Viewer to visually analyze and debug the design. It also supports cross-probing between the RTL and gate-level schematics, the design browser, and the source code in the HDLInventor™ text editor.

Optimization Strategies

You can configure most general settings in the **Quick Setup** tab in the LeonardoSpectrum user interface. Other Flow tabs provide additional options, and some Flow tabs include multiple Power tabs (at the bottom of the screen) with more options. Advanced optimization options in the LeonardoSpectrum software include timing-driven synthesis, encoding style, resource sharing, and mapping I/O registers.

Timing-Driven Synthesis

The LeonardoSpectrum software supports timing-driven synthesis through user-assigned timing constraints to optimize the performance of the design. The process for setting constraints in the LeonardoSpectrum software is straightforward. Constraints such as clock frequency can be specified globally or for individual clock signals. The following sections describe how to set the various types of timing constraints in the LeonardoSpectrum software.

The timing constraints described in "Global Power Tab" are set in the **Constraints** Flow tab. In this tab, there are Power tabs at the bottom, such as **Global** and **Clock**, for setting various constraints.

Global Power Tab

The **Global** tab is the default Power tab in the **Constraints** Flow tab where you can specify the global clock frequency. The **Clock Frequency** on the **Quick Setup** tab is equivalent to the **Registers to Registers** delay setting. You can also specify the **Input Ports to Registers**, **Registers to Output Ports**, and **Inputs to Outputs** delays that correspond to global t_{SU}, t_{CO}, and t_{PD} requirements, respectively, in the Quartus II software. The timing diagram on this tab reflects the settings you have made.

Clock Power Tab

You can set various constraints for each clock in your design. First, select the clock name in the Clock(s) window. The clock names appear after the design is read from the **Input** Flow tab. Configure settings for that particular clock and click **Apply**. If necessary, you can also set the **Duty Cycle** to a value other than the default 50%. The timing diagram shows these settings.

If a clock has an **Offset** from the main clock, which is considered to be time "0", this constraint corresponds to the `OFFSET_FROM_BASE_CLOCK` setting in the Quartus II software.

You can specify the pin number for the clock input pin in the **Pin Location** field. This pin number is passed to the Quartus II software for place-and-route, but does not affect synthesis in the LeonardoSpectrum software.

Input and Output Power Tabs

Configure settings for individual input or output pins in the **Input** and **Output** tabs. First, select a name in the Input Ports or Output Ports window. The names appear after the design is read from the **Input** Flow tab. Then make the setting for that pin as described below.

The **Arrival Time** setting indicates that the input signal arrives a specified time after the rising clock edge (time "0"). This setting constrains the path from the pin to the first register by including the arrival time in the total delay, and corresponds to the `EXTERNAL_INPUT_DELAY` assignment in the Quartus II software.

The **Required Time** setting indicates the maximum delay after time "0" that the output signal should arrive at the output pin. This setting directly constrains the register to output delay, and corresponds with the `EXTERNAL_OUTPUT_DELAY` assignment in the Quartus II software.

Specify the pin number for the I/O pin in the **Pin Location** field. This pin number is passed to the Quartus II software for place-and-route, but does not affect synthesis in the LeonardoSpectrum software.

Other Constraints

The following sections describe other constraints that can be set with the LeonardoSpectrum user interface and contain these topics:

- "Encoding Style"
- "Resource Sharing"
- "Mapping I/O Registers"

Encoding Style

The LeonardoSpectrum software encodes state machines during the synthesis process. To improve performance when coding state machines, separate state machine logic from all arithmetic functions and data paths. When encoded, a design cannot be re-encoded later in the optimization process. You must follow a particular VHDL or Verilog HDL coding style for the LeonardoSpectrum software to identify the state machine.

Table 12–2 shows the state machine encoding styles supported by the LeonardoSpectrum software.

Table 12–2. State Machine Encoding Styles in the LeonardoSpectrum Software

Style	Description
Binary	Generates state machines with the fewest possible flipflops. Binary state machines are useful for area-critical designs when timing is not the primary concern.
Gray	Generates state machines where only one flipflop changes during each transition. Gray-encoded state machines tend to be glitchless.
One-hot	Generates state machines containing one flipflop for each state. One-hot state machines provide the best performance and shortest clock-to-output delays. However, one-hot implementations are usually larger than binary implementations.
Random	Generates state machines using random state machine encoding. Only use random state machine encoding when no other implementation achieves the desired results.
Auto (Default)	Implements binary or one-hot encoding, depending on the size of enumerated types in the state machine.

The **Encoding Style** setting is created in the **Input** Flow tab. It instructs the software to use a particular state machine encoding style for all state machines. The default **Auto** selection implements binary or one-hot encoding, depending on the size of enumerated types in the state machine.

To ensure proper recognition and improve performance when coding state machines, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook* for design guidelines.

Resource Sharing

You can also enable the **Resource Sharing** setting in the **Input** Flow tab. This setting allows optimization to reduce device resources. You should generally leave this setting turned on.

Mapping I/O Registers

The **Map I/O Registers** option is located in the **Technology** Flow tab. The **Map I/O Registers** option applies to Altera FPGAs containing I/O cells (IOCs) or I/O elements (IOEs). If the option is turned on, input or output registers are moved into the device's I/O cells for faster setup or clock-to-output times.

Timing Analysis with the Leonardo-Spectrum Software

The LeonardoSpectrum software reports successful synthesis with an information message in the Transcript or Information window. Estimated device usage and timing results are reported in the Device Utilization section of this window. Figure 12–2 shows an example of a LeonardoSpectrum compilation report.

Figure 12–2. LeonardoSpectrum Compilation Report

```
*************************************************
Device Utilization for EP20K200EQC208
*************************************************
Resource                  Used    Avail   Utilization
-------------------------------------------------
IOs                       22      136      16.18%
LCs                       114     8320      1.37%
Memory Bits               0       106496    0.00%


-------------------------------------------------
                          Clock Frequency Report


        Clock                   : Frequency
      ------------------------------------


        clk                     : 52.2 MHz
        clk2                    : 149.5 MHz


                          Critical Path Report
```

The LeonardoSpectrum software estimates the timing results based on timing models. The LeonardoSpectrum software has no information about how the design is placed and routed in the Quartus II software, so it cannot report accurate routing delays. Additionally, if the design includes any black boxed Altera-specific functions, the LeonardoSpectrum software does not report timing information for these functions.

Final timing results are generated by the Quartus II software and are reported separately in the Transcript or Information window if the **Run Integrated Place and Route** option is turned on. Refer to "Integration with the Quartus II Software" for more information.

Exporting Designs Using NativeLink Integration

You can use NativeLink® integration to integrate the LeonardoSpectrum software and the Quartus II software with a single GUI for both the synthesis and place-and-route operations. You can run the Quartus II software from within the LeonardoSpectrum software GUI with NativeLink integration or you can run the LeonardoSpectrum software from within the Quartus II software GUI for device families supported in the Quartus II software.

Generating Netlist Files

The LeonardoSpectrum software generates an **.edif** netlist file readable as an input file in the Quartus II software for place-and-route. Select the **.edif** file option name in the **Output** Flow tab. The **.edif** netlist file is also generated if the **Auto** option is turned on in the **Output** Flow tab.

Including Design Files for Black Boxed Modules

If the design has black boxed megafunctions, be sure to include the MegaWizard™ Plug-In Manager-generated custom megafunction variation design file in the Quartus II project directory, or add it to the list of project files for place-and-route.

Passing Constraints with Scripts

The LeonardoSpectrum software can write out a **.tcl** file called <*project name*>**.tcl**. This file contains commands to create a Quartus II project along with constraints and other assignments. To output a Tcl script, turn on the **Write Vendor Constraint Files** option in the **Output** Flow tab.

To create and compile a Quartus II project using the **.tcl** file generated from the LeonardoSpectrum software, perform the following steps in the Quartus II software:

1. Place the **.edif** netlist files and Tcl scripts in the same directory.
2. On the View menu, point to **Utility**, and click **Tcl Console** to open the Quartus II Tcl Console.
3. Type `source <path>/<project name>.tcl` at a Tcl Console command prompt.
4. On the File menu, click **Open Project** to open the new project. On the Processing menu, click **Start Compilation**.

Integration with the Quartus II Software

You can launch the Quartus II software from within the LeonardoSpectrum software with the **Place And Route** section in the **Quick Setup** tab. Turn on the **Run Integrated Place and Route** option to start the compilation using the Quartus II software to show the fitting and performance results. You can also run the place-and-route software by turning on the **Run Quartus** option on the **Physical** Flow tab and clicking **Run PR**.

To use integrated place-and-route software, on the Options menu, point to **Place and Route Path** and click **Tools**. Specify the location of the Quartus II software executable file (browse to <*Quartus II software installation directory*>**/bin**).

Guidelines for Altera Megafunctions and LPM Functions

Altera provides parameterizable megafunctions ranging from simple arithmetic units, such as adders and counters, to advanced phase-locked loop (PLL) blocks, multipliers, and memory structures. These functions are performance-optimized for Altera devices. Megafunctions include the library of parameterized modules (LPM), device-specific megafunctions such as PLLs, LVDS, and digital signal processing (DSP) blocks, intellectual property (IP) available as Altera MegaCore® functions, and IP available through the Altera Megafunction Partners Program (AMPP℠).

Some IP cores require that you synthesize them in the LeonardoSpectrum software. Refer to the user guide for the specific IP.

There are two methods for handling megafunctions in the LeonardoSpectrum software: inference and instantiation.

The LeonardoSpectrum software supports inferring some of the Altera megafunctions, such as multipliers, DSP functions, and RAM and ROM blocks. The LeonardoSpectrum software supports all Altera megafunctions through instantiation.

Instantiating Altera Megafunctions

There are two methods of instantiating Altera megafunctions in the LeonardoSpectrum software. The first and least common method is to directly instantiate the megafunction in the Verilog HDL or VHDL code. The second method, to maintain target technology awareness, is to use the MegaWizard Plug-In Manager in the Quartus II software to set up and parameterize a megafunction variation. The megafunction wizard creates a wrapper file that instantiates the megafunction. The advantage of using the megafunction wizard in place of the instantiation method is the megafunction wizard properly sets all the parameters and you do not need the library support required in the direct instantiation method. This is referred to as black box methodology.

Altera recommends using the MegaWizard Plug-In Manager to ensure that the ports and parameters are set correctly.

When directly instantiating megafunctions, see the Quartus II Help for a list of the ports and parameters.

Inferring Altera Memory Elements

The LeonardoSpectrum software can infer memory blocks from Verilog HDL or VHDL code. When the LeonardoSpectrum software detects a RAM or ROM from the style of the RTL code at a technology-independent level, it then maps the element to a generic module in the RTL database. During the technology-mapping phase of synthesis, the LeonardoSpectrum software maps the generic module to the most optimal primitive memory cells, or Altera megafunction, for the target Altera technology.

For more information about inferring RAM and ROM megafunctions, including examples of VHDL and Verilog HDL code, see the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Inferring RAM

The LeonardoSpectrum software supports RAM inference for various device families. The following are the restrictions for the LeonardoSpectrum software to successfully infer RAM in a design:

- The write process must be synchronous
- The read process can be asynchronous or synchronous depending on the target Altera architecture
- Resets on the memory are not supported

Table 12–3 shows a summary of the minimum memory sizes and minimum address widths for inferring RAM in various device families.

Table 12–3. Inferring RAM Summary

Devices	RAM Primitive	Minimum RAM Size	Minimum Address Width
Stratix Series and Cyclone Series	`altsyncram`	2 bits	1 bit
APEX Series, Excalibur and Mercury	`altdpram`	64 bits	4 bits
FLEX 10KE and ACEX 1K	`altdpram`	128 bits	5 bits

To disable RAM inference, set the `extract_ram` and `infer_ram` variables to “false.” On the Tools menu, click **Variable Editor** to enter the value “false” when synthesizing in the user interface with the **Advanced** Flow tabs, or add the commands `set extract_ram false` and `set infer_ram false` to your synthesis script.

Inferring ROM

You can implement ROM behavior in HDL source code with CASE statements or specify the ROM as a table. The LeonardoSpectrum software infers both synchronous and asynchronous ROM depending on the target Altera device. For example, memory for the Stratix series devices must be synchronous to be inferred.

To disable ROM inference, set the `extract_rom` variable to “false.” To enter the value “false” when synthesizing in the user interface with the **Advanced Flow** tabs, on the Tools menu, click **Variable Editor**, or add the commands set `extract_rom` false to your synthesis script.

Inferring Multipliers and DSP Functions

Some Altera devices include dedicated DSP blocks optimized for DSP applications. The following Altera megafunctions are used with DSP block modes:

- LPM_MULT
- ALTMULT_ACCUM
- ALTMULT_ADD

You can instantiate these megafunctions in the design or have the LeonardoSpectrum software infer the appropriate megafunction by recognizing a multiplier, multiplier-accumulator (MAC), or multiplier-adder in the design. The Quartus II software maps the functions to the DSP blocks in the device during place-and-route.

For more information about inferring multipliers and DSP functions, including examples of VHDL and Verilog HDL code, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Simple Multipliers

The LPM_MULT megafunction implements the DSP block in the simple multiplier mode. The following functionality is supported in this mode:

- The DSP block includes registers for the input and output stages, and an intermediate pipeline stage
- Signed and unsigned arithmetic is supported

Multiplier Accumulators

The ALTMULT_ACCUM megafunction implements the DSP block in the multiply-accumulator mode. The following functionality is supported in this mode:

- The DSP block includes registers for the input and output stages, and an intermediate pipeline stage
- The output registers are required for the accumulator
- The input and pipeline registers are optional

- Signed and unsigned arithmetic is supported

If the design requires input registers to be used as shift registers, use the black box method to instantiate the ALTMULT_ACCUM megafunction.

Multiplier Adders

The LeonardoSpectrum software can infer multiplier adders and map them to either the two-multiplier adder mode or the four-multiplier adder mode of the DSP blocks. The LeonardoSpectrum software maps the HDL code to the correct ALTMULT_ADD function.

The following functionality is supported in these modes:

- The DSP block includes registers for the input and output stages, and an intermediate pipeline stage
- Signed and unsigned arithmetic is supported, but support for the Verilog HDL "signed" construct is limited

Controlling DSP Block Inference

Device features, such as dedicated DSP blocks, multipliers, multiply-accumulators, and multiply-adders can be implemented in DSP blocks or in logic. You can control this implementation through attribute settings in the LeonardoSpectrum software.

As shown in Table 12–4, attribute settings in the LeonardoSpectrum software control the implementation of the multipliers in DSP blocks or logic at the signal block (or module), and project level.

Table 12–4. Attribute Settings for DSP Blocks in the LeonardoSpectrum Software *(Note 1)*

Level	Attribute Name	Value	Description
Global	`extract_mac` *(2)*	TRUE	All multipliers in the project mapped to DSP blocks.
		FALSE	All multipliers in the project mapped to logic.
Module	`extract_mac` *(3)*	TRUE	Multipliers inside the specified module mapped to DSP blocks.
		FALSE	Multipliers inside the specified module mapped to logic.
Signal	`dedicated_mult`	ON	LPM inferred and multipliers implemented in DSP block.
		OFF	LPM inferred, but multipliers implemented in logic by the Quartus II software.
		LCELL	LPM not inferred, and multipliers implemented in logic by the LeonardoSpectrum software.
		AUTO	LPM inferred, but the Quartus II software automatically maps the multipliers to either logic or DSP blocks based on the Quartus II software place-and-route.

Notes to Table 12–4:

(1) The `extract_mac` attribute takes precedence over the `dedicated_mult` attribute.
(2) For devices with DSP blocks, the `extract_mac` attribute is set to "true" by default for the entire project.
(3) For devices with DSP blocks, the `extract_mac` attribute is set to "true" by default for all modules.

Global Attribute

You can set the global attribute `extract_mac` to control the implementation of multipliers in DSP blocks for the entire project. You can set this attribute using the script interface. The script command is:

`set extract_mac` *<value>*

Module Level Attributes

You can control the implementation of multipliers inside a module or component by setting attributes in the Verilog HDL source code. The attribute used is `extract_mac`. Setting this attribute for a module affects only the multipliers inside that module. The command is:

`//synthesis attribute` *<module name>* `extract_mac` *<value>*

The Verilog HDL and VHDL code samples in Example 12–1 and Example 12–2 show how to use the `extract_mac` attribute.

Example 12–1. Using Module Level Attributes in Verilog HDL Code

```
module mult_add ( dataa, datab, datac, datad, result);
//synthesis attribute mult_add extract_mac FALSE
// Port Declaration
input [15:0] dataa;
input [15:0] datab;
input [15:0] datac;
input [15:0] datad;

output [32:0] result;

// Wire Declaration
wire [31:0] mult0_result;
wire [31:0] mult1_result;

// Implementation
// Each of these can go into one of the 4 mults in a
// DSP block
assign mult0_result = dataa * `signed datab;
//synthesis attribute mult0_result preserve_signal TRUE

assign mult1_result = datac * datad;

// This adder can go into the one-level adder in a DSP
// block
assign result = (mult0_result + mult1_result);

endmodule
```

Example 12–2. Using Module Level Attributes in VHDL Code

```
library ieee ;
USE ieee.std_logic_1164.all;

USE ieee.std_logic_arith.all;

entity mult_acc is
   generic (size : integer := 4) ;
   port (
      a: in std_logic_vector (size-1 downto 0) ;
      b: in std_logic_vector (size-1 downto 0) ;
      clk : in std_logic;
       accum_out: inout std_logic_vector (2*size downto 0)
   ) ;
   attribute extract_mac : boolean;
   attribute extract_mac of mult_acc : entity is FALSE;
end mult_acc;

architecture synthesis of mult_acc is
    signal a_int, b_int : signed (size-1 downto 0);
    signal pdt_int : signed (2*size-1 downto 0);
    signal adder_out : signed (2*size downto 0);

begin
    a_int <= signed (a);
    b_int <= signed (b);
    pdt_int <= a_int * b_int;
    adder_out <= pdt_int + signed(accum_out);
    process (clk)
    begin
       if (clk'event and clk = '1') then
          accum_out <= std_logic_vector (adder_out);
       end if;
    end process;
end synthesis ;
```

Signal Level Attributes

You can control the implementation of individual LPM_MULT multipliers by using the `dedicated_mult` attribute, as shown below:

`//synthesis attribute` <*signal_name*> `dedicated_mult` <*value*>

The `dedicated_mult` attribute is only applicable to signals or wires; it is not applicable to registers.

Table 12–5 shows the supported values for the `dedicated_mult` attribute.

Table 12–5. Values for the dedicated_mult Attribute (Part 1 of 2)

Value	Description
ON	LPM inferred and multipliers implemented in DSP block.
OFF	LPM inferred and multipliers synthesized, implemented in logic, and optimized by the Quartus II software. *(1)*
LCELL	LPM not inferred and multipliers synthesized, implemented in logic, and optimized by the LeonardoSpectrum software. *(1)*

Table 12–5. Values for the dedicated_mult Attribute (Part 2 of 2)

Value	Description
AUTO	LPM inferred but the Quartus II software maps the multipliers automatically to either the DSP block or logic based on resource availability.

Note to Table 12–5:

(1) Although both dedicated_mult=OFF and dedicated_mult=LCELLS result in logic implementations, the optimized results in these two cases may differ.

Some signals for which the `dedicated_mult` attribute is set might get synthesized away by the LeonardoSpectrum software due to design optimization. In such cases, if you want to force the implementation, the signal is preserved from being synthesized away by setting the `preserve_signal` attribute to "true."

The `extract_mac` attribute must be set to "false" for the module or project level when using the `dedicated_mult` attribute.

Example 12–3 and Example 12–4 are samples of Verilog HDL and VHDL codes, respectively, using the `dedicated_mult` attribute.

Example 12–3. Signal Attributes for Controlling DSP Block Inference in Verilog HDL Code

```
module mult (AX, AY, BX, BY, m, n, o, p);
input [7:0] AX, AY, BX, BY;
output [15:0] m, n, o, p;
wire [15:0] m_i = AX * AY; // synthesis attribute m_i dedicated_mult ON
// synthesis attribute m_i preserve_signal TRUE
//Note that the preserve_signal attribute prevents
// signal m_i from getting synthesized away
wire [15:0] n_i = BX * BY; // synthesis attribute n_i dedicated_mult OFF
wire [15:0] o_i = AX * BY; // synthesis attribute o_i dedicated_mult AUTO
wire [15:0] p_i = BX * AY; // synthesis attribute p_i dedicated_mult
LCELL
// since n_i , o_i , p_i signals are not preserved,
// they may be synthesized away based on the design
assign m = m_i;
assign n = n_i;
assign o = o_i;
assign p = p_i;
endmodule
```

Example 12–4. Signal Attributes for Controlling DSP Block Inference in VHDL Code

```
library ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_signed.all;
ENTITY mult is

PORT( AX,AY,BX,BY: IN
std_logic_vector (17 DOWNTO 0);
m,n,o,p: OUT
std_logic_vector (35 DOWNTO 0));
attribute dedicated_mult: string;
attribute preserve_signal : boolean
END mult;
ARCHITECTURE struct of mult is

signal m_i, n_i, o_i, p_i : unsigned (35 downto 0);
attribute dedicated_mult of m_i:signal is "ON";
attribute dedicated_mult of n_i:signal is "OFF";
attribute dedicated_mult of o_i:signal is "AUTO";
attribute dedicated_mult of p_i:signal is "LCELL";

begin

m_i <= unsigned (AX) * unsigned (AY);
n_i <= unsigned (BX) * unsigned (BY);
o_i <= unsigned (AX) * unsigned (BY);
p_i <= unsigned (BX) * unsigned (AY);

m <= std_logic_vector(m_i);
n <= std_logic_vector(n_i);
o <= std_logic_vector(o_i);
p <= std_logic_vector(p_i);
end struct;
```

Guidelines for Using DSP Blocks

In addition to the guidelines mentioned earlier in this section, use the following guidelines while designing with DSP blocks in the LeonardoSpectrum software:

- To access all the control signals for the DSP block, such as `sign A`, `sign B`, and `dynamic addnsub`, use the black box technique.
- While performing signed operations, ensure that the specified data width of the output port matches the data width of the expected result. Otherwise, the sign bit might be lost or data might be incorrect because the sign is not extended. For example, if the data widths of input `A` and `B` are `width_a` and `width_b`, respectively, the maximum data width of the result can be (`width_a + width_b +2`) for the four-multipliers adder mode. Thus, the data width of the output port should be less than or equal to (`width_a + width_b +2`).
- While using the accumulator, the data width of the output port should be equal to or greater than (`width_a + width_b`). The maximum width of the accumulator can be (`width_a + width_b + 16`). Accumulators wider than this are implemented in logic.

- If the design uses more multipliers than are available in a particular device, you might get a no fit error in the Quartus II software. In such cases, use the attribute settings in the LeonardoSpectrum software to control the mapping of multipliers in your design to DSP blocks or logic.

Block-Based Design with the Quartus II Software

The incremental compilation and LogicLock™ block-based design flows enable users to design, optimize, and lock down a design one section at a time. You can independently create and implement each logic module into a hierarchical or team-based design. With this method, you can preserve the performance of each module during system integration and have more control over placement of your design. To maximize the benefits of the incremental compilation or LogicLock design methodology in the Quartus II software, you can partition a new design into a hierarchy of netlist files during synthesis in the LeonardoSpectrum software.

You can create different netlist files with the LeonardoSpectrum software for different sections of a design hierarchy. When you have different netlist files, it means that each section is independent of the others. When synthesizing the entire project, only portions of a design that have been updated have to be re-synthesized when you compile the design. You can make changes, optimize, and re-synthesize your section of a design without affecting other sections.

For more information about incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about the LogicLock feature, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Hierarchy and Design Considerations

You must plan your design's structure and partitioning carefully to use incremental compilation and LogicLock features effectively. Optimal hierarchical design practices include partitioning the blocks at functional boundaries, registering the boundaries of each block, minimizing the I/O between each block, separating timing-critical blocks, and keeping the critical path within one hierarchical block.

For more recommendations for hierarchical design partitioning, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

To ensure the proper functioning of the synthesis tool, you can apply the LogicLock option in the LeonardoSpectrum software only to modules, entities, or netlist files. In addition, each module or entity should have its own design file. It is difficult to maintain incremental synthesis if two different modules are in the same design file (but are defined as being part of different regions), because both regions have to be recompiled when you change one of the modules or entities.

If you use boundary tri-states in a lower-level block, the LeonardoSpectrum software pushes (or "bubbles") the tri-states through the hierarchy to the top-level to take advantage of the tri-state drivers on the output pins of the Altera device. Because bubbling tri-states require optimizing through hierarchies, lower-level tri-states are not supported with a block-level design methodology. You should use tri-state drivers only at the external output pins of the device and in the top-level block in the hierarchy.

If the hierarchy is flattened during synthesis, logic is optimized across boundaries, preventing you from making LogicLock assignments to the flattened blocks. Altera recommends preserving the hierarchy when compiling the design. In the **Optimize** command of your script, use the **Hierarchy Preserve** command, or in the user interface, select **Preserve** in the **Hierarchy** section on the **Optimize** Flow tab.

If you are compiling your design with a script, you can use an alternative method for preventing optimization across boundaries. In this case, use the **Auto** hierarchy setting and set the `auto_dissolve` attribute to false on the instances or views that you want to preserve (that is, the modules with LogicLock assignments) using the following syntax:

```
set_attribute -name auto_dissolve -value false \
     .work.<block1>.INTERFACE
```

This alternative method flattens your design according to the `auto_dissolve` limits, but does not optimize across boundaries where you apply the attribute as described.

For more details about LeonardoSpectrum attributes and hierarchy levels, refer to the LeonardoSpectrum documentation in the Help menu.

Creating a Design with Multiple .edif Files

The first stage of a hierarchical design flow is to generate multiple **.edif** files, so that you can take advantage of the incremental compilation flows in the Quartus II software. If the whole design is in one **.edif** file, changes in one block affect other blocks because of possible node name changes. You can generate multiple **.edif** files either by using the LogicLock option in the LeonardoSpectrum software, or by manually using a black box methodology on each block that you want to be part of a LogicLock region.

After you have created multiple **.edif** files with one of these methods, you must create the appropriate Quartus II project(s) to place-and-route the design.

Generating Multiple .edif Files Using the LogicLock Option

This section describes how to generate multiple **.edif** files using the LogicLock option in the LeonardoSpectrum software.

When synthesizing a top-level design that includes LogicLock regions, perform the following general steps:

1. Read in the Verilog HDL or VHDL source files.
2. Add LogicLock constraints.
3. Optimize and write output netlist files, or choose **Run Flow**.

To set the correct constraints and compile the design, perform the following steps in the LeonardoSpectrum software:

1. On the Tools menu, switch to the **Advanced Flow** tab instead of the **Quick Setup** tab.
2. Set the target technology and speed grade for the device on the **Technology Flow** tab.
3. Open the input source files on the **Input Flow** tab.
4. Click **Read** on the **Input Flow** tab to read the source files but not begin optimization.
5. Select the **Module Power** tab located at the bottom of the **Constraints** Flow tab.
6. Click on a module to be placed in a LogicLock region in the **Modules** section.
7. Turn on the **LogicLock** option.
8. Type the desired LogicLock region name in the text field under the **LogicLock** option.
9. Click **Apply**.
10. Repeat steps 6-9 for any other modules that you want to place in LogicLock regions.

In some cases, you are prompted to save your LogicLock and other non-global constraints in a Constraints File (**.ctr**) when you click anywhere off the **Constraints Flow** tab. The default name is <*project name*>**.ctr**. This file is added to your **Input** file list, and must be manually included later if you recreate the project.

The command written into the LeonardoSpectrum Information or Transcript Window is the Tcl command that gets written into the **.ctr** file. The format of the "path" for the module specified in the command should be work.<*module*>**.INTERFACE**. To ensure that you don't see an optimized version of the module, do not perform a **Run Flow** on the **Quick Setup** tab prior to setting LogicLock constraints. Always use the **Read** command, as described in step 4.

11. Continue making any other settings as required on the **Constraints** tab.
12. Select **Preserve** in the **Hierarchy** section on the **Optimize** tab to ensure that the hierarchy names are not flattened during optimization.
13. Continue making any other settings as required on the **Optimize** tab.
14. Run your synthesis flow with each Flow tab, or click **Run Flow**.

Synthesis creates an **.edif** file for each module that has a LogicLock assignment in the **Constraints** Flow tab. You can now use these files with the incremental compilation flows in the Quartus II software.

You might occasionally see multiple **.edif** files and LogicLock commands for the same module. An “unfolded” version of a module is created when you instantiate a module more than once and the boundary conditions of the instances are different. For example, if you apply a constant to one instance of the block, it might be optimized to eliminate unneeded logic. In this case, the LeonardoSpectrum software must create a separate module for each instantiation (unfolding). If this unfolding occurs, you see more than one **.edif** file, and each EDIF file has a LogicLock assignment to the same LogicLock region. When you import the **.edif** files to the Quartus II software, the **.edif** files created from the module are placed in different LogicLock regions. Any optimizations performed in the Quartus II software using the LogicLock methodology must be performed separately for each **.edif** netlist file.

Creating a Quartus II Project for Multiple .edif Files Including LogicLock Regions

The LeonardoSpectrum software creates **.tcl** files that provide the Quartus II software with the appropriate LogicLock assignments, creating a region for each **.edif** file along with the information to set up a Quartus II project.

The **.tcl** file contains the commands shown in Example 12–5 for each LogicLock region. This example is for module taps where the name `taps_region` was typed as the LogicLock region name in the **Constraints** Flow tab in the LeonardoSpectrum software.

Example 12–5. Tcl File for Module Taps with taps_region as LogicLock Region Name

```
project add_assignment {taps} {taps_region} {} {}
    {LL_AUTO_SIZE} {ON}
project add_assignment {taps} {taps_region} {} {}
    {LL_STATE} {FLOATING}
project add_assignment {taps} {taps_region} {} {}
    {LL_MEMBER_OF} {taps_region}
```

These commands create a LogicLock region with Auto-Size and Floating-Origin properties. This flexible LogicLock region allows the Quartus II Compiler to select the size and location of the region.

For more information about Tcl commands, refer to the *TCL Scripting* chapter in volume 2 of the *Quartus II Handbook*.

You can use the following methods to import the **.edif** file and corresponding **.tcl** file into the Quartus II software:

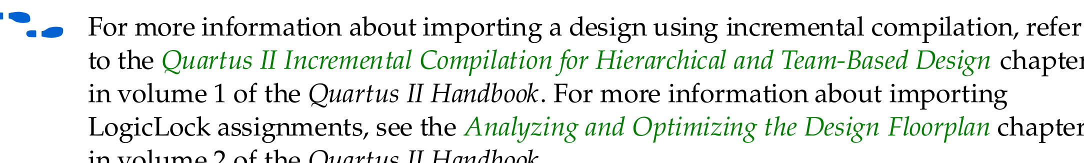
- Use the **.tcl** file that is created for each **.edif** file by the LeonardoSpectrum software. This method allows you to generate multiple Quartus II projects, one for each block in the design. Each designer in the project can optimize their block separately in the Quartus II software and preserve their results. Altera recommends this method for bottom-up incremental and hierarchical design methodologies because it allows each block in the design to be treated separately. Each block can be brought into one top-level project with the import function.

  or

- Use the <*top-level project*>**.tcl** file that contains the assignments for all blocks in the project. This method allows the top-level designer to import all the blocks into one Quartus II project. You can optimize all modules in the project at once in a top-down design flow. If additional optimization is required for individual blocks, each designer can use their **.edif** file to create a separate project at that time. You would then have to add new assignments to the top-level project using the import function.

In both methods, use the following steps to create the Quartus II project, import the appropriate LogicLock assignments, and compile the design:

1. Place the **.edif** and **.tcl** files in the same directory.
2. On the View menu, point to **Utility Windows** and click **Tcl Console** to open the **Quartus II Tcl Console**.
3. Type `source <path>/<project name>.tcl` ↵.
4. To open the new completed project, on the File menu, click **Open Project**. Browse to and select the project name, and click **Open**.



For more information about importing a design using incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about importing LogicLock assignments, see the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Generating Multiple .edif Files Using Black Boxes

This section describes how to manually generate multiple **.edif** files using the black box technique. The manual flow was supported in older versions of the LeonardoSpectrum software. The manual flow is discussed here because some designers want more control over the project for each submodule.

To create multiple **.edif** files in the LeonardoSpectrum software, create a separate project for each module and top-level design that you want to maintain as a separate **.edif** file. Implement black box instantiations of lower-level modules in your top-level project.

When synthesizing the projects for the lower-level modules and the top-level design, use the following general guidelines.

For lower-level modules:

- Turn off **Map IO Registers** for the target technology on the **Technology** Flow tab.
- Read the HDL files for the modules. Modules may include black box instantiations of lower-level modules that are also maintained as separate **.edif** files.
- Add constraints.
- Turn off **Add I/O Pads** on the **Optimize** Flow tab.

For the top-level design:

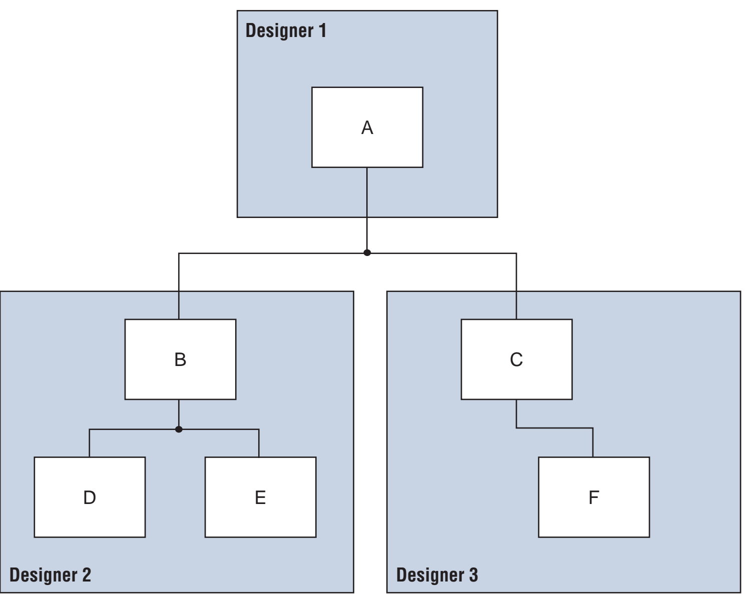
- Turn on **Map IO Registers** if you want to implement input and/or output registers in the IOEs for the target technology on the **Technology** Flow tab.

- Read the HDL files for the top-level design.
  - Black box lower-level modules in the top-level design.
- Add constraints (clock settings should be made at this time).

The following sections describe examples of black box modules in a block-based and team-based design flow.

In Figure 12–3, the top-level design A is assigned to one engineer (designer 1), while two-engineers work on the lower levels of the design. Designer 2 works on B and its submodules D and E, while designer 3 works on C and its submodule F.

Figure 12–3. Block-Based and Team-Based Design Example



One netlist is created for the top-level module A, another netlist is created for B and its submodules D and E, and another netlist is created for C and its submodule F. To create multiple **.edif** files, perform the following steps:

1. Generate an **.edif** file for module C. Use **C.v** and **F.v** as the source files.
2. Generate an **.edif** file for module B. Use **B.v**, **D.v**, and **E.v** as the source files.
3. Generate a top-level **.edif** file **A.v** for module A. Ensure that your black box modules B and C were optimized separately in steps 1 and 2.

Black Box Methodology in Verilog HDL

Any design block that is not defined in the project, or included in the list of files to be read for a project, is treated as a black box by the software. In Verilog HDL, you must also provide an empty module declaration for the module that you plan to treat as a black box.

Example 12–6 shows an example of the **A.v** top-level file. If any of your lower-level files also contain a black-boxed lower-level file in the next level of hierarchy, follow the same procedure.

Example 12–6. Verilog HDL Top-Level File Black Boxing Example

```
module A (data_in,clk,e,ld,data_out);
    input data_in, clk, e, ld;
    output [15:0] data_out;

    reg [15:0] cnt_out;
    reg [15:0] reg_a_out;

    B U1 ( .data_in (data_in),.clk (clk), .e(e), .ld (ld),
           .data_out(cnt_out) );

    C U2 ( .d(cnt_out), .clk (clk), .e(e), .q (reg_out));
    // Any other code in A.v goes here.

endmodule

// Empty Module Declarations of Sub-Blocks B and C follow here.
// These module declarations (including ports) are required for
blackboxing.

module B (data_in,e,ld,data_out );
    input data_in, clk, e, ld;
    output [15:0] data_out;
endmodule

module C (d,clk,e,q );
    input d, clk, e;
    output [15:0] q;
endmodule
```

Previous versions of the LeonardoSpectrum software required an attribute statement `//exemplar attribute U1 NOOPT TRUE`, which instructed the software to treat the instance `U1` as a black box. This attribute is no longer required, although it is still supported in the software.

Black Boxing in VHDL

Any design block that is not defined in the project, or included in the list of files to be read for a project, is treated as a black box by the software. In VHDL, a component declaration is required for the black box which is normal for any other block in the design.

Example 12–7 shows an example of the **A.vhd** top-level file. If any of your lower-level files also contain a black boxed lower-level file in the next level of hierarchy, follow the same procedure.

Example 12–7. VHDL Top-Level File Black Boxing Example

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY A IS
PORT ( data_in : IN INTEGER RANGE 0 TO 15;
    clk : IN STD_LOGIC;
    e : IN STD_LOGIC;
    ld : IN STD_LOGIC;
    data_out : OUT INTEGER RANGE 0 TO 15
);
END A;

ARCHITECTURE a_arch OF A IS

COMPONENT B PORT(
    data_in : IN INTEGER RANGE 0 TO 15;
    clk : IN STD_LOGIC;
    e : IN STD_LOGIC;
    ld : IN STD_LOGIC;
    data_out : OUT INTEGER RANGE 0 TO 15
);
END COMPONENT;

COMPONENT C PORT(
    d : IN INTEGER RANGE 0 TO 15;
    clk : IN STD_LOGIC;
    e : IN STD_LOGIC;
    q : OUT INTEGER RANGE 0 TO 15
);
END COMPONENT;

-- Other component declarations in A.vhd go here

signal cnt_out : INTEGER RANGE 0 TO 15;
signal reg_a_out : INTEGER RANGE 0 TO 15;
BEGIN
CNT : C
PORT MAP (
    data_in => data_in,
    clk => clk,
    e => e,
    ld => ld,
    data_out => cnt_out
);

REG_A : D
PORT MAP (
    d => cnt_out,
    clk => clk,
    e => e,
    q => reg_a_out
);

-- Any other code in A.vhd goes here

END a_arch;
```

Previous versions of the LeonardoSpectrum software required the attribute statement `noopt of C: component is TRUE`, which instructed the software to treat the component C as a black box. This attribute is no longer required, although it is still supported in the software.

After you have completed the steps outlined in this section, you have a different **.edif** netlist file for each block of code. You can now use these files for incremental compilation flows in the Quartus II software.

Creating a Quartus II Project for Multiple .edif Files

The LeonardoSpectrum software creates a **.tcl** file for each **.edif** file, which provides the Quartus II software with the information to set up a project.

As in the previous section, there are two different methods for bringing each **.edif** file and corresponding **.tcl** file into the Quartus II software:

- Use the **.tcl** file that is created for each **.edif** file by the LeonardoSpectrum software. This method generates multiple Quartus II projects, one for each block in the design. Each designer in the project can optimize their block separately in the Quartus II software and preserve their results. Designers should create a LogicLock region for each block; the top-level designer should then import all the blocks and assignments into the top-level project. Altera recommends this method for bottom-up incremental and hierarchical design methodology because it allows each block in the design to be treated separately; each block can be imported into one top-level project.

  or

- Use the <*top-level project*>**.tcl** file that contains the information to set up the top-level project. This method allows the top-level designer to create LogicLock regions for each block and bring all the blocks into one Quartus II project. Designers can optimize all modules in the project at once in a top-down design flow. If additional optimization is required for individual blocks, each designer can take their **.edif** file and create a separate Quartus II project at that time. New assignments would then have to be added to the top-level project manually or through the import function.

For more information about importing designs using incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about importing LogicLock regions, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

In both methods, use the following steps to create the Quartus II project and compile the design:

1. Place the **.edif** and **.tcl** files in the same directory.
2. On the View menu, point to **Utility Windows** and click **Tcl Console**. The Quartus II Tcl Console appears.
3. At a Tcl prompt, type `source <path>/<project name>.tcl` ↵.
4. On the File menu, click **Open Project**. In the New Project window, browse to and select the project name. Click **Open**.
5. To create LogicLock assignments, on the Assignments menu, click **LogicLock Regions Window**.
6. On the Processing menu, click **Start Compilation**.

Incremental Synthesis Flow

If you make changes to one or more submodules, you can manually create new projects in the LeonardoSpectrum software to generate a new **.edif** netlist file when there are changes to the source files. Alternatively, you can use incremental synthesis to generate a new netlist for the changed submodule(s). To perform incremental synthesis in the LeonardoSpectrum software, use the script described in this section to reoptimize and generate a new **.edif** netlist file for only the affected modules using the LeonardoSpectrum top-level project. This method applies only when you are using the **LogicLock** option in the LeonardoSpectrum software.

Modifications Required for the LogicLock_Incremental.tcl Script File

There are three sets of entries in the file that must be modified before beginning incremental synthesis. The variables in the **.tcl** file are surrounded by angle brackets (< >).

1. Add the list of source files that are included in the project. You can enter the full path to the file or just the file name if the files are located in the working directory.
2. Indicate which modules in the design have changed. These modules are the **.edif** files that are regenerated by the LeonardoSpectrum software. These modules contain a LogicLock assignment in the original compilation.

   ☞ Obtain the LeonardoSpectrum software path for each module by looking at the **.ctr** file that contains the LogicLock assignments from the original project. Each LogicLock assignment is applied to a particular module in the design.

3. Enter the target device family using the appropriate device keyword. The device keyword is written into the Transcript or Information window when you select a target Technology and click **Load Library** or **Apply** on the **Technology** Flow tab in the graphical user interface.

Example 12–8 shows the **LogicLock_Incremental.tcl** file for the incremental synthesis flow. You must modify the **.tcl** file before you can use it for your project.

Example 12–8. LogicLock_Interface.tcl Script File for Incremental Synthesis

```
##############################################
#### LogicLock Incremental Synthesis Flow ####
##############################################

## You must indicate which modules have changed (based on the source files
## that have changed) and provide the complete path to each module

## You must also specify the list of design files and the target Altera
## technology being used

# Read the design source files.
read <list of design files separated by spaces (such as block1.v block2.v)>

# Get the list of modified modules in bottom-up "depth first search" order
# where the lower-level blocks are listed first (these should be modules
# that had LogicLock assignments and separate EDIF netlist files in the
# first pass and had their source code modified)

set list_of_modified_modules {.work.<block2>.INTERFACE .work.<block1>.INTERFACE}

foreach module $list_of_modified_modules {
   set err_rc [regexp {\.(.*)\.(.*)\.(.*)} $module unused lib module_name arch]
   present_design $module

   # Run optimization, preserving hierarchy. You must specify a technology.
   optimize -ta <technology> -hierarchy preserve

   # Ensure that the lower-level module is not optimized again when
   # optimizing higher-level modules.
   dont_touch $module
}

foreach module $list_of_modified_modules {
   set err_rc [regexp {\.(.*)\.(.*)\.(.*)} $module unused lib module_name arch]
   present_design $module
   undont_touch $module
   auto_write $module_name.edf
   # Ensure that the lower-level module is not written out in the EDIF file
   # of the higher-level module.
   noopt $module
}
```

Running the Tcl Script File in LeonardoSpectrum

When you have modified the Tcl script, as described in "Modifications Required for the LogicLock_Incremental.tcl Script File" on page 12–24, you can compile your design using the script.

You can run the script in batch mode at the command line prompt using the following command:

spectrum -file <*Tcl_file*> ↵

To run the script from the interface, on the File menu, click **Run Script**, then browse to your **.tcl** file and click **Open**.

The LogicLock incremental design flow uses module-based design to help you preserve performance of modules and have control over placement. By tagging the modules that require separate **.edif** files, you can make multiple **.edif** files for use with the Quartus II software from a single LeonardoSpectrum software project.

Conclusion

Advanced synthesis is an important part of the design flow. Taking advantage of the Mentor Graphics LeonardoSpectrum software and the Quartus II design flow allows you to control how your design files are prepared for the Quartus II place-and-route process, as well as to improve performance and optimize a design for use with Altera devices. The methodologies outlined in this chapter can help optimize a design to achieve performance goals and save design time.

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*
- *LeonardoSpectrum Installation Guide* and the *LeonardoSpectrum User's Manual*.
- *Mentor Graphics Precision RTL Synthesis Support* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 12–6 shows the revision history of this chapter.

Table 12–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ No change to content. ■ Chapter 12 was previously Chapter 11 in software release 8.1.	Updated for the Quartus II 9.0 software release.
November 2008 v8.1.0	■ Changed to 8-1/2" x 11" page size. ■ Updated Table 12–3.	Updated for the Quartus II 8.1 software release.
May 2008 v8.0.0	Updated date and part number and added hypertext links.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

 © March 2009 Altera Corporation

13. Analyzing Designs with Quartus II Netlist Viewers

QII51013-9.0.0

Introduction

As FPGA designs grow in size and complexity, the ability to analyze how your synthesis tool interprets your design becomes critical. Often, with today's advanced designs, several design engineers are involved in coding and synthesizing different design blocks, making it difficult to analyze and debug the design. The Quartus® II RTL Viewer, State Machine Viewer, and Technology Map Viewer provide powerful ways to view your initial and fully mapped synthesis results during the debugging, optimization, or constraint entry process.

The first section in this chapter, "When to Use Viewers: Analyzing Design Problems", describes examples of using the viewers to analyze your design at various stages of the design cycle. The sections following this provide an introduction to the Quartus II design flow using netlist viewers, an overview of each viewer, and an explanation of the user interface. These sections describe the following tasks:

- How to navigate and filter schematics
- How to probe to and from other windows within the Quartus II software
- How to view a timing path from the Timing Analyzer report

This chapter contains the following sections regarding netlist viewers:

- "Introduction to the User Interface" on page 13–6
- "Navigating the Schematic View" on page 13–18
- "Customizing the Schematic Display in the RTL Viewer" on page 13–29
- "Filtering in the Schematic View" on page 13–29
- "Probing to Source Design File and Other Quartus II Windows" on page 13–35
- "Probing to the Viewers from Other Quartus II Windows" on page 13–37
- "Viewing a Timing Path" on page 13–38
- "Other Features in the Schematic Viewer" on page 13–39
- "Debugging HDL Code with the State Machine Viewer" on page 13–46

The final section provides a detailed example that uses the viewer to analyze a design and quickly resolve a design problem.

When to Use Viewers: Analyzing Design Problems

You can use netlist viewers to analyze your design to determine how it was interpreted by the Quartus II software. This section provides simple examples of how to use the RTL Viewer, State Machine Viewer, and Technology Map Viewer to analyze problems encountered in the design process.

The following sections contain information about how netlist viewers display your design:

- "Quartus II Design Flow with Netlist Viewers" on page 13–3
- "RTL Viewer Overview" on page 13–4
- "State Machine Viewer Overview" on page 13–5
- "Technology Map Viewer Overview" on page 13–5

Using the RTL Viewer is a good way to view your initial synthesis results to determine whether you have created the desired logic, and that the logic and connections have been interpreted correctly by the software. You can use the RTL Viewer and State Machine Viewer to check your design visually before simulation or other verification processes. Catching design errors at this early stage of the design process can save you valuable time.

If you see unexpected behavior during verification, use the RTL Viewer to trace through the netlist and ensure that the connections and logic in your design are as expected. You can also use the State Machine Viewer to view state machine transitions and transition equations. Viewing the design can help you find and analyze the source of design problems. If your design looks correct in the RTL Viewer, you know to focus your analysis on later stages of the design process and investigate potential timing violations or issues in the verification flow itself.

You can use the Technology Map Viewer to look at the results at the end of synthesis and technology mapping by running the viewer after performing Analysis and Synthesis. If you have compiled your design through the Fitter stage, you can view your post-mapping netlist in the Technology Map Viewer (Post-Mapping) and your post-fitting netlist in the Technology Map Viewer. If you perform only Analysis and Synthesis, both viewers display the same post-mapping netlist.

In addition, you can use the RTL Viewer or Technology Map Viewer to locate the source of a particular signal, which can help you debug your design. Use the navigation techniques described in this chapter to search easily through the design. You can trace back from a point of interest to find the source of the signal and ensure the connections are as expected.

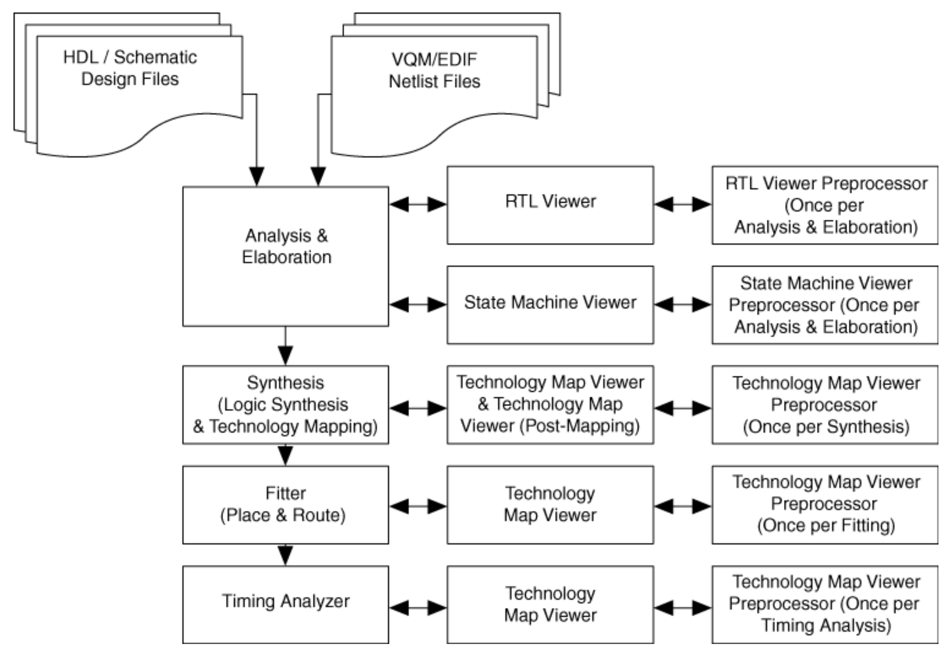
You can also use the Technology Map Viewer to help you locate post-synthesis nodes in your netlist and make assignments when optimizing your design. This functionality is useful, for example, when making a multicycle clock timing assignment between two registers in your design. Start at an I/O port and trace forward or backward through the design and through levels of hierarchy to find nodes that interest you, or locate a specific register by visually inspecting the schematic.

You can use the RTL Viewer, State Machine Viewer, and Technology Map Viewer in many other ways throughout the design, debugging, and optimization stages. Viewing the design netlist is a powerful way to analyze design problems. This chapter shows you how to use the various features of the netlist viewers to increase your productivity when analyzing a design.

Quartus II Design Flow with Netlist Viewers

The first time you open one of the netlist viewers after compiling the design, a preprocessor stage runs automatically before the viewer opens. If you close the viewer and open it again later without recompiling the design, the viewer opens immediately without performing the preprocessing stage. Figure 13–1 shows how the netlist viewers fit into the basic Quartus II design flow.

Figure 13–1. Quartus II Design Flow Including the RTL Viewer and Technology Map Viewer



To use a viewer, and before the viewer can run the preprocessor and open the design, compile your design with the following minimum compilation:

- To open the RTL Viewer or State Machine Viewer, first perform Analysis and Elaboration.
- To open the Technology Map Viewer or the Technology Map Viewer (Post-Mapping), first perform Analysis and Synthesis.

If you open one of the viewers without first compiling the design with the appropriate minimum compilation stage, the viewer does not appear. Instead, the Quartus II software issues an error message instructing you to run the necessary compilation stage and restart the viewer.

Both viewers display the results of the last successful compilation. Therefore, if you make a design change that causes an error during Analysis and Elaboration, you cannot view the netlist for the new design files, but you can still see the results from the last successfully compiled version of the design files. If you receive an error during compilation and you have not yet successfully run the appropriate compilation stage for your project, the viewer cannot be displayed; in this case, the Quartus II software issues an error message when you try to open the viewer.

If the viewer window is open when you start a new compilation, the viewer closes automatically. You must open the viewer again to view the new design netlist after compilation completes successfully.

RTL Viewer Overview

The Quartus II RTL Viewer allows you to view a register transfer level (RTL) graphical representation of your Quartus II integrated synthesis results or your third-party netlist file within the Quartus II software.

You can view results after Analysis and Elaboration when your design uses any supported Quartus II design entry method, including Verilog HDL Design Files (**.v**), SystemVerilog Design Files (**.sv**), VHDL Design Files (**.vhd**), AHDL Text Design Files (**.tdf**), schematic Block Design Files (**.bdf),** or schematic Graphic Design Files (**.gdf)** imported from the MAX+PLUS® II software. You can also view the hierarchy of atom primitives (such as device logic cells and I/O ports) when your design uses a synthesis tool to generate a Verilog Quartus Mapping File (**.vqm**) or Electronic Design Interchange Format (.**edf**) netlist file. Refer to Figure 13–1 for a flow diagram.

The Quartus II RTL Viewer displays a schematic view of the design netlist after Analysis and Elaboration or netlist extraction is performed by the Quartus II software, but before technology mapping and any synthesis or fitter optimization algorithms occur. This view is not the final design structure because optimizations have not yet occurred. This view most closely represents your original source design. If you synthesized your design using the Quartus II integrated synthesis, this view shows how the Quartus II software interpreted your design files. If you are using a third-party synthesis tool, this view shows the netlist written by your synthesis tool.

When displaying your design, the RTL Viewer optimizes the netlist to maximize readability in the following ways:

- Logic with no fan-out (its outputs are unconnected) and logic with no fan-in (its inputs are unconnected) are removed from the display.
- Default connections such as V_{CC} and GND are not shown.
- Pins, nets, wires, module ports, and certain logic are grouped into buses where appropriate.
- Constant bus connections are grouped.
- Values are displayed in hexadecimal format.
- NOT gates are converted to bubble inversion symbols in the schematic.
- Chains of equivalent combinational gates are merged into a single gate. For example, a 2-input AND gate feeding a 2-input AND gate is converted to a single 3-input AND gate.
- State machine logic is converted into a state diagram, state transition table, and state encoding table, which are displayed in the State Machine Viewer.

To run the RTL Viewer for a Quartus II project, first analyze the design to generate an RTL netlist. To analyze the design and generate an RTL netlist, on the Processing menu, point to **Start** and click **Start Analysis & Elaboration**. You can also perform a full compilation on any process that includes the initial Analysis and Elaboration stage of the Quartus II compilation flow.

To run the RTL Viewer, on the Tools menu, point to **Netlist Viewers** and click **RTL Viewer**.

You can set the RTL Viewer preprocessing to run during a full compilation, which allows you to launch the RTL Viewer after Analysis and Synthesis has completed, but while the Fitter is still running. In this case, you do not have to wait for the Fitter to finish before viewing the schematic. This technique is useful for a large design that requires a substantial amount of time in the place-and-route stage.

To set the RTL Viewer preprocessing to run during compilation, on the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Process Settings** and turn on **Run RTL Viewer preprocessing during compilation**. By default, this option is turned off.

State Machine Viewer Overview

The State Machine Viewer presents a high-level view of finite state machines in your design. The State Machine Viewer provides a graphical representation of the states and their related transitions, as well as a state transition table that displays the condition equation for each of the state transitions, and encoding information for each state.

To run the State Machine Viewer, on the Tools menu, point to **Netlist Viewers** and click **State Machine Viewer**. To open the State Machine Viewer for a particular state machine, double-click the state machine instance in the RTL Viewer or right-click the state machine instance and click **Hierarchy Down**.

Technology Map Viewer Overview

The Quartus II Technology Map Viewer provides a technology-specific, graphical representation of your design after Analysis and Synthesis or after the Fitter has mapped your design into the target device. The Technology Map Viewer shows the hierarchy of atom primitives (such as device logic cells and I/O ports) in your design. For supported families, you can also view internal registers and look-up tables (LUTs) inside logic cells (LCELLs) and registers in I/O atom primitives. Refer to "Viewing Contents of Atom Primitives" on page 13–19 for details.

Where possible, the port names of each hierarchy are maintained throughout synthesis. However, port names might change or be removed from the design. For example, if a port is unconnected or driven by GND or V_{CC}, it is removed during synthesis. When a port name is changed, the port is assigned a related user logic name in the design or a generic port name such as `IN1` or `OUT1`.

You can view your Quartus II technology-mapped results after synthesis, fitting, or timing analysis. To run the Technology Map Viewer for a Quartus II project, on the Processing menu, point to **Start** and click **Start Analysis & Synthesis** to synthesize and map the design to the target technology. At this stage, the Technology Map Viewer shows the same post-mapping netlist as does the Technology Map Viewer (Post-Mapping). You can also perform a full compilation, or any process that includes the synthesis stage in the compilation flow.

If you have completed the Fitter stage, the Technology Map Viewer shows the changes made to your netlist by the Fitter, such as physical synthesis optimizations, while the Technology Map Viewer (Post-Mapping) shows the post-mapping netlist. If you have completed the Timing Analysis stage, you can locate timing paths from the Timing Analyzer report in the Technology Map Viewer (refer to "Viewing a Timing Path" on page 13–38 for details). Refer to Figure 13–1 on page 13–3 for a flow diagram.

To run the Technology Map Viewer, on the Tools menu, point to **Netlist Viewers** and click **Technology Map Viewer**, or select **Technology Map Viewer** from the Applications toolbar.

To run the Technology Map Viewer (Post-Mapping), on the Tools menu, point to **Netlist Viewers** and click **Technology Map Viewer (Post-Mapping)**.

Introduction to the User Interface

The RTL Viewer window and Technology Map Viewer window each consist of two main parts: the schematic view and the hierarchy list. Figure 13–2 shows the RTL Viewer window and indicates these two parts. Both viewers also contain a toolbar that provides tools to use in the schematic view. In the Quartus II software version 9.0, the toolbar contains a **Hierarchy List** button. This tool enables you to refine your searches. For more information, refer to "Find Command" on page 13–44.

You can have only one RTL Viewer, one Technology Map Viewer, one Technology Map Viewer (Post-Mapping), and one State Machine Viewer window open at the same time, although each window can show multiple pages. For example, you cannot have two RTL Viewer windows open at the same time. The viewer window has characteristics similar to other "child" windows in the Quartus II software; it can be resized and moved, minimized or maximized, tiled or cascaded, and moved in front of or behind other windows.

You can detach the window and move it outside the Quartus II main interface. To detach a window, click the **Detach Window** icon on the toolbar, or, on the Window menu, click **Detach Window**. To attach the detached window back to the Quartus II main interface, click the **Attach Window** icon on the toolbar, or, on the Window menu, click **Attach Window**.

Figure 13–2. RTL Viewer Window and RTL Toolbar

Schematic View

The schematic view is shown on the right side of the RTL Viewer and Technology Map Viewer. It contains a schematic representing the design logic in the netlist. This view is the main screen for viewing your gate-level netlist in the RTL Viewer and your technology-mapped netlist in the Technology Map Viewer.

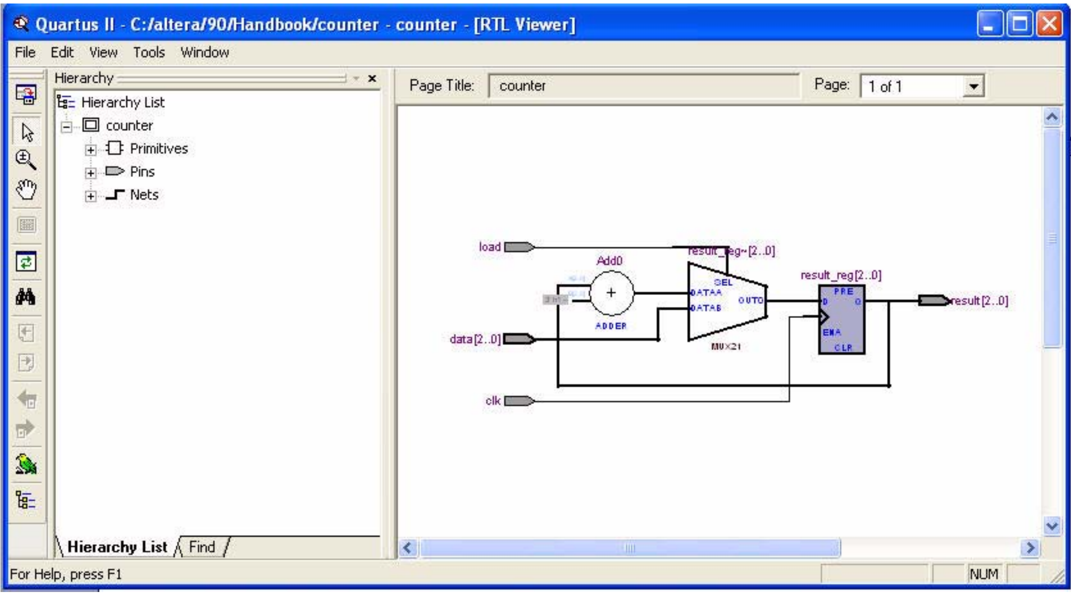
Schematic Symbols

The symbols for nodes in the schematic represent elements of your design netlist. These elements include input and output ports, registers, logic gates, Altera® primitives, high-level operators, and hierarchical instances.

Figure 13–3 shows an example of an RTL Viewer schematic for a 3-bit synchronous loadable counter. Example 13–1 shows the Verilog HDL code that produced this schematic. This example includes multiplexers and a group of registers (Table 13–1) in a bus along with an ADDER operator (Table 13–3 on page 13–12) inferred by the counting function in the HDL code.

The schematic in Figure 13–3 displays wire connections between nodes with a thin black line and bus connections with a thick black line.

Figure 13–3. Example Schematic Diagram in the RTL Viewer



Example 13–1. Code Sample for Counter Schematic Shown in Figure 13–3

```
module counter (input [2:0] data, input clk, input load, output [2:0]
result);
   reg [2:0] result_reg;
   always @ (posedge clk)
      if (load)
         result_reg <= data;
      else
         result_reg <= result_reg + 1;
   assign result = result_reg;
endmodule
```

Figure 13–4 shows a portion of the corresponding Technology Map Viewer schematic with a compiled design that targets a Stratix® device. In this schematic, you can see the LCELL (logic cell) device-specific primitives that represent the counter function, labeled with their post-synthesis node names. The `REGOUT` port represents the output of the register in the LCELL; the `COMBOUT` port represents the output of the combinational logic in the LUT of the LCELL. The hexadecimal number in parentheses below each LCELL primitive represents the LUT mask, which is a hexadecimal representation of the logic function of the LCELL.

Figure 13–4. Example Schematic Diagram in the Technology Map Viewer

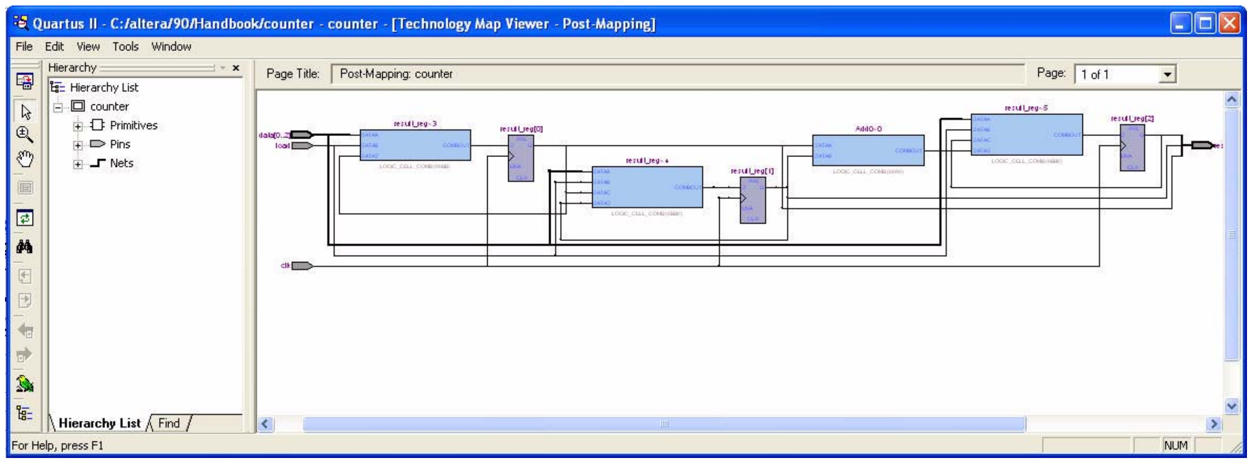


Table 13–1 lists and describes the primitives and basic symbols that you can display in the schematic view of the RTL Viewer and Technology Map Viewer. Table 13–3 on page 13–12 lists and describes the additional higher-level operator symbols used in the RTL Viewer schematic view.

The logic gates and operator primitives appear only in the RTL Viewer. Logic in the Technology Map Viewer is represented by atom primitives, such as registers and LCELLs.

Table 13–1. Symbols in the Schematic View (Part 1 of 3)

Symbol	Description
I/O Ports	An input, output, or bidirectional port in the current level of hierarchy. A device input, output, or bidirectional pin when viewing the top-level hierarchy. The symbol can also represent a bus. Only one wire is shown connected to the bidirectional symbol, representing both the input and output paths. Input symbols appear on the left-most side of the schematic. Output and bidirectional symbols appear on the right-most side of the schematic.
I/O Connectors	An input or output connector, representing a net that comes from another page of the same hierarchy (refer to "Partitioning the Schematic into Pages" on page 13–26). To go to the page that contains the source or the destination, right-click on the net and choose the page from the menu (refer to "Following Nets Across Schematic Pages" on page 13–27).
Hierarchy Port Connector	A connector representing a port relationship between two different hierarchies. A connector indicates that a path passes through a port connector in a different level of hierarchy.
OR, AND, XOR Gates	An OR, AND, or XOR gate primitive (the number of ports can vary). A small circle (bubble symbol) on an input or output port indicates the port is inverted.

Table 13–1. Symbols in the Schematic View (Part 2 of 3)

Symbol	Description
MUX	A multiplexer (MUX) primitive with a selector port that selects between port 0 and port 1. A MUX with more than two inputs is displayed as an operator (refer to "Operator Symbols in the RTL Viewer Schematic View" on page 13–12).
BUFFER	A buffer primitive. The figure shows the tri-state buffer, with an inverted output enable port. Other buffers without an enable port include LCELL, SOFT, CARRY, and GLOBAL. The NOT gate and EXP expander buffers use this symbol without an enable port and with an inverted output port.
CARRY_SUM	A CARRY_SUM buffer primitive with the following ports: ■ SI – SUM IN ■ SO – SUM OUT ■ CI – CARRY IN ■ CO – CARRY OUT
LATCH	A latch primitive with the following ports: ■ D – data input ■ ENA – enable input ■ Q – data output ■ PRE – preset ■ CLR – clear
DFFE/DFFEA/DFFAE	A DFFE (data flipflop with enable) primitive, with the same ports as a latch and a clock trigger. The other flipflop primitives are similar: ■ DFFEA (data flipflop with enable and asynchronous load) primitive with additional ALOAD asynchronous load and ADATA data signals ■ DFFEAS (data flipflop with enable and both synchronous and asynchronous load), which has ASDATA as the secondary data port
Atom Primitive	Primitives are low-level nodes that cannot be expanded to any lower hierarchy. The symbol displays the port names, the primitive type, and its name. The blue shading indicates an atom primitive in the Technology Map Viewer that allows you to view the internal details of the primitive. Refer to "Viewing Contents of Atom Primitives" on page 13–19 for details.
Other Primitive	Any primitive that does not fall into the categories above. Primitives are low-level nodes that cannot be expanded to any lower hierarchy. The symbol displays the port names, the primitive or operator type, and its name. The figure shows an LCELL WYSIWYG primitive, with DATAA to DATAD and COMBOUT port connections. This type of LCELL primitive is found in the Technology Map Viewer for technology-specific atom primitives when the contents of the atom primitive cannot be viewed. The RTL Viewer contains similar primitives if the source design is a VQM or EDIF netlist.

Table 13–1. Symbols in the Schematic View (Part 3 of 3)

Symbol	Description
Instance	An instance in the design that does not correspond to a primitive or operator (generally a user-defined hierarchy block), indicated by the double outline and green shading. The symbol displays the instance name. To open the schematic for the lower-level hierarchy, right-click and choose the appropriate command (refer to "Traversing and Viewing the Design Hierarchy" on page 13–18).
Encrypted Instance	A user-defined encrypted instance in the design, indicated by the double outline and gray shading. The symbol displays the instance name. You cannot open the schematic for the lower level hierarchy, because the source design is encrypted.
State Machine Instance	A finite state machine instance in the design, indicated by the double outline and yellow shading. Double-clicking this instance opens the State Machine Viewer. Refer to "State Machine Viewer" on page 13–16 for more details.
RAM	A synchronous memory instance with registered inputs and optionally registered outputs, indicated by purple shading. The symbol shows the device family and the type of TriMatrix memory block. This figure shows a true dual-port memory block in a Stratix M-RAM block.
Logic Cloud	A logic cloud is a group of combinational logic, indicated by a cloud symbol. Refer to "Grouping Combinational Logic into Logic Clouds" on page 13–22 for more details.
Constant 16' d129 -- 16' h0081 -- 16' b10000001 --	A constant signal value that is highlighted in gray and displayed in hexadecimal format by default throughout the schematic. To change the format, refer to "Changing the Constant Signal Value Formatting" on page 13–24.

Table 13–2 lists and describes the symbol used only in the State Machine Viewer.

Table 13–2. Symbol Available Only in the State Machine Viewer

Symbol	Description
State Node tap1	The node representing a state in a finite state machine. State transitions are indicated with arcs between state nodes. The double circle border indicates the state connects to logic outside the state machine, while a single circle border indicates the state node does not feed outside logic.

Table 13–3 lists and describes the additional higher level operator symbols used in the RTL Viewer schematic view.

Table 13–3. Operator Symbols in the RTL Viewer Schematic View (Part 1 of 2)

Symbol	Description
Add0 A[7..0] B[7..0] + ADDER	An adder operator: `OUT = A + B`
Mult0 A[7..0] B[7..0] X MULTIPLIER	A multiplier operator: `OUT = A ¥ B`
Div0 A[7..0] B[7..0] / DIVIDER	A divider operator: `OUT = A / B`
Equal0 A[7..0] B[7..0] = EQUAL	Equals
ShiftLeft0 A[7..0] COUNT[7..0] << LEFT_SHIFT	A left shift operator: `OUT = (A << COUNT)`
ShiftRight0 A[7..0] COUNT[7..0] >> RIGHT_SHIFT	A right shift operator: `OUT = (A >> COUNT)`

Table 13–3. Operator Symbols in the RTL Viewer Schematic View (Part 2 of 2)

Symbol	Description
Mod0 A[7..0] B[7..0] % MODULO	A modulo operator: `OUT = (A % B)`
LessThan0 A[7..0] B[7..0] < LESS_THAN	A less than comparator: `OUT = (A < B : A > B)`
Mux~0 SEL[2..0] DATA[7..0] OUT MUX	A multiplexer: `OUT = DATA [SEL]` The data range size is $2^{\text{sel range size}}$
Select~0 SEL[3..0] DATA[3..0] OUT SELECTOR	A selector: A multiplexer with one-hot select input and more than two input signals
Decoder~1 IN[1..0] OUT[3..0] DECODER	A binary number decoder: `OUT` = (binary_number (`IN`) == x) for x = 0 to x = $2^{(n+1)} - 1$

Selecting an Item in the Schematic View

To select an item in the schematic view, ensure that the Selection Tool is enabled in the viewer toolbar (this tool is enabled by default). Click on an item in the schematic view to highlight it in red.

Select multiple items by pressing the **Shift** or **Ctrl** key while selecting with your mouse. You can also select all nodes in a region by selecting a rectangular box area with your mouse cursor when the Selection Tool is enabled. To select nodes in a box, move your mouse to one corner of the area you want to select, click the mouse button, and drag the mouse to the opposite corner of the box, then release the mouse button. By default, creating a box like this highlights and selects all nodes in the selected area (instances, primitives, and pins), but not the nets. The **Viewer Options** dialog box provides an option to select nets. To include nets, right-click in the schematic and click **Viewer Options**. In the **Net Selection** section, turn on the **Select entire net when segment is selected** option.

Items selected in the schematic view are automatically selected in the hierarchy list (refer to the "Hierarchy List" on page 13–14). The list expands automatically if required to show the selected entry. However, the list does not collapse automatically when entries are not being used or are deselected.

When you select a hierarchy box, node, or port in the schematic view, the item is highlighted in red but none of the connecting nets are highlighted. When you select a net (wire or bus) in the schematic view, all connected nets are highlighted in red. The selected nets are highlighted across all hierarchy levels and pages. Net selection can be useful when navigating a netlist because you see the net highlighted when you traverse between hierarchy levels or pages.

In some cases, when you select a net that connects to nets in other levels of the hierarchy, these connected nets also are highlighted in the current hierarchy. If you prefer that these nets not be highlighted, use the **Viewer Options** dialog box option to highlight a net only if the net is in the current hierarchy. Right-click in the schematic and click **Viewer Options**. In the **Net Selection** section, turn on the **Limit selections to current hierarchy** option.

Moving and Panning in the Schematic View

When the schematic view page is larger than the portion currently displayed, you can use the scroll bars at the bottom and right side of the schematic view to see other areas of the page.

You can also use the Hand Tool to "grab" the schematic page and drag it in any direction. Enable the Hand Tool with the toolbar button. Click and drag to move around the schematic view without using the scroll bars.

In addition to the scroll bars and Hand Tool, you can use the middle-mouse/wheel button to move and pan in the schematic view. Click the middle-mouse/wheel button once to enable the feature. Move the mouse or scroll the wheel to move around the schematic view. Click the middle-mouse/wheel button again to turn the feature off.

Hierarchy List

The hierarchy list is displayed on the left side of the viewer window. The hierarchy list displays the entire netlist in a tree format based on the hierarchical levels of the design. Within each level, similar elements are grouped into sub-categories. Using the hierarchy list, traverse through the design hierarchy to view the logic schematic for each level. You can also select an element in the hierarchy list to be highlighted in the schematic view.

Nodes inside atom primitives are not listed in the hierarchy list.

For each module in the design hierarchy, the hierarchy list displays the applicable elements listed in Table 13–4. Click the "+" icon to expand an element.

Table 13–4. Hierarchy List Elements (Part 1 of 2)

Elements	Description
Instances	Modules or instances in the design that can be expanded to lower hierarchy levels.
State Machines	State machine instances in the design that can be viewed in the State Machine Viewer.

Table 13–4. Hierarchy List Elements (Part 2 of 2)

Elements	Description
Primitives	Low-level nodes that cannot be expanded to any lower hierarchy level. These include: ■ Registers and gates that you can view in the RTL Viewer when using Quartus II integrated synthesis ■ Logic cell atoms in the Technology Map Viewer or in the RTL Viewer when using a VQM or EDIF from third-party synthesis software In the Technology Map Viewer, you can view the internal implementation of certain atom primitives, but you cannot traverse into a lower level of hierarchy.
Pins	The I/O ports in the current level of hierarchy. ■ Pins are device I/O pins when viewing the top hierarchy level and are I/O ports of the design when viewing the lower levels. ■ When a pin represents a bus or an array of pins, expand the pin entry in the list view to see individual pin names.
Nets	Nets or wires connecting the nodes. When a net represents a bus or array of nets, expand the net entry in the tree to see individual net names.
Logic Clouds	A group of related combinational logics of a particular source. You can automatically or manually group combinational logics or ungroup logic clouds in your design.

Selecting an Item in the Hierarchy List

When you click any item in the hierarchy list, the viewer performs the following actions:

- Searches for the item in the currently viewed pages and displays the page containing the selected item in the schematic view if it is not currently displayed. (If you are currently viewing a filtered netlist, for example, the relevant page within the filtered netlist is displayed.)
- If the selected item is not found in the currently viewed pages, the entire design netlist is searched and the item is displayed in a default view.
- Highlights the selected item in red in the schematic view.

When you double-click an instance in the hierarchy list, the viewer displays the underlying implementation of the instance.

You can select multiple items by pressing the **Shift** or **Ctrl** key while selecting with your mouse. When you right-click an item in the hierarchy list, you can navigate in the schematic view using the **Filter** and **Locate** commands. Refer to "Filtering in the Schematic View" on page 13–29 and "Probing to Source Design File and Other Quartus II Windows" on page 13–35 for more information.

Enable or Disable the Auto Hierarchy List

When you select any node or net in a schematic, the hierarchy list is expanded automatically to show the selected node or net in the list. This allows you to easily identify the node or net when you have a complex schematic. By default, this option is disabled.

To enable the auto hierarchy list option, perform the following steps:

1. On the Tools menu, click **Options**.
2. In the **Options** dialog box, click **Netlist Viewers** under **Category**.

3. Turn on the **Enable Auto Hierarchy Expansion** option.
4. Click **OK**.

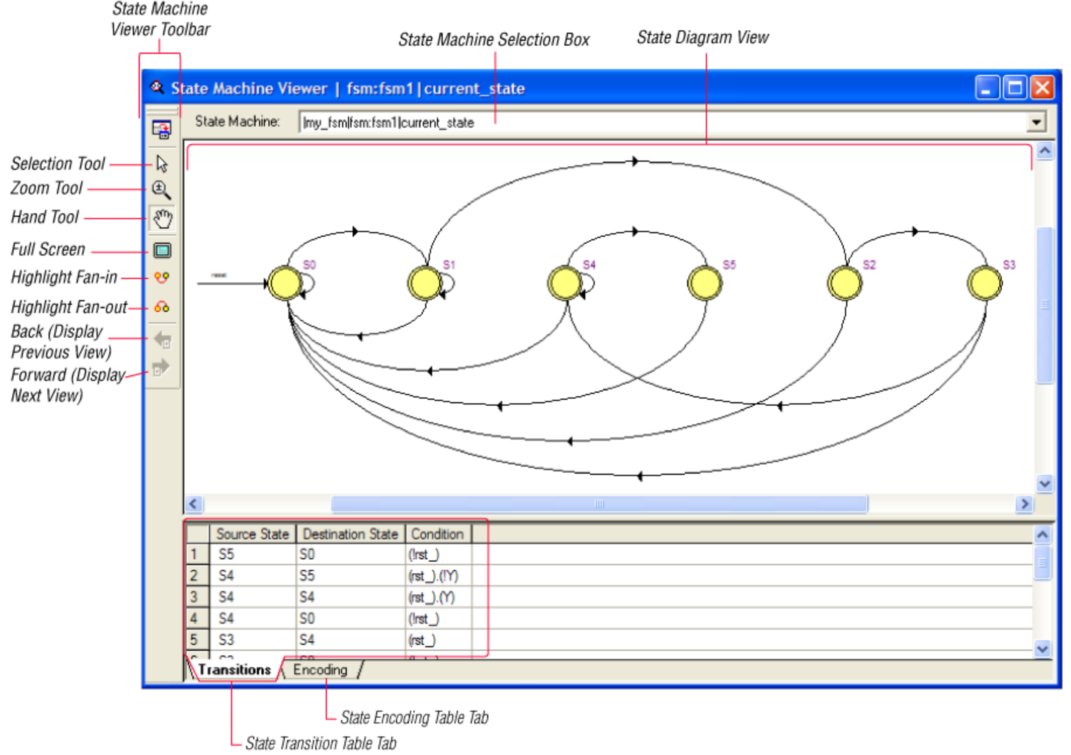
State Machine Viewer

The State Machine Viewer displays a graphical representation of the state machines in your design. You can open the State Machine Viewer in any of the following ways:

- On the Tools menu, point to **Netlist Viewers** and click **State Machine Viewer**
- Double-click on a state machine instance in the RTL Viewer
- Right-click on a state machine instance in the RTL Viewer and click **Hierarchy Down**
- Select a state machine instance in the RTL Viewer, and on the Project menu, point to **Hierarchy** and click **Down**

Figure 13–5 shows an example of the State Machine Viewer for a simple state machine. The State Machine toolbar on the left side of the viewer provides tools you can use in the state diagram view.

Figure 13–5. State Machine in the State Machine Viewer



State Diagram View

The state diagram view is shown at the top of the State Machine Viewer window. It contains a diagram of the states and state transitions.

The nodes that represent each state are arranged horizontally in the state diagram view with the initial state (the state node that receives the reset signal) in the left-most position. Nodes that connect to logic outside of the state machine instance are represented by a double circle. The state transition is represented by an arc with an arrow pointing in the direction of the transition.

When you select a node in the state diagram view, if you turn on the **Highlight Fan-in** or **Highlight Fan-out** command from the View menu or the State Machine Viewer toolbar, the respective fan-in or fan-out transitions from the node are highlighted in red.

An encrypted block with a state machine displays encoding information in the state encoding table, but does not display a state transition diagram or table.

State Transition Table

The state transition table on the Transitions tab at the bottom of the State Machine Viewer window displays the condition equation for each state transition. Each transition (each arc in the state diagram view) is represented by a row in the table. The table has the following three columns:

- **Source State**—the name of the source state for the transition
- **Destination State**—the name of the destination state for the transition
- **Condition**—the condition equation that causes the transition from source state to destination state

To see all of the transitions to and from each state name, click the appropriate column heading to sort on that column.

The text in each column is left-aligned by default; to change the alignment and more easily see the relevant part of the text, right-click in the column and click **Align Right**. To change back to left alignment, click **Align Left**.

Click in any cell in the table to select it. To select all cells, right-click in the cell and click **Select All**; or, on the Edit menu, click **Select All**. To copy selected cells to the clipboard, right-click the cells and click **Copy Table**; or, on the Edit menu, point to **Copy** and click **Copy Table**. You can paste the table into any text editor as tab-separated columns.

State Encoding Table

The state encoding table on the **Encoding** tab at the bottom of the State Machine Viewer window displays encoding information for each state transition.

To view state encoding information in the State Machine Viewer, you must have synthesized your design using Start Analysis & Synthesis. If you have only elaborated your design using Start Analysis & Elaboration, the encoding information is not displayed.

Selecting an Item in the State Machine Viewer

You can select and highlight each state node and transition in the State Machine Viewer. To select a state transition, click the arc that represents the transition.

When you select a state node, transition arc, or both in the state diagram view, the matching state node and equation conditions in the state transition table are highlighted. Conversely, when you select a state node, equation condition, or both in the state transition table, the corresponding state node and transition arc are highlighted in the state diagram view.

Switching Between State Machines

A design may contain multiple state machines. To choose which state machine to view, use the **State Machine** selection box located at the top of the State Machine Viewer. Click in the drop-down box and select the desired state machine.

Navigating the Schematic View

The previous sections provided an overview of the user interface for each netlist viewer, and how to select an item in each viewer. This section describes methods to navigate through the pages and hierarchy levels in the schematic view of the RTL Viewer and Technology Map Viewer.

Traversing and Viewing the Design Hierarchy

You can open different hierarchy levels in the schematic view using the hierarchy list (refer to "Hierarchy List" on page 13–14), or the **Hierarchy Up** and **Hierarchy Down** commands (right-click menu) in the schematic view.

Use the **Hierarchy Down** command to go down into, or expand an instance's hierarchy, and open a lower level schematic showing the internal logic of the instance. Use the **Hierarchy Up** command to go up in hierarchy, or collapse a lower level hierarchy, and open the parent higher level hierarchy. When the Selection Tool is selected, the appropriate option is available when your mouse pointer is located over an area of the schematic view that has a corresponding lower or higher level hierarchy.

The mouse pointer changes as it moves over different areas of the schematic to indicate whether you can move up, down, or both up and down in the hierarchy (Figure 13–6). To open the next hierarchy level, right-click in that area of the schematic and click **Hierarchy Down** or **Hierarchy Up**, as appropriate, or double-click in that area of the schematic.

Figure 13–6. Mouse Pointers Indicate How to Traverse Hierarchy

Flattening the Design Hierarchy

You can flatten the design hierarchy to view the design without hierarchical boundaries. To flatten the hierarchy from the current level and all the lower level hierarchies of the current design hierarchy, right-click in the schematic and click **Flatten Netlist**. To flatten the entire design, choose **Flatten Netlist** from the top-level schematic of the design.

Viewing the Contents of a Design Hierarchy within the Current Schematic

You can use the **Display Content** and **Hide Content** (right-click menu) commands to show or hide a lower hierarchy level for a specific instance within the schematic for the current hierarchy level.

To display the lower hierarchy netlist of an instance on the same schematic as the remaining logic in the currently viewed netlist, right-click the selected instance and click **Display Content**.

To hide all of the lower hierarchy logic of a hierarchy box into a closed instance, right-click the selected instance and click **Hide Content**.

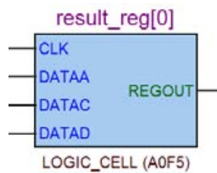
Viewing Contents of Atom Primitives

In the Technology Map Viewer, you can view the contents of certain device atom primitives to see their underlying implementation details. For logic cell (LCELL) atoms in the Stratix and Cyclone® series of devices, in Arria® GX devices, and in MAX® II devices, you can view LUTs, registers, and logic gates. For I/O atoms in the Stratix and Cyclone series of devices, in Arria GX devices, and HardCopy® II devices, you can view registers and logic gates.

In addition, you can view the implementation of RAM and DSP blocks in certain devices in the RTL Viewer or Technology Map Viewer. You can view the implementation of RAM blocks in the Stratix and Cyclone series of devices, and in Arria GX devices. You can view the implementation of DSP blocks only in the Stratix series of devices and Arria GX devices.

If you can view the contents of an atom instance, it is blue in the schematic view (Figure 13–7).

Figure 13–7. Instance That Can Be Expanded to View Internal Contents



To view the contents of one or more atom primitive instances, select the desired atom instances. Right-click a selected instance and click **Display Content**. You can also double-click on the desired atom instance to view the contents. Figure 13–8 shows an expanded version of the instance in Figure 13–7.

Figure 13–8. Internal Contents of the Atom Instance in Figure 13–7.

To hide the contents (and revert to the compact format), select and right-click the atom instance(s), and click **Hide Content**.

In the schematic view, the internal details within an atom instance cannot be selected as individual nodes. Any mouse action on any of the internal details is treated as a mouse action on the atom instance.

Viewing the Properties of Instances and Primitives

You can view the properties of an instance or primitive using the **Properties** dialog box. To view the properties of an instance or a primitive in the RTL Viewer or Technology Map Viewer, right-click the node and click **Properties**.

The **Properties** dialog box contains the following information about the selected node:

- The parameter values of an instance.
- The active level of the port (for example, active high or active low). An active low port is denoted with an exclamation mark "!".
- The port's constant value (for example, V_{CC} or GND). Table 13–5 describes the possible value of a port.

Table 13–5. Possible Port Values

Value	Description
V_{CC}	The port is not connected and has V_{CC} value (tied to V_{CC})
GND	The port is not connected and has GND value (tied to GND)
--	The port is connected and has value (other than V_{CC} or GND)
Unconnected	The port is not connected and has no value (hanging)

In the LUT of a logic cell (LCELL), the **Properties** dialog box contains the following additional information:

- The schematic of the LCELL.
- The Truth Table representation of the LCELL.

- The Karnaugh map representation of the LCELL.

Viewing LUT Representations in the Technology Map Viewer

You can view different representations of an LUT by right-clicking on the selected LUT and selecting **Properties**. This feature is supported for the Stratix and Cyclone series of devices, Arria GX devices, and MAX II devices only. There are three tabs in the **Properties** dialog box, which you can choose from to view the LUT representations:

- The **Schematic** tab (see Figure 13–9) shows you the equivalent gate representations of the LUT.
- The **Truth Table** tab (see Figure 13–10) shows the truth table representations.
- The **Karnaugh Map** tab (see Figure 13–11) shows the Karnaugh map representations of the LUT. The Karnaugh map supports up to 6 input LUTs.

For details about the **Ports** tab, refer to "Viewing the Properties of Instances and Primitives".

Figure 13–9. Schematic Tab

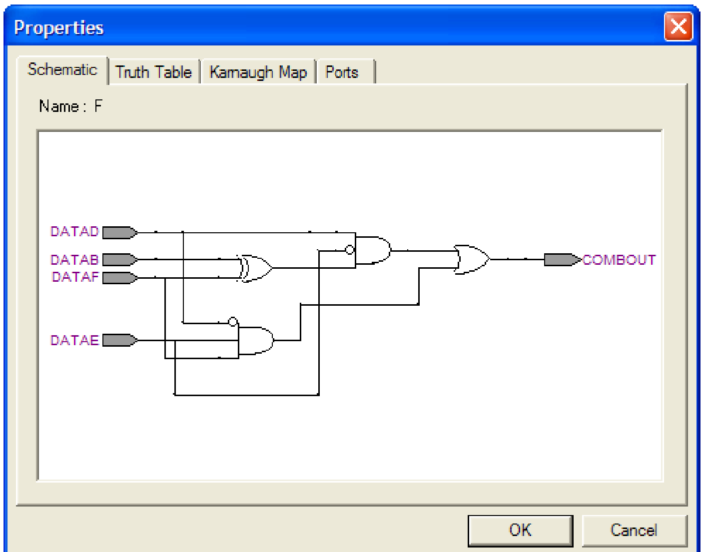
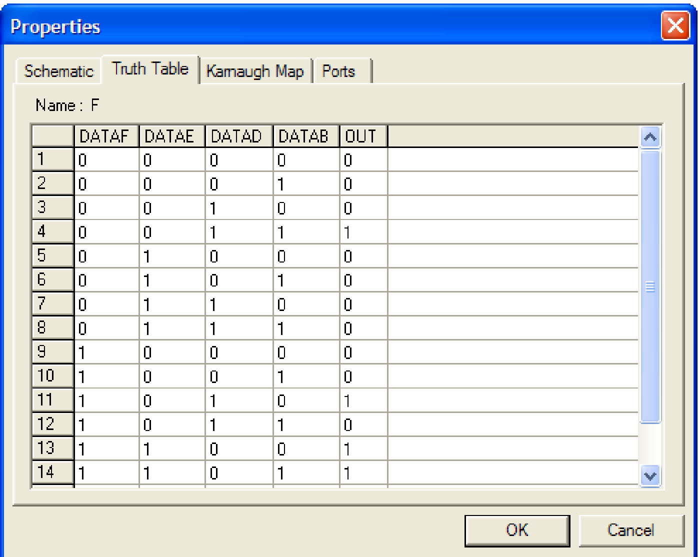
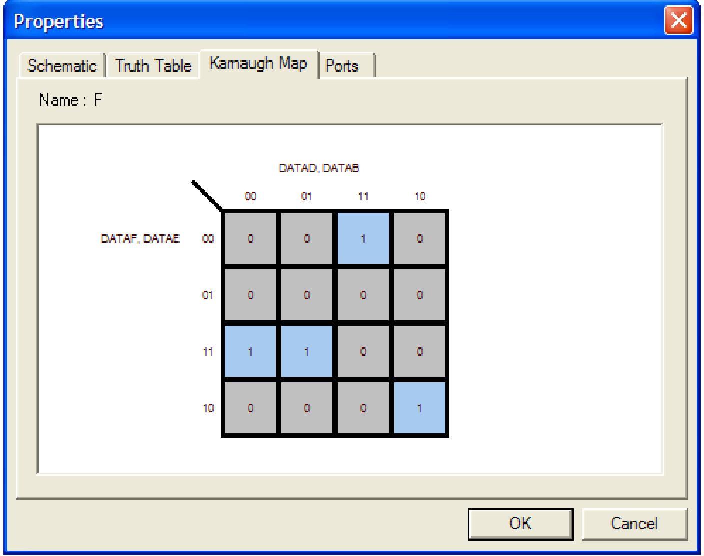


Figure 13–10. Truth Table Tab



	DATAF	DATAE	DATAD	DATAB	OUT
1	0	0	0	0	0
2	0	0	0	1	0
3	0	0	1	0	0
4	0	0	1	1	1
5	0	1	0	0	0
6	0	1	0	1	0
7	0	1	1	0	0
8	0	1	1	1	0
9	1	0	0	0	0
10	1	0	0	1	0
11	1	0	1	0	1
12	1	0	1	1	0
13	1	1	0	0	1
14	1	1	0	1	1

Figure 13–11. Karnaugh Map Tab



Grouping Combinational Logic into Logic Clouds

The following sections describes how to group combinational logic into logic clouds.

☞ For the definition of a logic cloud, refer to Table 13–1 on page 13–9.

Logic Clouds in the RTL Viewer

You can automatically group all combinational logic nodes in your design into logic clouds. On the Tools menu, click **Options**, and in the **Category** list, click the "+" to expand **Netlist Viewers** and select **RTL Viewer**. On the **RTL Viewer** page, turn on **Group combinational logic into logic cloud**. You can also turn on this option by right-clicking in the schematic and clicking **Viewer Options**. In the **RTL/Technology Map Viewer Options** dialog box, click the **Customize View** tab. Under **Customize Groups** section, turn on **Group combinational logic into logic cloud**. Figure 13–12 and Figure 13–13 show the schematic before and after the combinational logic grouping operation in the RTL Viewer.

Logic Clouds in the Technology Map Viewer

In the Technology Map Viewer, the **Group combinational logic into logic clouds** option is supported for Stratix II, Cyclone II, and HardCopy families of devices only. To set this option, right-click in the schematic and click **Viewer Options**. In the **RTL/Technology Map Viewer Options** dialog box, click on the **Customize View** tab. Turn on the **Group combinational logic into logic cloud** option.

Figure 13–12. Schematic Before Combinational Logic Grouping

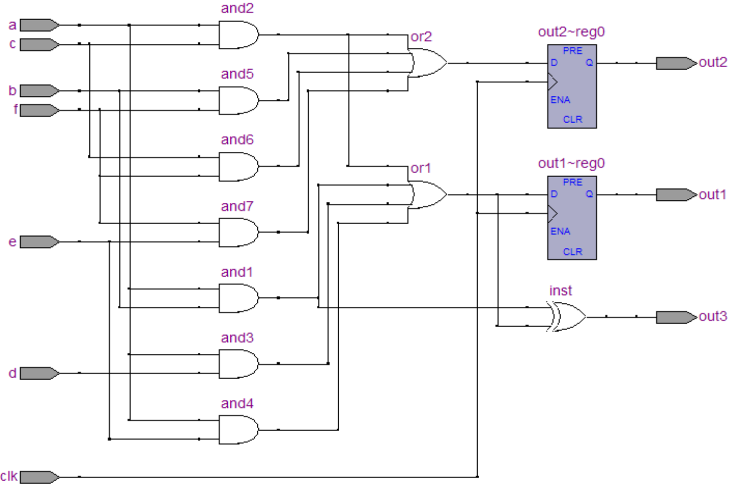
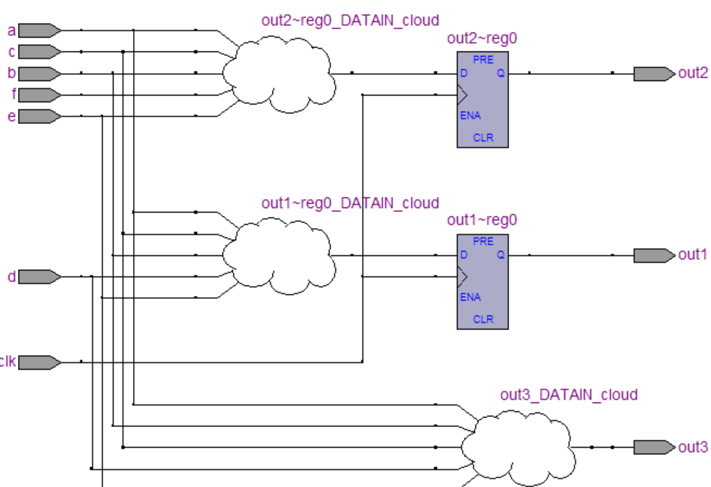


Figure 13–13. Schematic After Combinational Logic Grouping



Manually Group and Ungroup Logic Clouds

To group logic nodes into a logic cloud manually, right-click the selected node or input port and select **Group source logic into logic cloud**. To ungroup a logic cloud manually, right-click on the selected logic cloud and select **Ungroup source logic from logic cloud**. You can also ungroup a logic cloud manually by double-clicking on the selected logic cloud. These options are not available if the nodes cannot be grouped.

Changing the Constant Signal Value Formatting

The constant signal value is highlighted in gray in the schematic view. By default, the value is displayed in hexadecimal format, but you can also choose binary or decimal format. To change the value formatting, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and select the desired format from the **Constant Signal Format** list.

Changing the format affects all constant signal values throughout the schematic. Refer to Table 13–3 on page 13–12 to see what constant signal values look like in the schematic.

Zooming and Magnification

You can control the magnification of your schematic on the View menu, with the Zoom Tool in the toolbar, or the **Ctrl** key and mouse wheel button, as described in this section.

The **Fit in Window**, **Fit Selection in Window**, **Zoom In**, **Zoom Out**, and **Zoom** commands are available on the View menu, by right-clicking in the schematic view and selecting **Zoom**, or from the Zoom toolbar. To enable the Zoom toolbar, on the Tools menu, click **Customize**. Click the **Toolbars** tab and click **Zoom** to enable the toolbar.

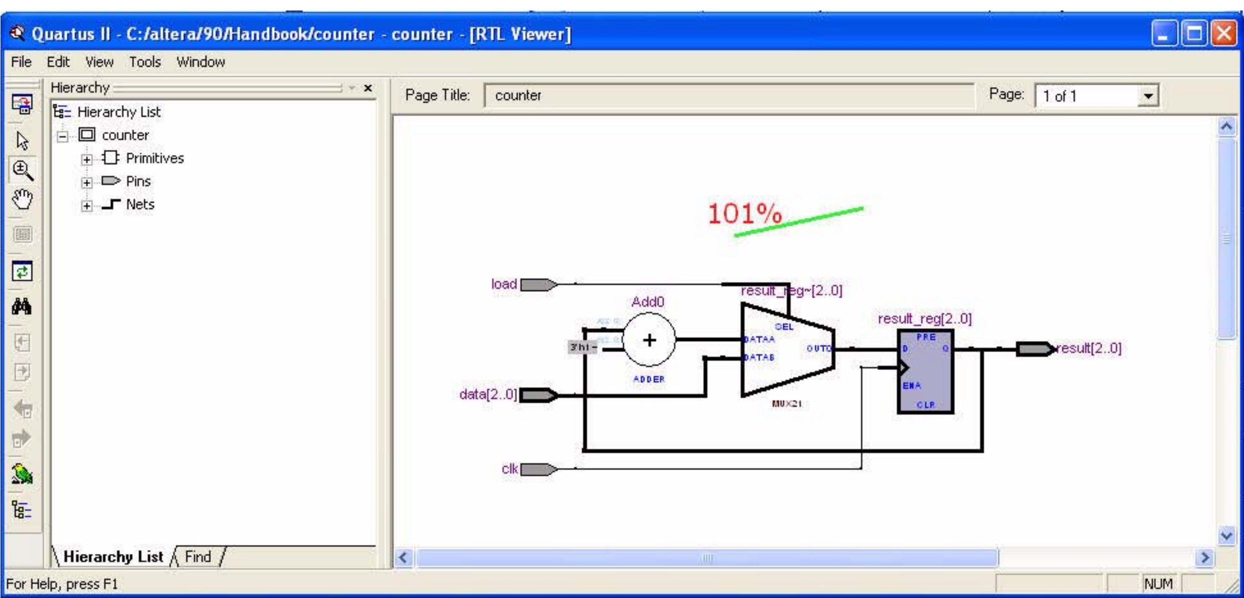
By default, the viewer displays most pages sized to fit in the window. If the schematic page is very large, the schematic is displayed at the minimum zoom level, and the view is centered on the first node. Click **Zoom In** to view the image at a larger size, and click **Zoom Out** to view the image (when the entire image is not displayed) at a smaller size. The **Zoom** command allows you to specify a magnification percentage (100% is considered the normal size for the schematic symbols). To change the minimum and maximum zoom level, on the Tools menu, click **Options**. In the **Options** dialog box, in the **Category** list, select **Netlist Viewers** and set the desired minimum and maximum zoom level.

The **Fit Selection in Window** command zooms in on the selected nodes in a schematic to fit within the window. Use the Selection Tool to select one or more nodes (instances, primitives, pins, and nets), then select **Fit Selection in Window** to enlarge the area covered by the selection. This feature is helpful when you want to see a particular element in a large schematic. After you select a node, you can easily zoom in to view the particular node.

You can also use the Zoom Tool on the viewer toolbar to control magnification in the schematic view. When you select the Zoom Tool in the toolbar, clicking in the schematic zooms in and centers the view on the location you clicked. Right-click on the schematic to zoom out and center the view on the location you clicked. When you select the Zoom Tool, you can also zoom in to a certain portion of the schematic by selecting a rectangular box area with your mouse cursor. The schematic is enlarged to show the selected area.

Alternatively, you can specify the magnification percentage by right-clicking on the desired area and dragging the mouse toward your right to zoom in or toward your left to zoom out with the Zoom Tool. You will see a green line with the zoom percentage above it. The zoom percentage is proportional to the length of the green line (Figure 13–14). Release the mouse button at the desired zoom percentage.

Figure 13–14. Dragging the Mouse Pointer to Change Zoom Percentage



By default, the viewers maintain the zoom level when filtering on the schematic (refer to "Filtering in the Schematic View" on page 13–29). To change the behavior so that the zoom level is always reset to "Fit in Window," on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers**, and turn off **Maintain zoom level**.

Schematic Debugging and Tracing Using the Bird's Eye View

Viewing the entire schematic can be useful when debugging and tracing through a large netlist. The Quartus II software allows you to view the entire schematic in a single window. The bird's eye view is displayed in a separate window that is linked directly to the netlist viewers. This feature is available in the RTL, Technology Map, and Technology Map (Post-Mapping) viewers.

The bird's eye view shows the current area of interest. Select the desired area by clicking and dragging the indicator or using the right-mouse button to form a rectangular box around the desired area. You can also click and drag the rectangular box to move around the schematic. To open the bird's eye view, on the View menu, click **Bird's Eye View**, or click on the **Bird's Eye View** icon in the Viewer toolbar (Figure 13–15).

Figure 13–15. Bird's Eye View and Full Screen Icon

Full Screen View

To set the viewer window to fill the whole screen, on the View menu, click **Full Screen**, or click the **Full Screen** icon in the viewer toolbar (Figure 13–15), or press **Ctrl+Alt+Space**. The keyboard shortcut toggles between the full screen and standard screen views.

Partitioning the Schematic into Pages

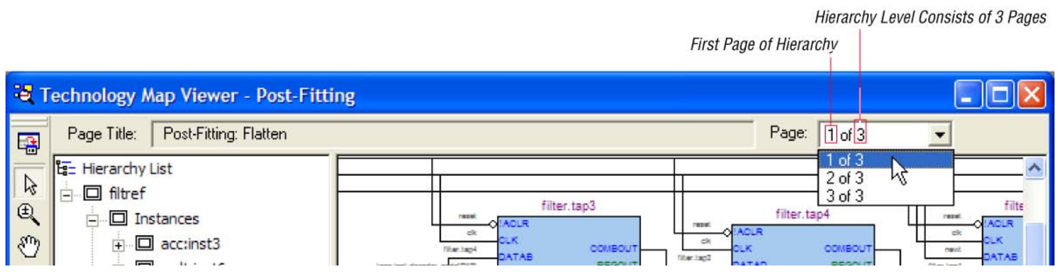
For large design hierarchies, the RTL Viewer and Technology Map Viewer partition your netlist into multiple pages in the schematic view. To control how much of the design is visible on each page, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and set the desired options under **Display Settings**.

The **Nodes per page** option specifies the number of nodes per partitioned page. The default value is 50 nodes; the range is 1 to 1,000 nodes. The **Ports per page** option specifies the number of ports (or pins) per partitioned page. The default value is 1,000 ports (or pins); the range is 1 to 2,000 ports (or pins). The viewers partition your design into a new page if either the node number or the port number exceeds the limit you have specified. You might occasionally see the number of ports exceed the limit, depending on the configuration of nodes on the page.

If the **Display boundary around hierarchy levels** option is turned on and the total number of nodes or ports within the hierarchy exceeds the value of **Nodes per page** or **Ports per page**, the boundary is displayed as a hierarchy port connector (refer to Table 13–1 on page 13–9). For more information about the **Display boundary around hierarchy levels** option, refer to "Filtering Across Hierarchies" on page 13–33.

When a hierarchy level is partitioned into multiple pages, the title bar for the schematic window indicates which page is displayed and how many total pages exist for this level of hierarchy (shown in the format:
Page <*current page number*> **of** <*total number of pages*>), as shown in Figure 13–16.

Figure 13–16. RTL Viewer Title Bars Indicating Page Number Information



When you change the number of nodes or ports per page, the change applies only to new pages that are shown or opened in the viewer. To refresh the current page so that it displays the changed number of nodes or ports, click the **Refresh** button in the toolbar.

Moving between Schematic Pages

To move to another schematic page, on the View menu, click **Previous Page** or **Next Page**, or click the **Previous Page** icon or the **Next Page** icon in the viewer toolbar.

To go to a particular page of the schematic, on the Edit menu, click **Go To**, or right-click in the schematic view and click **Go To**. In the **Page** list, select the desired page number. You can also go to a particular page by selecting the desired page number from the pull-down list on the top right of the viewer window.

Moving Back and Forward Through Schematic Pages

To return to the previous view after changing the page view, click **Back** on the View menu, or click the **Back** icon on the viewer toolbar. To go to the next view, click **Forward** on the View menu, or click the **Forward** icon on the viewer toolbar.

You can go forward only if you have not made any changes to the view since going back. Use the **Back** and **Forward** commands to switch between page views. These commands do not undo an action, such as selecting a node.

Following Nets Across Schematic Pages

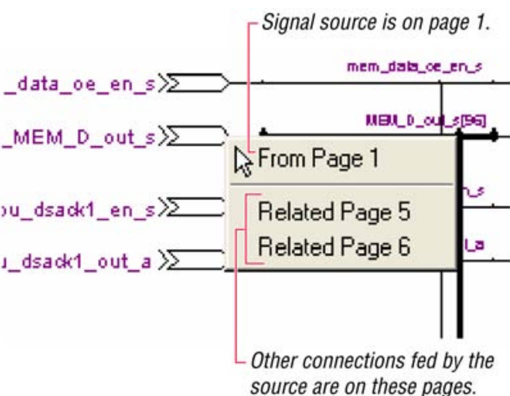
Input and output connectors indicate nodes that connect across pages of the same hierarchy. Right-click on a connector to display a menu of commands that trace the net through the pages of the hierarchy.

After you right-click to follow a connector port, the viewer opens a new page, which centers the view on the particular source or destination net using the same zoom factor used by the previous page. To trace a specific net to the new page of the hierarchy, Altera recommends that you first select the desired net, which highlights it in red, before you right-click to traverse pages.

Input Connectors

Figure 13–17 shows an example of the menu that appears when you right-click an input connector. The **From** command opens the page containing the source of the signal. The **Related** commands, if applicable, open the specified page containing another connection fed by the same source.

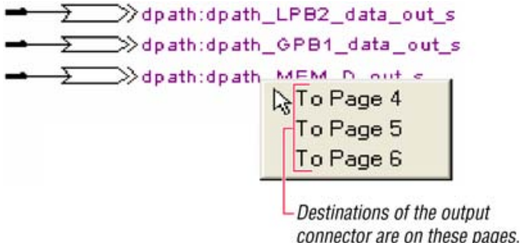
Figure 13–17. Input Connector Right Button Pop-Up Menu



Output Connectors

Figure 13–18 shows an example of the menu that appears when you right-click an output connector. The **To** command opens the specified page that contains a destination of the signal.

Figure 13–18. Output Connector Right Button Pop-Up Menu



Go to Net Driver

To locate the source of a particular net in the schematic view, select the net to highlight it, right-click the selected net, point to **Go to Net Driver** and click **Current page**, **Current hierarchy**, or **Across hierarchies**. Refer to Table 13–6 for details.

Table 13–6. Go to Net Driver Commands

Command	Action
Current page	Locates the source or driver on the current page of the schematic only.
Current hierarchy	Locates the source within the current level of hierarchy, even if the source is located on another page of the netlist schematic.
Across hierarchies	Locates the source across hierarchies until the software reaches the source at the top hierarchy level.

The schematic view opens the correct page of the schematic if required, and adjusts the centering of the page so that you can see the net source. The schematic shows the default page for the net driver. The view is an unfiltered view, so no filtering results are kept.

Customizing the Schematic Display in the RTL Viewer

You can customize the schematic display for better viewing and to speed up your debugging process. The options that control the schematic display are available in the **Customize View** tab of the **RTL/Technology Map Viewer Options** dialog box. To open the dialog box, right-click in the schematic and click **Viewer Options**. You can turn on the options to remove fan-out free nodes, simplify logic, group or ungroup related nodes, and group combinational logic into a logic cloud.

You can also customize the schematic view in the RTL Viewer by clicking **Options** on the Tools menu. In the **Category** list, click the "+" icon to expand **Netlist Viewers** and select **RTL Viewer**. Set the desired customization for your schematic display.

When the settings are changed, the list of previously viewed pages is cleared. The settings are revision-specific, so different revisions can have different settings.

To remove fan-out free registers from your schematic display, turn on **Remove registers without fan-out**. By default, this option is turned on.

To remove all single-input nodes and merge a chain of equivalent combinational gates that have direct connections (without inversion in between) into a single multiple-input gate, turn on **Show simplified logic.** By default, this option is turned on.

To group all related nodes into a single node, turn on **Group all related nodes**. This option is turned on by default. You can manually group or ungroup any nodes by right-clicking the selected nodes in the schematic and selecting **Group Related Nodes** to group or **Ungroup Selected Nodes** to ungroup.

Filtering in the Schematic View

Filtering allows you to filter out nodes and nets in your netlist to view only the logic that interests you.

Filter your netlist by selecting hierarchy boxes, nodes, ports of a node, nets, or states in a state machine that are part of the path you want to see. The following filter commands are available:

- **Sources—**Displays the sources of the selection
- **Destinations—**Displays the destinations of the selection
- **Sources & Destinations—**Displays both the sources and destinations of the selection
- **Selected Nodes and Nets—**Displays only the selected nodes and nets with the connections between them
- **Between Selected Nodes—**Displays nodes and connections in the path between the selected nodes

- **Bus Index**—Displays the sources or destinations for one or more indices of an output or input bus port

Select a hierarchy box, node, port, net, or state node, right-click in the window, point to **Filter** and click the appropriate filter command. The viewer generates a new page showing the netlist that remains after filtering.

When filtering in a state diagram in the State Machine Viewer, sources and destinations refer to the previous and next transition states or paths between transition states in the state diagram. The transition table and encoding table also reflect the filtering.

You can go back to the netlist page before it was filtered using the **Back** command, described in "Moving Back and Forward Through Schematic Pages" on page 13–27.

When viewing a filtered netlist, clicking an item in the hierarchy list causes the schematic view to display an unfiltered view of the appropriate hierarchy level. You cannot use the hierarchy list to select items or navigate in a filtered netlist.

Filter Sources Command

To filter out all but the source of the selected item, right click the item, point to **Filter** and click Sources. The selected object type determines what is displayed, as outlined in Table 13–7 and shown in Figure 13–19.

Table 13–7. Selected Objects Determine Filter Sources Display

Selected Object	Result Shown in Filtered Page
Node or hierarchy box	Shows all the sources of the node's input ports. For an example, refer to Figure 13–19.
Net	Shows the sources that feed the net.
Input port of a node	Shows only the input source nodes that feed this port.
Output port of a node	Shows only the selected node.
State node in a state machine	Shows the states that feed the selected state (previous transition states).

Filter Destinations Command

To filter out all but the destinations of the selected node or port as outlined in Table 13–8 and shown in Figure 13–19, right-click the node or port, point to **Filter** and click **Destinations**.

Table 13–8. Selected Objects Determine Filter Destinations Display

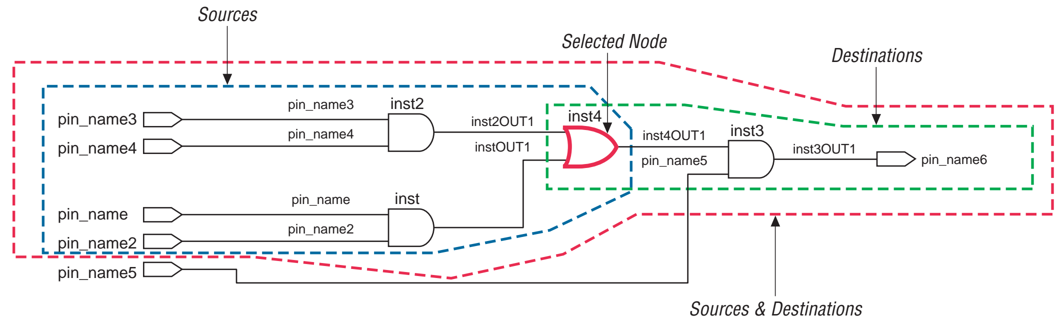
Selected Object	Result Shown in Filtered Page
Node or hierarchy box	Shows all the destinations of the node's output ports. For an example, refer to Figure 13–19.
Net	Shows the destinations fed by the net.
Input port of a node	Shows only the selected node.
Output port of a node	Shows only the fan-out destination nodes fed by this port.
State node in a state machine	Shows the states that are fed by the selected states (next transition states).

 © March 2009 Altera Corporation

Filter Sources and Destinations Command

The **Sources & Destinations** command is a combination of the **Sources** and **Destinations** filtering commands, in which the filtered page shows both the sources and the destinations of the selected item. To select this option, right-click on the desired object, point to **Filter** and click **Sources & Destinations**. Refer to the example in Figure 13–19.

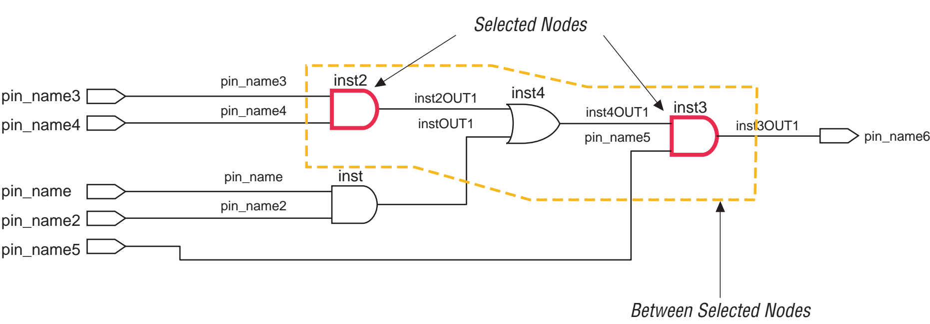
Figure 13–19. Sources, Destinations, and Sources and Destinations Filtering for inst4



Filter Between Selected Nodes Command

To show the nodes in the path between two or more selected nodes or hierarchy boxes, right-click, point to **Filter** and click **Between Selected Nodes**. For this option, selecting a port of a node is the same as selecting the node. For an example, refer to Figure 13–20.

Figure 13–20. Between Selected Nodes Filtering Between inst2 and inst3



Filter Selected Nodes and Nets Command

To create a filtered page that shows only the selected nodes, nets, or both, and, if applicable, the connections between the selected nodes, nets, or both, right-click, point to **Filter**, and click **Selected Nodes & Nets**. Figure 13–21 shows a schematic with several nodes selected.

Figure 13–21. Using Selected Nodes and Nets to Select Nodes

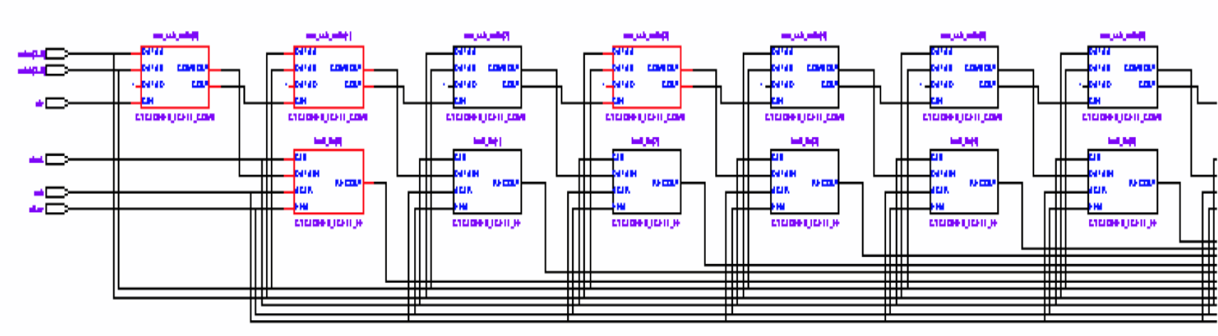
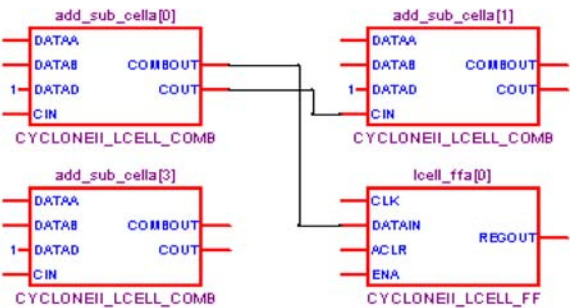


Figure 13–22 shows the schematic after filtering has been performed. If you select a net, the filtered page shows the immediate sources and destinations of the selected net.

Figure 13–22. Selected Nodes and Nets Filtering on Figure 13–21 Schematic



Filter Bus Index Command

To show the path related to a specific index of a bus input or output port in the RTL Viewer, right-click the port, point to **Filter**, and click **Bus Index**. The **Select Bus Index** dialog box allows you to select the indices of interest.

Filter Command Processing

The options to control filtering are available in the **Tracing** section of the **RTL/Technology Map Viewer Options** dialog box. Right-click in the schematic and click **Viewer Options** to open the dialog box.

For all the filtering commands, the viewer stops tracing through the netlist to obtain the filtered netlist when it reaches one of the following objects:

- A pin
- A specified number of filtering levels, counting from the selected node or port; the default value is 3

Specify the number of filtering levels in the **Tracing** section of the **RTL/Technology Map Viewer Options** dialog box. The default value is 3, which ensures optimal processing time when performing filtering, but you can specify a value from 1 to 15. To see more than 15 levels, set the **Expanding** option. For more information, refer to "Expanding a Filtered Netlist".

- A register (optional; turned on by default)

Turn the **Stop filtering at register** option on or off in the **Tracing** section of the **RTL/Technology Map Viewer Options** dialog box. Right-click in the schematic and click **Viewer Options** to open the dialog box.

By default, the filtered schematic shows all possible connections between the nodes shown in the schematic. To remove the connections that are not directly part of the path that was traced to generate a filtered netlist, turn off the **Shows all connections between nodes** option in the **Tracing** section of the **RTL/Technology Map Viewer Options** dialog box.

Filtering Across Hierarchies

The filtering commands display nodes in all hierarchies by default. When the filtered path passes through levels of hierarchy on the same schematic page, green hierarchy boxes group the logic and show the hierarchy boundaries. A green rectangular symbol appears on the border that represents the port relationship between two different hierarchies (Figure 13–23 and Figure 13–24).

The **RTL/Technology Map Viewer Options** dialog box provides an option to control filtering if you prefer to filter only within the current hierarchy. Right-click in the schematic and click **Viewer Options**. In the **Tracing** section, turn off the **Filter across hierarchy** option.

To disable the box hierarchy display, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and turn off **Display boundary around hierarchy levels**.

Netlists of the same hierarchy that are displayed over more than one page are not grouped with a box. Filtering and expanding on a blue atom primitive does not trace the underlying netlist even when **Filter across hierarchy** is enabled.

Figure 13–23 and 13–34 show examples of filtering across hierarchical boundaries. Figure 13–23 shows an example after the **Sources** filter has been applied to an input port of the `taps` instance, where the input port of the lower level hierarchical block connects directly to an input pin of the design. The name of the instance is indicated within the green border and appears as a tooltip when you move your mouse pointer over the instance.

Figure 13–23. Filtering Across Hierarchical Boundaries, Small Example

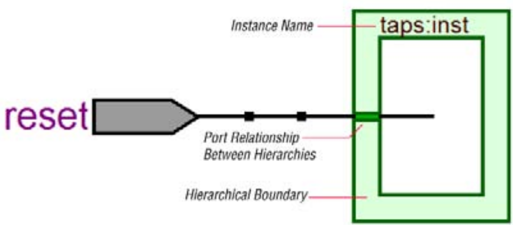
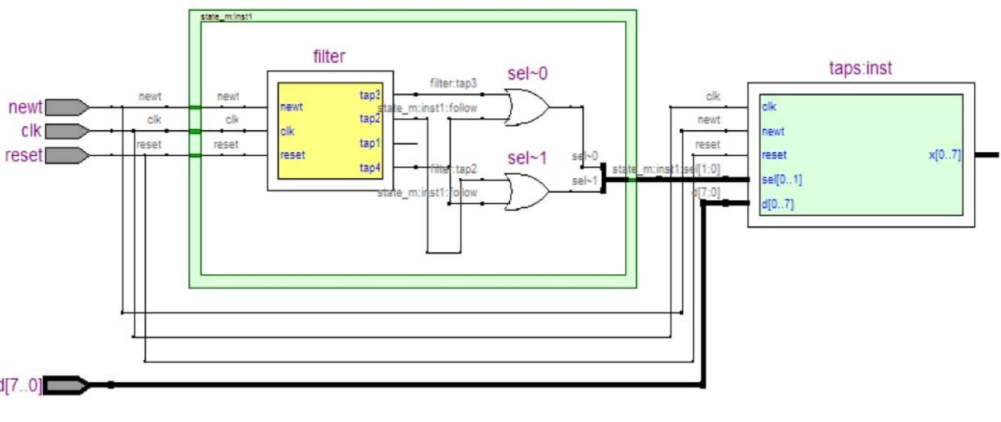


Figure 13–24 shows a larger example after the **Sources** filter has been applied to an input port of an instance, in which the source comes from input pins that are fed through another level of hierarchy.

Figure 13–24. Filtering Across Hierarchical Boundaries, Large Example



Sources command applied to an input port of an instance in which the source comes from input pins that are fed through another level of hierarchy.

Expanding a Filtered Netlist

After a netlist is filtered, some ports might not have connections displayed because their connections are not part of the main path through the netlist. Two expansion features, immediate expansion and the **Expand** command, allow you to add the fan-in or fan-out signals of these ports to the schematic display of a filtered netlist.

You can immediately expand any port whose connections are not displayed. When you double-click that port in the filtered schematic, one level of logic is expanded.

To expand more than one level of logic, right-click the port and click the **Expand** command. This command expands logic from the selected port by the amount specified in **Viewer Options**. To set these options, right-click in the schematic view and click **Viewer Options**. In the **Expansion** section, set the **Number of expansion levels** option to specify the number of levels to expand (the default value is 3 and the range is 1 to 15 levels). You can also set the **Stop expanding at register** option (which is turned on by default) to specify whether netlist expansion should stop when a register is reached.

You can select multiple nodes to expand when you use the **Expand** command. If you select ports that are located on multiple schematic pages, only the ports on the currently viewed page appear in the expanded schematic.

In the State Machine Viewer, the **Expand** command has the following three options:

- **Sources**—Displays the states that feed the selected states (previous transition states)
- **Destinations**—Displays the states that are fed by the selected states (next transition states)
- **Sources & Destinations**—Displays both the previous and next transition states

The state transition table and state encoding table also reflect the changes to the filter.

The expansion feature works across hierarchical boundaries if the filtered page containing the port to be expanded was generated with the **Filter across hierarchy** option turned on (refer to "Filtering in the Schematic View" on page 13–29 for details about this option). When viewing timing paths in the Technology Map Viewer, the **Expand** command always works across hierarchical boundaries because filtering across hierarchy is always turned on for these schematics (refer to "Viewing a Timing Path" on page 13–38 for details about these schematics).

Reducing a Filtered Netlist

In some cases, removing logic from a filtered schematic or state diagram makes the schematic view easier to read or minimizes distracting logic that you do not require to view in the schematic.

To reduce elements in the filtered schematic or state diagram view, right-click the node or nodes you want to remove and click **Reduce**.

Probing to Source Design File and Other Quartus II Windows

The RTL Viewer, Technology Map Viewer, and State Machine Viewer let you cross-probe from the viewer to the source design file and to various other windows within the Quartus II software. You can select one or more hierarchy boxes, nodes, nets, state nodes, or state transition arcs that interest you in the viewer and locate the corresponding items in another applicable Quartus II software window. You can then view and make changes or assignments in the appropriate editor or floorplan.

To locate an item from the viewer in another window, right-click the items of interest in the schematic or state diagram view, point to **Locate**, and click the appropriate command. The following commands are available:

- **Locate in Assignment Editor**
- **Locate in Pin Planner**
- **Locate in Timing Closure Floorplan**
- **Locate in Chip Planner**
- **Locate in Resource Property Editor**
- **Locate in RTL Viewer**
- **Locate in Technology Map Viewer**

- **Locate in Design File**

The options available for locating depend on the type of node and whether it exists after placement and routing. If a command is enabled in the menu, it is available for the selected node. You can use the **Locate in Assignment Editor** command for all nodes, but assignments might be ignored during placement and routing if they are applied to nodes that do not exist after synthesis.

The viewer automatically opens another window for the appropriate editor or floorplan and highlights the selected node or net in the newly opened window. You can switch back to the viewer by selecting it in the Window menu or by closing, minimizing, or moving the new window.

When probing to a logic cloud in the RTL Viewer, a message box appears that prompts you to ungroup the logic cloud or allow it to remain grouped.

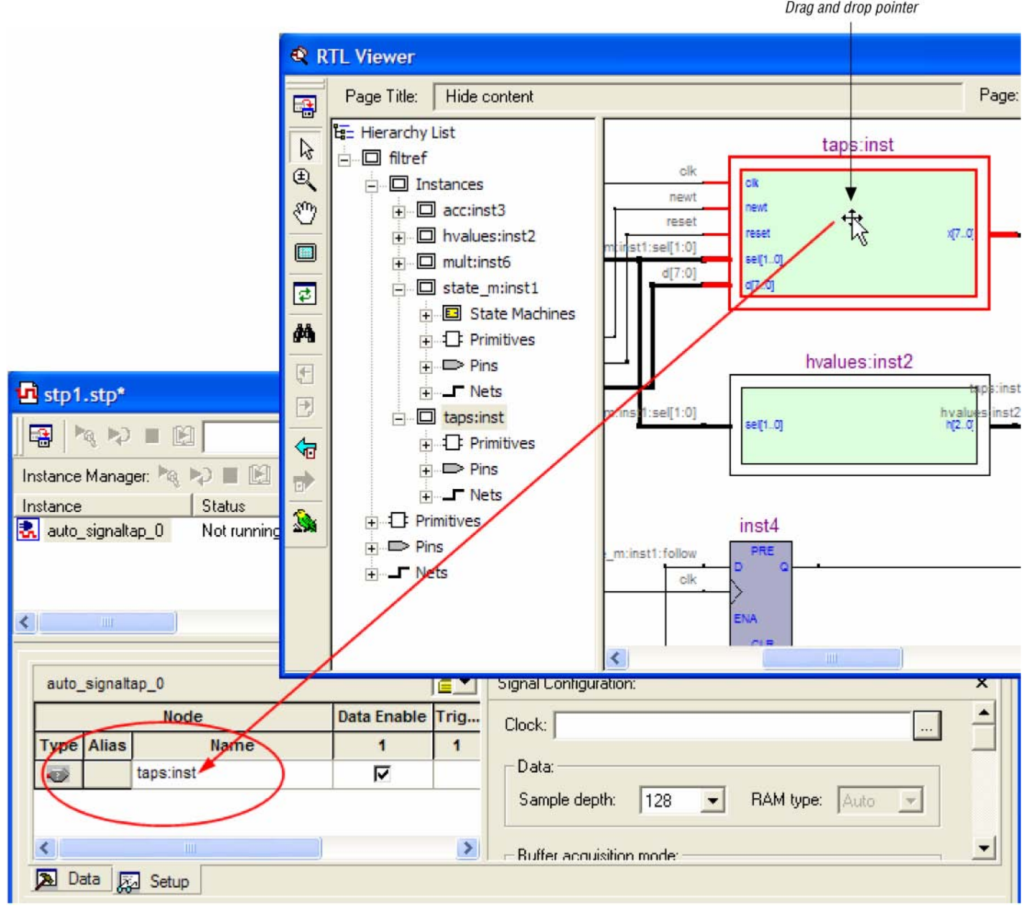
Moving Selected Nodes to Other Quartus II Windows

You can drag selected nodes from the netlist viewers to the Text Editor, Block Editor, Pin Planner, SignalTap® II Embedded Logic Analyzer, and Waveform Editor windows within the Quartus II software. Whenever you see the drag-and-drop pointer on the selected node in the netlist viewers, it means that the node can be dragged to other child windows within the Quartus II software.

To tap a node from the schematic in the Technology Map Viewer to an open SignalTap II Embedded Logic Analyzer window or to a new SignalTap II file (**.stf**), right-click on the selected node in the schematic diagram or in the hierarchy list and then click **Add Node to SignalTap II Logic Analyzer**. If the node cannot be tapped, the option is unavailable.

Figure 13–25 shows the drag-and-drop pointer and an example of dragging a node from the RTL Viewer to the SignalTap II Logic Analyzer.

Figure 13–25. Dragging a Node to the SignalTap II Logic Analyzer



Probing to the Viewers from Other Quartus II Windows

You can cross-probe to the RTL Viewer and Technology Map Viewer from other windows within the Quartus II software. You can select one or more nodes or nets in another window and locate them in one of the viewers.

You can locate nodes between the RTL Viewer, State Machine Viewer, and Technology Map Viewer and you can locate nodes in the RTL Viewer or Technology Map Viewer from the following Quartus II software windows:

- Project Navigator
- Timing Closure Floorplan
- Chip Planner
- Resource Property Editor
- Node Finder
- Assignment Editor
- Messages Window
- Compilation Report

- TimeQuest Timing Analyzer (supports the Technology Map Viewer only)

To locate elements in the viewer from another Quartus II window, select the node or nodes in the appropriate window; for example, select an entity in the **Entity** list on the **Hierarchy** tab in the Project Navigator, or select nodes in the Timing Closure Floorplan, or select node names in the **From** or **To** column in the Assignment Editor. Next, right-click the selected object, point to **Locate**, and click **Locate in RTL Viewer** or **Locate in Technology Map Viewer**. After you choose this command, the viewer window opens, or is brought to the foreground if the viewer window is already open.

☞ The first time the window opens after a compilation, the preprocessor stage runs before the viewer window opens.

The viewer shows the selected nodes and, if applicable, the connections between the nodes. The display is similar to what you see if you right-click the object, point to **Filter,** and click **Selected Nodes & Nets using Filter Across Hierarchy**. If the nodes cannot be found in the viewer, a message box displays the message: **Can't find requested location**.

Viewing a Timing Path

To see a visual representation of a timing path, cross-probe from the Timing Analysis section of the Compilation Report with the Classic Timing Analyzer, or from a report panel in the TimeQuest Timing Analyzer.

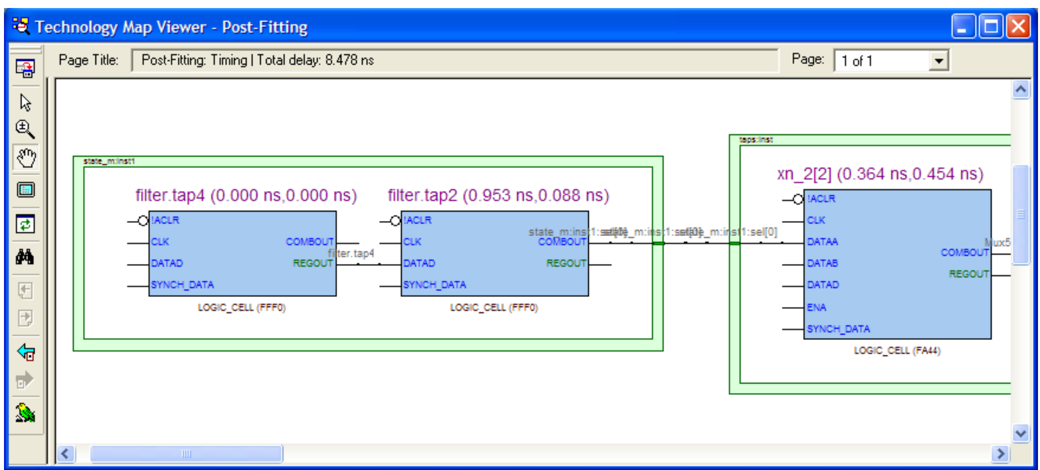
To take advantage of this feature, you must first successfully complete a full compilation of your design, including the timing analyzer stage. To access the timing analyzer report that contains the timing results for your design, on the Processing menu, click **Compilation Report**. On the left side of the Compilation Report, select **Timing Analyzer** or **TimeQuest Timing Analyzer**. When you select a detailed report, the timing information is listed in a table format on the right side of the Compilation Report; each row of the table represents a timing path in the design. You can also view timing paths in TimeQuest report panels. To view a particular timing path in the Technology Map Viewer or RTL Viewer, highlight the appropriate row in the table, right-click, point to **Locate**, and click **Locate in Technology Map Viewer** or **Locate in RTL Viewer**.

In the Technology Map Viewer, the schematic page displays the nodes along the timing path with a summary of the total delay. If you locate from the Classic Timing Analyzer, the timing path also includes timing data representing the interconnect (IC) and cell delays associated with each node. The delay for each node is shown in the following format: *<post-synthesis node name>* (*<IC delay>* ns, *<cell delay>* ns).

When you locate the timing path from the TimeQuest Timing Analyzer to the Technology Map Viewer, the interconnect and cell delay associated with each node is displayed on top of the schematic symbols. The total slack of the selected timing path is displayed in the **Page Title** section of the schematic. If the nodes are grouped in a logic cloud, the delay information displayed with the logic cloud is the total sum delay of the grouped nodes. The delay information for each node in the logic cloud is displayed in a tooltip. Move the mouse pointer over the logic cloud to see the tooltip. For more information about tooltips, refer to "Tooltips" on page 13–39.

Figure 13–26 shows a portion of a Classic Timing Analyzer timing path represented in the Technology Map Viewer. The total delay for the entire path through several levels of logic (only three levels are shown in Figure 13–26) is 7.159 ns. The delays are indicated for each level of logic. For example, the IC delay to the first LCELL primitive is 0.383 ns and the cell delay through the LCELL is 0.075 ns. When the timing path passes through a level of hierarchy, green hierarchy boxes group the logic and show the hierarchical boundaries. A green rectangular symbol on the border indicates the path passes between two different hierarchies.

Figure 13–26. Timing Path Schematic in the Technology Map Viewer



In the RTL Viewer, the schematic page displays the nodes in the paths between the source and destination registers with a summary of the total delay.

The RTL Viewer netlist is based on an initial stage of synthesis, so the post-fitting nodes might not exist in the RTL Viewer netlist. Therefore, the internal delay numbers are not displayed in the RTL Viewer as they are in the Technology Map Viewer, and the timing path might not be displayed exactly as it appears in the timing analysis report. If multiple paths exist between the source and destination registers, the RTL Viewer might display more than just the timing path. There are also some cases in which the path cannot be displayed, such as paths through state machines, encrypted intellectual property (IP), or registers that are created during the fitter process. In cases where the timing path displayed in the RTL Viewer might not be the correct path, the compiler issues messages.

Other Features in the Schematic Viewer

This section describes other features in the schematic view that enhance usability and help you analyze your design.

Tooltips

A tooltip is displayed whenever the mouse pointer is held over an element in the schematic. The tooltip contains useful information about a node, net, logic cloud, input port, and output port. Table 13–9 lists the information contained in the tooltip for each type of node.

The tooltip information for an instance (the first row in Table 13–9) includes a list of the primitives found within that level of hierarchy and the number of each primitive contained in the current instance. The number includes all hierarchical blocks below the current instance in the hierarchy. This information lets you estimate the size and complexity of a hierarchical block without navigating into the block.

The tooltip information for atom primitives in the Technology Map Viewer (the second row of Table 13–9) shows the equation for the design atom. The equations are an expanded version of the equations you can view in the Equations window in the Timing Closure Floorplan. Advanced users can use these equations to analyze the design implementation in detail.

For details about understanding equations, refer to the Quartus II Help.

To copy tooltips into the clipboard for use in other applications, right-click the desired node or netlist and click **Copy Tooltip**.

To turn off tooltips or change the duration of time that a tooltip is displayed in the view, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and set the desired options under **Tooltip settings**.

The **Show names in tooltip for** option specifies the number of seconds to display the names of assigned nodes and pins in a tooltip when the pointer is over the assigned nodes and pins. Selecting **Unlimited** displays the tooltip as long as the pointer remains over the node or pin. Selecting **0** turns off tooltips. The default value is 5 seconds.

The **Delay showing tooltip for** option specifies the number of seconds you must hold the mouse pointer over assigned nodes and pins before the tooltip displays the names of the assigned nodes and pins. Selecting **0** displays the tooltip immediately when the pointer is over an assigned node or pin. Selecting **Unlimited** prevents tooltips from being displayed. The default value is 1 second.

Table 13–9. Tooltip Information (Part 1 of 2)

<table>
<tr><th>Tooltip Format</th><th>Description</th><th>Example Tooltips</th></tr>
<tr><td>Instance</td><td>Format: <<i>instance name</i>>, <<i>instance type</i>>
<<i>primitive type</i>>, <<i>number of primitives</i>>...
<<i>primitive type</i>>, <<i>number of primitives</i>></td><td>taps:inst, INST
DFF 32
OPERATOR(SELECTOR) 8
OPERATOR(DECODER) 1</td></tr>
<tr><td>Atom Primitive</td><td>Format: <instance name>, <<i>primitive name</i>> (<<i>LUT Mask Value</i>>)
{(<code>r</code> | <code>c</code> <<i>Register or Combinational equation</i>>)}
...
An <code>r</code> (as in the first example) represents the equation for a register, and a <code>c</code> (as in the second example) represents the equation for combinational logic.</td><td>inst5[3], LCELL (0000)
<r> inst5[3]
= DFFEAS(
(GND), GLOBAL(CLK), VCC, , ENA, SYNCH_DATA, , , VCC)

CLK = clkx2
ENA = inst4
SYNCH_DATA = result[7]

acc:inst3|ynm[2]~133, LCELL (00F0)
<c> ynm[2]~133
= DATAC & !DATAD

DATAC = result[2]
DATAD = filter.tap1</td></tr>
<tr><td>Primitive</td><td>Format:<<i>primitive name</i>>, <<i>primitive type</i>></td><td>clocks:inst7|Mux~1, OPER (MUX)

md_me:inst18|data[3..3], DFFE</td></tr>
</table>

Table 13–9. Tooltip Information (Part 2 of 2)

Tooltip Format	Description	Example Tooltips
Pin	Format: <*pin name*>, <*pin type*>	pc_clock, INPUT Test_probe, OUTPUT
Connector	Format: <*connector name*>	inst4_CLK
Net	Format: <*net name*>, `fan-out` = <*number of fan-out signals*>	state_m:inst1:decoder_node[2][0], fan-out = 1
Output Port	Format: `fan-out` = <*number of fan-out signals*>	fan-out = 9
Input Port	The information displayed depends on the type of source net. The examples of the tooltips shown represent the following types of source nets: *(1)* Single net *(2)* Individual nets, part of the same bus net *(3)* Combination of different bus nets *(4)* Constant inputs *(5)* Combination of single net and constant input *(6)* Bus net **Source from**—refers to the source net name that connects to the input port. **Destination Index**—refers to the bit(s) at the destination input port to which the source net is connected (not applicable for single nets).	Source from: reset:reset_i:rst *(1)* < Destination Index > | Source from: *(2)* < [11] > | sample~0:OUT1 < [10] > | sample~1:OUT1 < [9] > | sample~2:OUT1 < [8] > | sample~3:OUT1 < [7] > | sample~4:OUT1 < [6] > | sample~5:OUT1 < [5] > | sample~6:OUT1 < [4] > | sample~7:OUT1 < [3] > | sample~8:OUT1 < [2] > | sample~9:OUT1 < [1] > | sample~10:OUT1 < [0] > | sample~11:OUT1 < Destination Index > | Source from: *(3)* < [7..6] > | node2:OUT1 < [5] > | ct[3]:OUT1 < [4] > | node2:OUT1 < [3..2] > | ct[3]:OUT1 < [1] > | node2:OUT1 < [0] > | ct[3]:OUT1 < Destination Index > | Source from: *(4)* < [11..0] > | 12' h000 < Destination Index > | Source from: *(5)* < [2..1] > | 2' h1 < [0] > | always7~2:OUT1 < Destination Index > | Source from: *(6)* < [15..0] > | md_me:inst18:dout[15:0]
State Machine Node	Format: <*node name*>	state_m:inst1|filter.tap1
State Machine Transition Arc	This information is displayed when you hold your mouse over the arrow on the arc representing the transition between two states. Format: (<*equation for transition between states*>)	(!newt)

Radial Menu

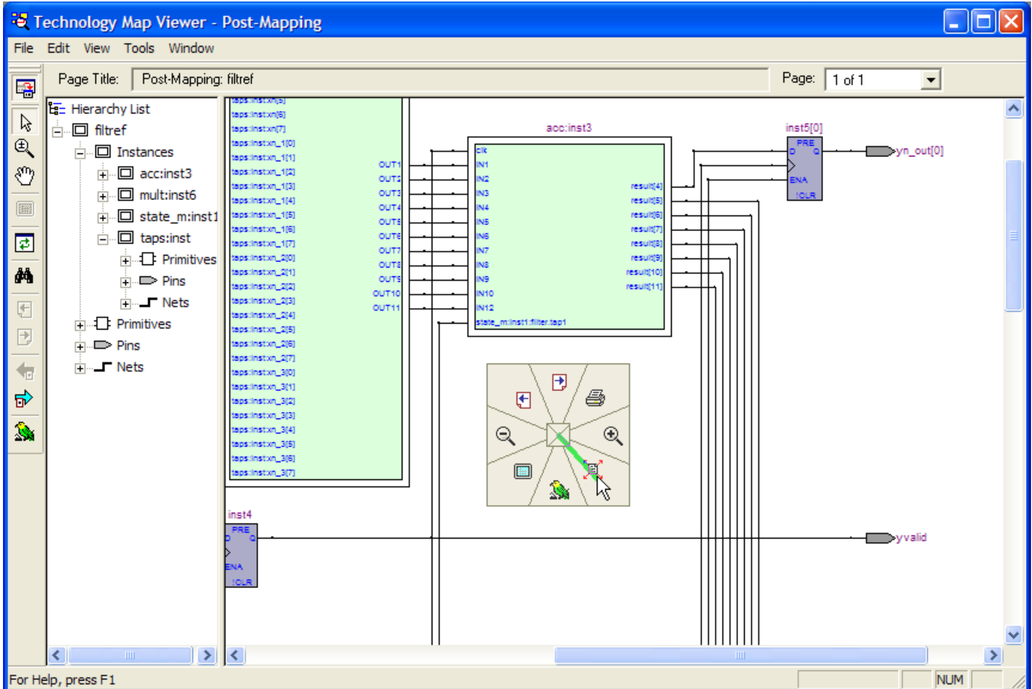
The radial menu is a rectangle-shaped menu with eight commands you can choose from. The menu provides a quick way to perform any of the commands with a single click whenever you are in the schematic view. The radial menu feature is enabled by default.

To open the radial menu, right-click and hold anywhere in the schematic view and wait for the menu to appear. By default, the menu appears after 0.2 seconds. The radial menu appears with the mouse pointer always at the center point. The small rectangle at the center of the menu indicates a non-trigger boundary where no command is started when you click within the rectangle.

To start the desired command, hold down the right mouse button, drag the mouse onto the command, and then release the mouse button. If you decide not to trigger any command after the radial menu appears, press the ESC key or drag the pointer back into the small rectangle and release the mouse button.

Figure 13–27 shows the radial menu.

Figure 13–27. Radial Menu



Enabling and Disabling the Radial Menu

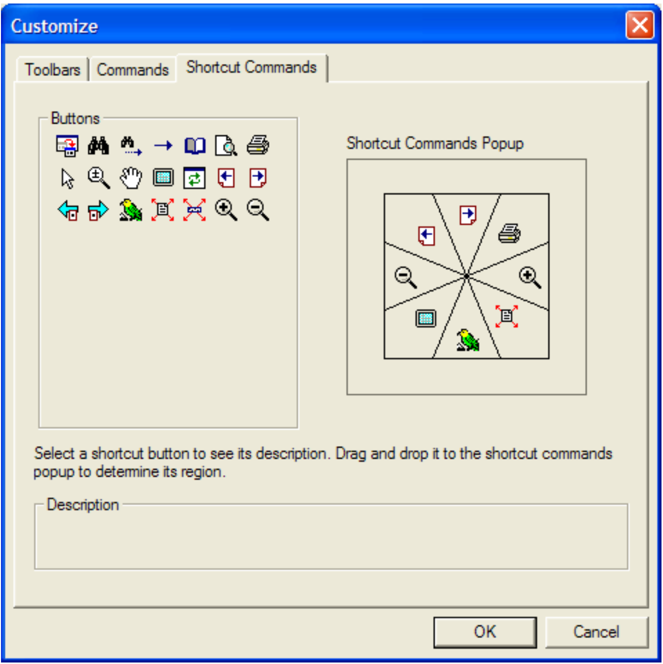
To enable the radial menu feature, on the Tools menu, click **Options**. In the **Options** dialog box, click **Netlist Viewers** and turn on the **Enable Radial Menu** option under **Radial Menu settings**. Turn off the **Enable Radial Menu option** to disable the feature.

Customizing the Shortcut Commands

The radial menu consists of eight commands that are separated into eight different regions. There are 8 out of 24 commands to choose from, and the command can appear more than once. To customize the command list on the menu, first launch the RTL Viewer, the Technology Map Viewer, or the Technology Map Viewer (Post-Mapping). On the Tools menu, click **Customize RTL Viewer**, **Customize Technology Map Viewer**, or **Customize Technology Map Viewer (Post-Mapping)**. On the **Shortcut Commands** tab, drag and drop the icon under **Buttons** into any region under **Shortcut Commands Popup**. You can click on the icon under **Buttons** to see its description.

Figure 13–28 shows the **Shortcut Commands** tab for customizing the radial menu.

Figure 13–28. Shortcut Commands Tab



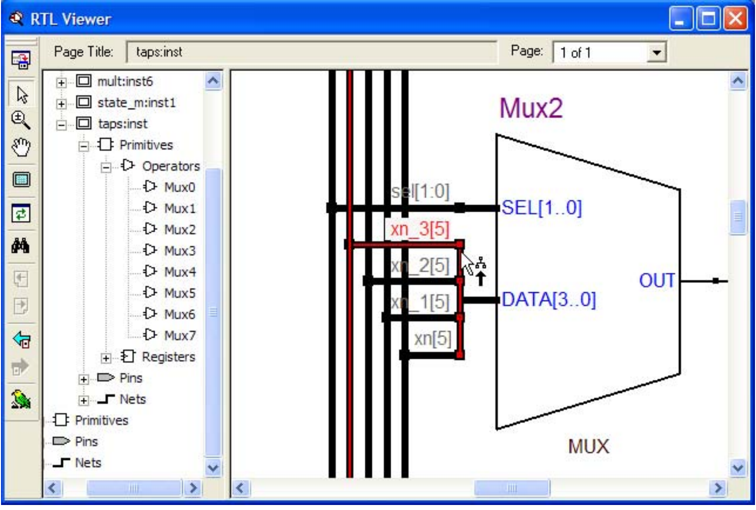
Changing the Time Interval

To change the amount of time you have to wait before the Radial menu appears, on the Tools menu, click **Options**. In the **Options** dialog box, select **Netlist Viewers**. Select the desired time interval in the pull-down list for **Delay showing radial menu for**. The default delay is 0.2 seconds. The Radial menu feature must be enabled before you can change this setting. Refer to "Enabling and Disabling the Radial Menu" on page 13–42 for details about how to enable the Radial menu feature.

Rollover

You can highlight an element and view its name in your schematic using the Rollover feature. When you place your mouse pointer over an object, the object is highlighted and the name is displayed (Figure 13–29). This feature is enabled by default in the netlist viewers. To turn off the Rollover feature, on the Tools menu, click **Options**. In the **Options** dialog box, in the **Category** list, select **Netlist Viewers** and turn off **Enable Rollover**.

Figure 13–29. Rollover in the RTL Viewer and Technology Map Viewer



Displaying Net Names in the Schematic

To see the names of all the nets displayed in your schematic, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and under **Display Settings**, turn on **Show Net Name**. This option is disabled by default. If you turn on this option, the schematic view refreshes automatically to display the net names.

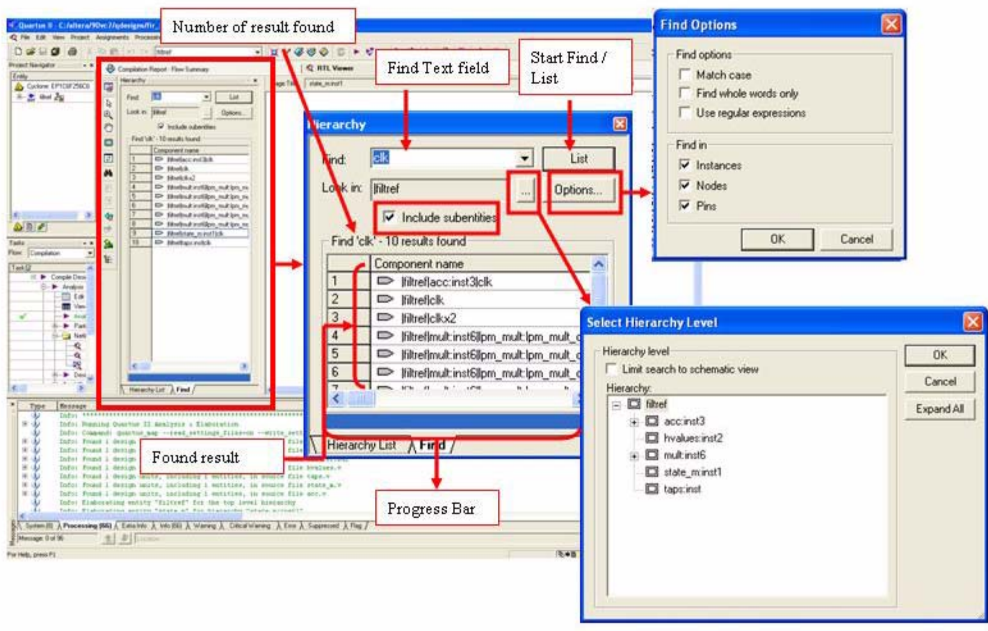
Displaying Node Names in the Schematic

In some designs, nodes have long names that overlap the ports of other symbols in the schematic. To remove the node names from the schematic, on the Tools menu, click **Options**. In the **Category** list, select **Netlist Viewers** and under **Display Settings**, turn off **Show node name**. This option is turned on by default.

Find Command

To open the **Hierarchy** dialog box shown in Figure 13–30, on the Edit menu, click **Find**, or click the **Find** icon in the viewer toolbar, or right-click in the schematic view and click **Find**. There is also a **Hierarchy List** button at the bottom of the toolbar in the viewer window. You can use the **Tab** button to switch between the Find and Hierarchy lists.

Figure 13–30. Hierarchy Dialog Box



You can narrow the range of the search process by setting the following options in the **Hierarchy** dialog box:

- Click on the **Select Hierarchy Level** button to specify the level of the search. In the **Select Hierarchy Level** dialog box, choose the particular instance that you want to search.
- Turn on the **Include subentities** option to include child hierarchies of the parent instance during the search.
- Click **Options** to open the **Find Options** dialog box. Turn on **Instances**, **Nodes**, **Pins**, or any combination of the three to further refine the parameters of the search.

When you click **List**, a progress bar appears below the **Find** box. When the bar is full, the search process is finished.

All results that match the criteria you set are listed in a table. The number of results is shown. When you double-click on an item in the table, the related node is highlighted in red in the schematic view.

For more details about using the **Hierarchy** dialog box, refer to “Finding Nodes in the RTL Viewer and Technology Map Viewer” in the Quartus II Help.

Exporting and Copying a Schematic Image

You can export the schematic view of the RTL Viewer or Technology Map Viewer into various types of image formats. This allows you to include the schematic in project documentation or share it with other project members. The currently supported formats are JPEG File Interchange Format (**.jpg**), Portable Network Graphics (**.png**), Graphics Interchange Format (**.gif**), or Windows Bitmap (**.bmp**). To export the

schematic view, on the File menu, click **Export**. In the **Export** dialog box, type a file name and location and select the desired file type. The default file name is based on the current instance name; the default file type is **.jpg**. However, for pages that use filtering, expanding, or reducing operations, the default name is **Filter**<*number of export operation*>**.jpg**.

Nodes grouped as logic clouds are not shown in the exported or copied schematic image; the logic clouds are shown instead.

You can copy the whole image or only a portion of the image. To copy the full image, on the Edit menu, point to **Copy** and click **Full Image**. To copy a portion of the image, on the Edit menu, point to **Copy** and click **Partial Image**. The cursor changes to a plus sign to indicate that you can draw a box shape. Drag the mouse pointer around the portion of the schematic you want to copy. When you release the mouse button, the partial image is copied to the clipboard.

Occasionally, due to the design size and objects selected, an image is too large to copy to the clipboard. In this case, the Quartus II software displays an error message.

To export or copy a schematic that is too large to copy in one piece, first split the design into multiple pages to export or to copy smaller portions of the design. For information about how to control how much of your design is shown on each schematic page, refer to "Partitioning the Schematic into Pages" on page 13–26. As an alterative, use the Partial Image feature to copy a portion of the image.

The Copy feature is not available on UNIX platforms.

Printing

To print your schematic page, on the File menu, click **Print**. You can print each schematic page onto one full page, or you can print the selected parts of your schematic onto one page with the **Selection** option. Refer to "Partitioning the Schematic into Pages" on page 13–26 to control how much of your design is shown on each schematic page.

Before printing, you can modify the page orientation. On the File menu, click **Page Setup**. Change the page orientation from **Portrait** to **Landscape**, or to the setting that best fits your design. You can also adjust the page margins in the **Page Setup** dialog box.

The hierarchy list in the viewers and the table view of the State Machine Viewer cannot be printed. You can use the State Machine Viewer **Copy** command to copy the table to a text editor and print from the text editor.

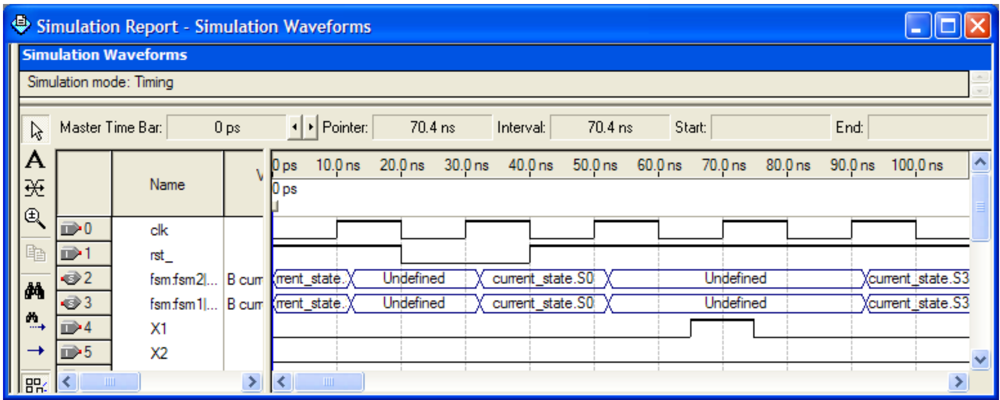
Debugging HDL Code with the State Machine Viewer

This section provides an example of using the State Machine Viewer to help debug HDL code. This example shows how you can use the various features in the netlist viewers to help solve design problems.

Simulation of State Machine Gives Unexpected Results

This section presents a design scenario in which you compiled your design and performed a simulation in the Quartus II Simulator. The simulation result is shown in Figure 13–31 and has unexpected undefined states.

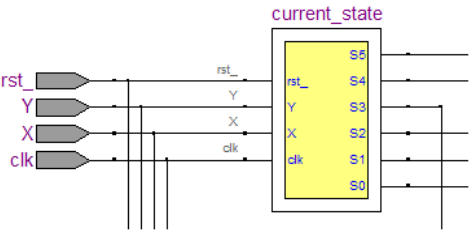
Figure 13–31. Simulation Result Showing Undefined States



To analyze the state machine design in the State Machine Viewer, follow these steps:

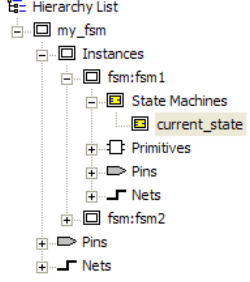
1. Open the State Machine Viewer for the state machine of interest. You can do this in any of the following ways:
   - On the Tools menu, point to **Netlist Viewers** and click **State Machine Viewer**. In the **State Machine** selection box, choose the state machine that you want to view.
   - On the Tools menu, point to **Netlist Viewers** and click **RTL Viewer**. Browse to the hierarchy block that contains the state machine definition and double-click the yellow state machine instance to open the State Machine Viewer (Figure 13–32).You can open the State Machine Viewer using either of two methods:
   - In the schematic view, double-click an instance in the hierarchy to open the lower hierarchy level. You can traverse through the schematic hierarchy in this way to open the schematic page that contains the state machine (Figure 13–32).

Figure 13–32. State Machine Instance in RTL Viewer Schematic View



- In the hierarchy list, click the "**+** " symbol next to **Instances** to open a list of the instances in that hierarchy level of the design. You can traverse down the hierarchy tree in this way to find the instance that contains the state machine. Click on the name of the state machine in the **State Machines** folder (Figure 13–33) to open the appropriate schematic in the schematic view (Figure 13–32).

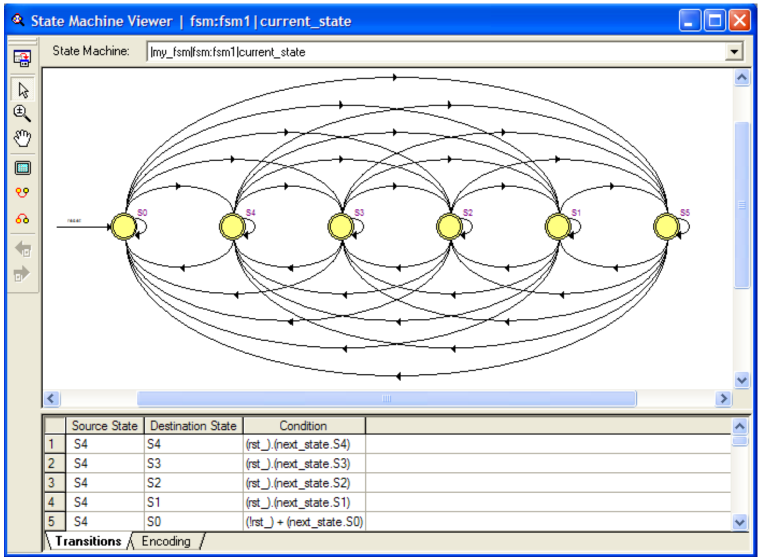
Figure 13–33. State Machine Instance in RTL Viewer Hierarchy List



Double-click the state machine instance (Figure 13–32) to see its state transition diagram in the State Machine Viewer (Figure 13–34).

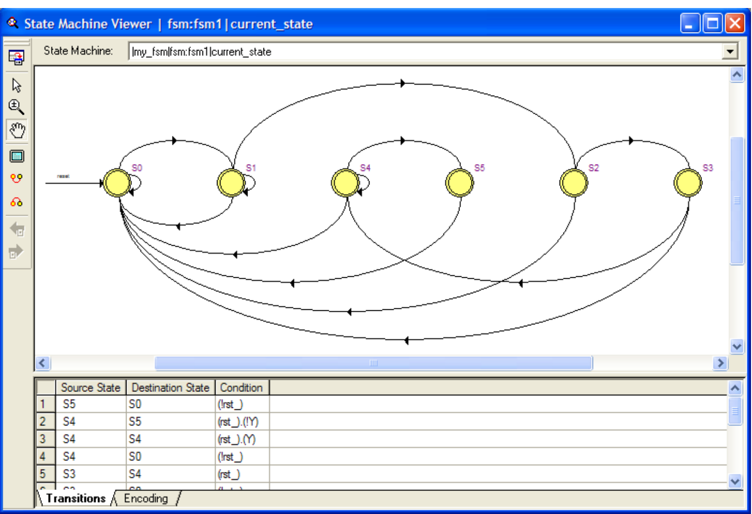
2. You can analyze this state machine instance using the state machine diagram, transition table, and encoding table. Clearly something is wrong with the state machine because every state has a transition to every other state (Figure 13–34). After inspecting the state machine behavior, you determine that in this scenario, the designer forgot to create default assignments for the next state (that is, `next_state = current_state` if the conditions are not met).

Figure 13–34. State Machine Viewer Showing Incorrect Transitions



3. After fixing the error in the HDL code, recompile the design and repeat steps 1-2 to view the new state machine diagram and transition table (shown in Figure 13–35), and check that the state transitions now occur correctly.

Figure 13–35. State Machine Viewer Showing Correct Transitions



4. Perform a new simulation, as shown in Figure 13–36, and verify that the state machine performs as expected.

Figure 13–36. Simulation Result Showing Correct States

Conclusion

The Quartus II RTL Viewer, State Machine Viewer, and Technology Map Viewer allow you to explore and analyze your initial synthesis netlist, post-synthesis netlist, or post-fitting and physical synthesis netlist. The viewers provide a number of features in the hierarchy list and schematic view to help you quickly trace through your netlist and find specific hierarchies or nodes of interest. These capabilities can help you debug, optimize, or constrain your design more efficiently to increase your productivity.

Document Revision History

Table 13–10 shows the revision history for this chapter.

Table 13–10. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Chapter 13 was formerly Chapter 12 in version 8.1.0 ■ Updated Figure 13–2, Figure 13–3, Figure 13–4, Figure 13–14, and Figure 13–30 ■ Added "Enable or Disable the Auto Hierarchy List" on page 13–15 ■ Updated "Find Command" on page 13–44	Updated for the Quartus II software version 9.0 release.

Table 13–10. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
November 2008 v8.1.0	Changed page size to 8.5” × 11”	—
May 2008 v8.0.0	■ Added Arria GX support ■ Updated operator symbols ■ Updated information about the radial menu feature ■ Updated zooming feature ■ Updated information about probing from schematic to SignalTap II Analyzer ■ Updated constant signal information ■ Added .png and .gif to the list of supported image file formats ■ Updated several figures and tables ■ Added new sections “Enabling and Disabling the Radial Menu”, “Changing the Time Interval”, “Changing the Constant Signal Value Formatting”, “Logic Clouds in the RTL Viewer”, “Logic Clouds in the Technology Map Viewer”, “Manually Group and Ungroup Logic Clouds”, “Customizing the Shortcut Commands” ■ Renamed several sections ■ Removed section “Customizing the Radial Menu” ■ Moved section “Grouping Combinational Logic into Logic Clouds” ■ Updated document content based on the Quartus II software version 8.0	Updated for Quartus II software version 8.0.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 9.0.

How to Contact Altera

For the most up-to-date information about Altera products, see the following table.

Contact *(Note 1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note:

(1) You can also contact your local Altera sales office or sales representative.

Third-Party Software Product Information

Third-party software products described in this handbook are not Altera products, are licensed by Altera from third parties, and are subject to change without notice. Updates to these third-party software products may not be concurrent with Quartus II software releases. Altera has assumed responsibility for the selection of such third-party software products and its use in the Quartus II 9.0 software release. To the extent that the software products described in this handbook are derived from third-party software, no third party warrants the software, assumes any liability regarding use of the software, or undertakes to furnish you any support or information relating to the software. EXCEPT AS EXPRESSLY SET FORTH IN THE APPLICABLE ALTERA PROGRAM LICENSE SUBSCRIPTION AGREEMENT UNDER WHICH THIS SOFTWARE WAS PROVDED TO YOU, ALTERA AND THIRD-PARTY LICENSORS DISCLAIM ALL WARRANTIES WITH RESPECT TO THE USE OF SUCH THIRD-PARTY SOFTWARE CODE OR DOCUMENTATION IN THE SOFTWARE, INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT. For more information, including the latest available version of specific third-party software products, refer to the documentation for the software in question.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, file names, file name extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>*.**pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
(feet)	The feet direct you to more information on a particular topic.

Quartus II Handbook Version 9.0

Volume 2: Design Implementation and Optimization

101 Innovation Drive
San Jose, CA 95134
www.altera.com

QII5V2-9.0

Copyright © 2009 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

NSAI

I.S. EN ISO 9001

Contents

Chapter Revision Dates xix

Section I. Scripting and Constraint Entry

Chapter 1. Assignment Editor

Introduction 1-1
Overview of the Assignment Editor 1-1
Dynamic Syntax Checking 1-2
Viewing and Saving Assignments in the Assignment Editor 1-2
User Interface 1-3
Category Bar 1-4
Node Filter Bar 1-4
Information Bar 1-4
Edit Bar 1-4
Assignment Spreadsheet 1-5
Toolbar 1-5
Navigating the Assignment Editor Spreadsheet 1-5
Entering Values into the Spreadsheet 1-5
Wildcards 1-6
Assignment Groups 1-7
Customizing the Spreadsheet Columns 1-7
Exporting and Importing Assignments 1-7
Exporting Assignments 1-8
Importing Assignments 1-9
Creating Timing Constraints Using the Assignment Editor 1-9
Tcl Interface 1-10
Probing to Source Design Files and Other Quartus II Windows 1-10
Probing to the Assignment Editor from Other Quartus II Windows 1-11
Conclusion 1-11
Referenced Documents 1-11
Document Revision History 1-12

Chapter 2. Command-Line Scripting

Introduction 2-1
The Benefits of Command-Line Executables 2-1
Introductory Example 2-2
Command-Line Executables 2-3
Command-Line Scripting Help 2-5
Command-Line Option Details 2-6
Option Precedence 2-7
Design Flow 2-9
Compilation with quartus_sh --flow 2-9
Text-Based Report Files 2-9
Makefile Implementation 2-11
The MegaWizard Plug-In Manager 2-13
Command-Line Support 2-14
Module and Wizard Names 2-15
Ports and Parameters 2-15

Invalid Configurations . . . 2-16
Strategies to Determine Port and Parameter Values . . . 2-16
Optional Files . . . 2-16
Parameter File . . . 2-18
Working Directory . . . 2-18
Variation File Name . . . 2-18
Command-Line Scripting Examples . . . 2-19
Create a Project and Apply Constraints . . . 2-19
Check Design File Syntax . . . 2-20
Create a Project and Synthesize a Netlist Using Netlist Optimizations . . . 2-20
Archive and Restore Projects . . . 2-21
Perform I/O Assignment Analysis . . . 2-21
Update Memory Contents without Recompiling . . . 2-21
Create a Compressed Configuration File . . . 2-22
Fit a Design as Quickly as Possible . . . 2-22
Fit a Design Using Multiple Seeds . . . 2-23
The QFlow Script . . . 2-24
Referenced Documents . . . 2-24
Document Revision History . . . 2-25

Chapter 3. Tcl Scripting

Introduction . . . 3-1
What is Tcl? . . . 3-2
Quartus II Tcl Packages . . . 3-2
Loading Packages . . . 3-4
Quartus II Tcl API Help . . . 3-4
Executables Supporting Tcl . . . 3-7
Command-Line Options: -t, -s, and --tcl_eval . . . 3-7
Run a Tcl Script . . . 3-7
Interactive Shell Mode . . . 3-8
Evaluate as Tcl . . . 3-8
Using the Quartus II Tcl Console Window . . . 3-8
End-to-End Design Flows . . . 3-9
Creating Projects and Making Assignments . . . 3-10
HardCopy Device Design . . . 3-10
Using LogicLock Regions . . . 3-11
Compiling Designs . . . 3-14
The flow Package . . . 3-14
Compile All Revisions . . . 3-15
Reporting . . . 3-15
Creating .csv Files for Excel . . . 3-17
Timing Analysis . . . 3-18
Classic Timing Analysis . . . 3-18
Advanced Classic Timing Analysis . . . 3-20
TimeQuest Timing Analysis . . . 3-22
TimeQuest Scripting . . . 3-22
Automating Script Execution . . . 3-22
Making the Assignment . . . 3-23
Script Execution . . . 3-24
Execution Example . . . 3-24
Controlling Processing . . . 3-25
Displaying Messages . . . 3-25
Other Scripting Features . . . 3-25
Natural Bus Naming . . . 3-25

Short Option Names . . . 3-26
Using Collection Commands . . . 3-26
The foreach_in_collection Command . . . 3-26
The get_collection_size Command . . . 3-26
Using the post_message Command . . . 3-27
Accessing Command-Line Arguments . . . 3-27
Using the cmdline Package . . . 3-28
Using the Quartus II Tcl Shell in Interactive Mode . . . 3-29
Quartus II Legacy Tcl Support . . . 3-32
Using the tclsh Shell . . . 3-32
Tcl Scripting Basics . . . 3-33
Hello World Example . . . 3-33
Variables . . . 3-33
Substitutions . . . 3-33
Variable Value Substitution . . . 3-34
Nested Command Substitution . . . 3-34
Backlash Substitution . . . 3-34
Arithmetic . . . 3-34
Lists . . . 3-34
Arrays . . . 3-35
Control Structures . . . 3-36
Procedures . . . 3-37
File I/O . . . 3-37
Syntax and Comments . . . 3-38
External References . . . 3-39
Referenced Documents . . . 3-39
Document Revision History . . . 3-39

Chapter 4. Managing Quartus II Projects
Introduction . . . 4-1
Creating a New Project . . . 4-2
Using Revisions with Your Design . . . 4-2
Creating and Deleting Revisions . . . 4-3
Create a Revision . . . 4-3
Delete a Revision . . . 4-4
Compare Revisions . . . 4-4
Creating New Copies of Your Design . . . 4-5
Archiving Projects with the Quartus II Archive Project Feature . . . 4-6
Archive a Project . . . 4-6
Restore an Archived Project . . . 4-8
Version Compatible Databases . . . 4-8
Migrate to a New Version . . . 4-9
Save the Database in a Version Compatible Format . . . 4-9
Quartus II Project Platform Migration . . . 4-9
Filenames and Hierarchies . . . 4-10
Relative Paths . . . 4-11
Specifying Libraries . . . 4-11
Specifying User Libraries . . . 4-12
Specifying Global Libraries . . . 4-12
Quartus II Search Path Precedence Rules . . . 4-12
Quartus II-Generated Files for Third-Party EDA Tools . . . 4-13
Migrating Database Files between Platforms . . . 4-14
Working with Messages . . . 4-14
Messages Window . . . 4-15

Hiding Messages 4-16
Message Suppression 4-17
Message Suppression Methods 4-18
Message Supression Details and Limitations 4-18
Message Suppression Manager 4-19
Suppressible Messages 4-19
Suppression Rules 4-19
Suppressed Messages 4-20
Quartus II Settings File 4-21
QSF Format Preservation 4-21
Quartus II Default Settings File 4-22
Scripting Support 4-22
Managing Revisions 4-23
Creating Revisions 4-23
Setting the Current Revision 4-23
Getting a List of Revisions 4-23
Deleting Revisions 4-23
Archiving Projects 4-23
Restoring Archived Projects 4-24
Importing and Exporting Version Compatible Databases 4-24
Specifying Libraries Using Scripts 4-25
Conclusion 4-25
Referenced Documents 4-26
Document Revision History 4-26

Section II. I/O and PCB Tools

Chapter 5. I/O Management

Introduction 5-1
Understanding Altera FPGA Pin Terminology 5-2
Package Pins 5-2
Pads 5-3
I/O Banks 5-3
VREF Groups 5-4
I/O Planning Overview 5-5
Device Selection 5-7
Early I/O Planning Using the Pin Planner 5-8
Create or Import a Megafunction or IP MegaCore Variation from the Pin Planner 5-9
Configure Nodes 5-10
Configure Megafunction Nodes 5-10
Configure User Nodes 5-12
I/O Analysis for Designs with Pins Only 5-14
Importing and Exporting Pin Assignments 5-16
Spreadsheets and .csv Files 5-16
Quartus II Settings Files 5-16
Tcl Script 5-17
FPGA Xchange File 5-17
.pin File 5-18
Creating Pin-Related Assignments 5-19
Using the Pin Planner 5-20
Groups List 5-21
All Pins List 5-24
Pad View Window 5-26

Package View 5-28
Pin Migration View 5-30
Using the Pin Finder to Find Compatible Pin Locations 5-33
SSN Visualization View 5-34
Creating Reserved Pin Assignments 5-34
Creating Pin Location Assignments 5-35
Changing Pin Locations 5-42
Swapping Pin Locations 5-43
Show I/O Banks 5-44
Show VREF Groups 5-45
Show Edges 5-47
Show DQ/DQS Pins 5-48
Displaying and Accepting Fitter Placements 5-49
Assignment Editor 5-50
Setting Pin Locations from the Device Pin Number List 5-50
Setting Pin Locations from the Design Signal Name List 5-51
Tcl Scripts 5-53
Chip Planner or Timing Closure Floorplan 5-53
Using HDL 5-54
Synthesis Attributes 5-55
Using Low-Level I/O Primitives 5-56
Validating Pin Assignments 5-56
Using the Live I/O Check Feature to Validate Pin Assignments 5-57
Using I/O Assignment Analysis to Validate Pin Assignments 5-59
I/O Assignment Analysis Design Flows 5-59
Inputs for I/O Assignment Analysis 5-65
Understanding the I/O Assignment Analysis Report and Messages 5-67
Scripting Support 5-72
Validating Pin Assignments after Full Compilation 5-74
Accepting Fitter Placements—Back-Annotating Assignments 5-74
I/O Timing Analysis 5-75
I/O Timing and Power with Capacitive Loading 5-77
Advanced I/O Timing in the Quartus II Software 5-78
Enabling and Configuring Advanced I/O Timing 5-78
Define Overall Board Trace Models 5-78
Customize the Board Trace Model in the Pin Planner 5-80
Create Signal Integrity Result Reports 5-82
Incorporating PCB Design Tools 5-83
Conclusion 5-84
Referenced Documents 5-84
Document Revision History 5-85

Chapter 6. Simultaneous Switching Noise (SSN) Analysis and Optimizations

Introduction 6-1
Definitions 6-1
Understanding SSN and Its Effects 6-2
SSN Estimation Tools from Altera 6-5
Design Factors Affecting SSN Results 6-5
Using the SSN Analyzer in the Quartus II Software 6-5
Tools Overview 6-6
I/O Standards Supported in the Quartus II SSN Analyzer 6-6
Tool Inputs 6-7
Board Trace Models 6-7
PCB Layers and PCB Layer Thickness 6-8

Signal Breakout Layers 6-9
I/O Assignments 6-10
Automatic Aggressor Identification 6-10
Group Assignments 6-10
Running the SSN Analyzer 6-11
Understanding the SSN Reports 6-12
Settings Report 6-12
Summary Report 6-12
Input Pins Report 6-12
Output Pins Report 6-12
Confidence Metric Details Report 6-13
Unanalyzed Pins Report 6-13
Visualizing SSN in the Pin Planner 6-13
Invoking the SSN Map 6-13
SSN Analyzer Usage Models 6-14
Early Pin-Out SSN Analysis 6-15
Early Pin-Out SSN Analysis Using the Early SSN Estimator Spreadsheet 6-15
Early Pin-Out SSN Analysis Using the Quartus II SSN Analyzer 6-15
SSN Aware Fitter 6-17
Default Assignments Used in Early SSN Analysis 6-17
Final Pin-Out Analysis: Fully Constrained Design SSN Analysis 6-18
Scripting Support 6-18
Run Time Considerations in SSN Analysis 6-18
Running SSN Analyzer with Multi-CPU machines 6-19
Running the Complete Design for SSN Analysis after I/O Assignment Analysis 6-19
Running the Complete Design for SSN Analysis after a Full Fit 6-19
Making ECO Changes and Rerunning SSN Analysis 6-19
Running SSN Analysis for One I/O Bank 6-19
SSN Optimization 6-20
Back-Annotating the Fitter Results 6-22
SSN Optimization in Your System 6-22
Conclusion 6-22
Referenced Documents 6-23
Document Revision History 6-23

Chapter 7. Signal Integrity Analysis with Third-Party Tools

Introduction 7-1
I/O Model Selection: IBIS or HSPICE 7-3
FPGA to Board Signal Integrity Analysis Flow 7-3
Create I/O and Board Trace Model Assignments 7-5
Output File Generation 7-6
Customize the Output Files 7-6
Set Up and Run Simulations in Third-Party Tools 7-7
Interpret Simulation Results 7-7
Simulation with IBIS Models 7-7
Elements of an IBIS Model 7-7
Creating Accurate IBIS Models 7-8
Download IBIS Models 7-8
Generate Custom IBIS Models with the IBIS Writer 7-9
Design Simulation Using the Mentor Graphics HyperLynx Software 7-10
Configuring LineSim to Use Altera IBIS Models 7-12
Integrating Altera IBIS Models into LineSim Simulations 7-14
Running and Interpreting LineSim Simulations 7-15
Simulation with HSPICE Models 7-17

Supported Devices and Signaling . . . 7-17
Accessing HSPICE Simulation Kits . . . 7-18
The Double Counting Problem in HSPICE Simulations . . . 7-18
Defining the Double Counting Problem . . . 7-18
The Solution to Double Counting . . . 7-19
HSPICE Writer Tool Flow . . . 7-20
Applying I/O Assignments . . . 7-20
Enabling HSPICE Writer . . . 7-20
Enabling HSPICE Writer Using Assignments . . . 7-21
Naming Conventions for HSPICE Files . . . 7-21
Invoking HSPICE Writer . . . 7-22
Invoking HSPICE Writer from the Command Line . . . 7-22
Customizing Automatically Generated HSPICE Decks . . . 7-22
Running an HSPICE Simulation . . . 7-23
Interpreting the Results of an Output Simulation . . . 7-24
Interpreting the Results of an Input Simulation . . . 7-24
Viewing and Interpreting Tabular Simulation Results . . . 7-24
Viewing Graphical Simulation Results . . . 7-24
Making Design Adjustments Based on HSPICE Simulations . . . 7-26
Sample Input for I/O HSPICE Simulation Deck . . . 7-28
Header Comment . . . 7-28
Simulation Conditions . . . 7-29
Simulation Options . . . 7-29
Constant Definition . . . 7-30
Buffer Netlist . . . 7-30
Drive Strength . . . 7-31
I/O Buffer Instantiation . . . 7-31
Board Trace and Termination . . . 7-32
Stimulus Model . . . 7-32
Simulation Analysis . . . 7-32
Sample Output for I/O HSPICE Simulation Deck . . . 7-32
Header Comment . . . 7-33
Simulation Conditions . . . 7-34
Simulation Options . . . 7-35
Constraint Definition . . . 7-35
I/O Buffer Netlist . . . 7-36
Drive Strength . . . 7-36
Slew Rate and Delay Chain . . . 7-37
I/O Buffer Instantiation . . . 7-37
Board and Trace Termination . . . 7-38
Double-Counting Compensation Circuitry . . . 7-38
Simulation Analysis . . . 7-39
Advanced Topics . . . 7-40
PVT Simulations . . . 7-40
Hold Time Analysis . . . 7-41
I/O Voltage Variations . . . 7-41
Correlation Report . . . 7-41
Conclusion . . . 7-41
Referenced Documents . . . 7-42
Document Revision History . . . 7-42

Chapter 8. Mentor Graphics PCB Design Tools Support
Introduction . . . 8-1
FPGA-to-PCB Design Flow . . . 8-2

Setting Up the Quartus II Software 8-4
Generating Pin-Out Files 8-6
Generating FPGA Xchange Files 8-6
Creating a Backup Quartus II Settings File 8-6
FPGA-to-Board Integration with the I/O Designer Software 8-7
I/O Designer Database Wizard 8-9
Updating Pin Assignments from the Quartus II Software 8-13
Sending Pin Assignment Changes to the Quartus II Software 8-16
Protecting Assignments in the Quartus II Software 8-18
Generating Symbols for the DxDesigner Software 8-18
Setting Up the I/O Designer Software to Work with the DxDesigner Software 8-19
Create Symbols with the Symbol Wizard 8-20
Export Symbols to the DxDesigner Software 8-22
Scripting Support 8-22
FPGA-to-Board Integration with the DxDesigner Software 8-23
DxDesigner Project Settings 8-24
DxDesigner Symbol Wizard 8-25
Conclusion 8-28
Referenced Documents 8-28
Document Revision History 8-29

Chapter 9. Cadence PCB Design Tools Support

Introduction 9-1
Product Comparison 9-2
FPGA-to-PCB Design Flow 9-2
Setting Up the Quartus II Software 9-4
Generating .pin Files 9-5
FPGA-to-Board Integration with the Cadence Allegro Design Entry HDL Software 9-5
Symbol Creation 9-5
Allegro PCB Librarian Part Developer 9-6
Instantiating the Symbol in the Cadence Allegro Design Entry HDL Software 9-14
FPGA-to-Board Integration with Allegro Design Entry CIS 9-14
Allegro Design Entry CIS Project Creation 9-15
Generate Part 9-16
Split Part 9-17
Instantiate Symbol in Design Entry CIS Schematic 9-19
Altera Libraries for Design Entry CIS 9-20
Conclusion 9-22
Referenced Document 9-22
Document Revision History 9-22

Section III. Area, Timing and Power Optimization

Introduction III-1
Physical Implementation III-1
Trade Offs and Limitations III-1
Preserving Results and Enabling Teamwork III-2
Reducing Area III-2
Reducing Critical Path Delay III-2
Reduce Power Consumption III-3
Reducing Runtime III-3
Using Quartus II Tools III-3
Design Analysis III-3
Advisors III-4
Design Space Explorer III-4

Further Reading III-4

Chapter 10. Area and Timing Optimization

Introduction 10-1
Optimizing Your Design 10-2
Initial Compilation: Required Settings 10-3
Device Settings 10-3
I/O Assignments 10-3
Timing Requirement Settings 10-4
Timing Constraint Check—Report Unconstrained Paths 10-5
Device Migration Settings 10-5
Partitions and Floorplan Assignments for Incremental Compilation 10-6
Initial Compilation: Optional Settings 10-6
Design Assistant 10-7
Smart Compilation Setting 10-7
Early Timing Estimation 10-7
Optimize Hold Timing 10-8
Asynchronous Control Signal Recovery/Removal Analysis 10-9
Limit to One Fitting Attempt 10-9
Optimize Multi-Corner Timing 10-10
Fitter Effort Setting 10-10
Auto Fit 10-10
Standard Fit 10-11
Fast Fit 10-11
Design Analysis 10-11
Error and Warning Messages 10-12
Ignored Timing Assignments 10-12
Resource Utilization 10-12
I/O Timing (Including t_{PD}) 10-13
Register-to-Register Timing 10-14
Timing Analysis with the TimeQuest Timing Analyzer 10-14
Timing Analysis with the Classic Timing Analyzer 10-15
Tips for Analyzing Failing Paths 10-17
Tips for Analyzing Failing Clock Paths that Cross Clock Domains 10-17
Global Routing Resources 10-18
Compilation Time 10-19
Resource Utilization Optimization Techniques (LUT-Based Devices) 10-19
Using the Resource Optimization Advisor 10-19
Resolving Resource Utilization Issues Summary 10-20
I/O Pin Utilization or Placement 10-20
Use I/O Assignment Analysis 10-20
Modify Pin Assignments or Choose a Larger Package 10-20
Logic Utilization or Placement 10-21
Optimize Synthesis for Area, Not Speed 10-21
Restructure Multiplexers 10-21
Perform WYSIWYG Resynthesis with Balanced or Area Setting 10-22
Use Register Packing 10-22
Remove Fitter Constraints 10-24
Change State Machine Encoding 10-25
Flatten the Hierarchy During Synthesis 10-25
Retarget Memory Blocks 10-25
Use Physical Synthesis Options to Reduce Area 10-26
Retarget or Balance DSP Blocks 10-27
Optimize Source Code 10-28

Use a Larger Device 10-28
Routing 10-29
Set Auto Register Packing to Sparse or Sparse Auto 10-29
Set Fitter Aggressive Routability Optimizations to Always 10-29
Increase Placement Effort Multiplier 10-30
Increase Router Effort Multiplier 10-30
Remove Fitter Constraints 10-30
Optimize Synthesis for Area, Not Speed 10-31
Optimize Source Code 10-32
Use a Larger Device 10-32
Timing Optimization Techniques (LUT-Based Devices) 10-32
Timing Optimization Advisor 10-32
Metastability Analysis and Optimization Techniques 10-32
I/O Timing Optimization 10-33
Improving Setup and Clock-to-Output Times Summary 10-34
Timing-Driven Compilation 10-35
Fast Input, Output, and Output Enable Registers 10-36
Programmable Delays 10-36
Use PLLs to Shift Clock Edges 10-39
Use Fast Regional Clock Networks and Regional Clocks Networks 10-39
Change How Hold Times are Optimized for MAX II Devices 10-40
Register-to-Register Timing Optimization Techniques (LUT-Based Devices) 10-40
Improving Register-to-Register Timing Summary 10-40
Physical Synthesis Optimizations 10-41
Turn Off Extra-Effort Power Optimization Settings 10-44
Optimize Synthesis for Speed, Not Area 10-44
Flatten the Hierarchy During Synthesis 10-45
Set the Synthesis Effort to High 10-45
Change State Machine Encoding 10-45
Duplicate Logic for Fan-Out Control 10-45
Prevent Shift Register Inference 10-46
Use Other Synthesis Options Available in Your Synthesis Tool 10-46
Fitter Seed 10-47
Set Maximum Router Timing Optimization Level 10-47
Enable Beneficial Skew Optimization 10-48
Optimize Source Code 10-48
LogicLock Assignments 10-49
Hierarchy Assignments 10-50
Path Assignments 10-50
Location Assignments and Back-Annotation 10-51
Custom Regions 10-52
Back-Annotation and Manual Placement 10-52
Optimizing Placement for Stratix, Stratix II, Arria GX, and Cyclone II Devices 10-54
Optimizing Placement for Cyclone Devices 10-54
Optimizing Placement for APEX II and APEX 20KE/C Devices 10-54
Resource Utilization Optimization Techniques (Macrocell-Based CPLDs) 10-55
Use Dedicated Inputs for Global Control Signals 10-55
Reserve Device Resources 10-56
Pin Assignment Guidelines and Procedures 10-56
Control Signal Pin Assignments 10-56
Output Enable Pin Assignments 10-56
Estimate Fan-In When Assigning Output Pins 10-57
Outputs Using Parallel Expander Pin Assignments 10-57
Resolving Resource Utilization Problems 10-58

Resolving Macrocell Usage Issues 10-58
Resolving Routing Issues 10-58
Using LCELL Buffers to Reduce Required Resources 10-59
Timing Optimization Techniques (Macrocell-Based CPLDs) 10-60
Improving Setup Time 10-61
Improving Clock-to-Output Time 10-61
Improving Propagation Delay (t_{PD}) 10-62
Improving Maximum Frequency (f_{MAX}) 10-63
Optimizing Source Code—Pipelining for Complex Register Logic 10-64
Compilation-Time Optimization Techniques 10-64
Incremental Compilation 10-64
Use Multiple Processors for Parallel Compilation 10-65
Reduce Synthesis Time and Synthesis Netlist Optimization Time 10-66
Synthesis Netlist Optimizations 10-67
Check Early Timing Estimation before Fitting 10-67
Reduce Placement Time 10-67
Fitter Effort Setting 10-67
Placement Effort Multiplier Settings 10-68
Final Placement Optimization Levels 10-68
Physical Synthesis Effort Settings 10-68
Limit to One Fitting Attempt 10-68
Preserving Placement, Incremental Compilation, and LogicLock Regions 10-69
Reduce Routing Time 10-69
Identify Routing Congestion in the Chip Planner 10-69
Identify Routing Congestion in the Timing Closure Floorplan for Legacy Devices 10-69
Placement Effort Multiplier Setting 10-70
Preserve Routing Incremental Compilation and LogicLock Regions 10-70
Setting Process Priority 10-70
Other Optimization Resources 10-70
Design Space Explorer 10-70
Other Optimization Advisors 10-71
Scripting Support 10-71
Initial Compilation Settings 10-71
Resource Utilization Optimization Techniques (LUT-Based Devices) 10-72
I/O Timing Optimization Techniques (LUT-Based Devices) 10-73
Register-to-Register Timing Optimization Techniques (LUT-Based Devices) 10-74
Duplicate Logic for Fan-Out Control 10-74
Conclusion 10-74
Referenced Documents 10-75
Document Revision History 10-76

Chapter 11. Power Optimization

Introduction 11-1
Power Dissipation 11-2
Design Space Explorer 11-3
Power-Driven Compilation 11-4
Power-Driven Synthesis 11-4
Power-Driven Fitter 11-8
Recommended Flow for Power-Driven Compilation 11-10
Area-Driven Synthesis 11-10
Gate-Level Register Retiming 11-11
Design Guidelines 11-12
Clock Power Management 11-12
LAB-Wide Clock Enable Example 11-13

Reducing Memory Power Consumption . . . 11-14
Memory Power Reduction Example . . . 11-16
Pipelining and Retiming . . . 11-17
Architectural Optimization . . . 11-18
I/O Power Guidelines . . . 11-19
Dynamically-Controlled On-Chip Terminations . . . 11-20
Power Optimization Advisor . . . 11-21
Power Optimization Advisor Example . . . 11-21
Conclusion . . . 11-24
Referenced Documents . . . 11-24
Document Revision History . . . 11-25

Chapter 12. Analyzing and Optimizing the Design Floorplan
Introduction . . . 12-1
Chip Planner Overview . . . 12-2
Starting the Chip Planner . . . 12-3
The Chip Planner Toolbar . . . 12-3
Chip Planner Tasks and Layers . . . 12-4
LogicLock Regions . . . 12-6
Creating LogicLock Regions . . . 12-7
Placing LogicLock Regions . . . 12-7
Placing Device Features into LogicLock Regions . . . 12-8
LogicLock Regions Window . . . 12-8
Hierarchical (Parent and Child) LogicLock Regions . . . 12-9
Reserved LogicLock Region . . . 12-10
Creating Non-Rectangular LogicLock Regions . . . 12-10
Examples of Non-Rectangular LogicLock Regions Using Reserved Property . . . 12-11
Example 1: Creating an L-Shaped Region . . . 12-11
Example 2: Region with Disjoint Areas . . . 12-13
Excluded Resources . . . 12-14
Additional Quartus II LogicLock Design Features . . . 12-15
Tooltips . . . 12-15
Show Critical Paths . . . 12-15
Analysis and Synthesis Resource Utilization by Entity . . . 12-15
Path-Based Assignments . . . 12-15
Quartus II Revisions Feature . . . 12-16
LogicLock Assignment Precedence . . . 12-16
Virtual Pins . . . 12-17
Using LogicLock Regions in the Chip Planner . . . 12-18
Assigning LogicLock Region Content . . . 12-18
Creating LogicLock Regions with the Chip Planner . . . 12-19
Viewing Connections Between LogicLock Regions in the Chip Planner . . . 12-19
Design Floorplan Analysis Using the Chip Planner . . . 12-19
Chip Planner Floorplan Views . . . 12-20
First-Level View . . . 12-20
Second-Level View . . . 12-21
Third-Level View . . . 12-21
Bird's Eye View . . . 12-22
Viewing Architecture-Specific Design Information . . . 12-24
Viewing Available Clock Networks in the Device . . . 12-24
Viewing Critical Paths . . . 12-26
Viewing Physical Timing Estimates . . . 12-27
Viewing Routing Congestion . . . 12-29
Viewing I/O Banks . . . 12-30

Generating Fan-In and Fan-Out Connections 12-31
Generating Immediate Fan-In and Fan-Out Connections 12-32
Highlight Routing 12-33
Show Delays 12-34
Exploring Paths in the Chip Planner 12-35
Locate Path from the Timing Analysis Report to the Chip Planner 12-35
Analyzing Connections for a Path 12-36
Viewing Assignments in the Chip Planner 12-37
Viewing Routing Channels for a Path in the Chip Planner 12-37
Cell Delay Table 12-38
Viewing High-Speed and Low Power Tiles in Stratix III Devices in the Chip Planner 12-39
Design Analysis Using the Timing Closure Floorplan 12-40
Timing Closure Floorplan Views 12-41
Field View 12-42
Other Views 12-42
Viewing Assignments 12-42
Viewing Critical Paths 12-43
Physical Timing Estimates 12-46
Viewing Routing Congestion 12-47
Timing Closure Floorplan View 12-48
LogicLock Regions in the Timing Closure Floorplan 12-49
Creating LogicLock Regions in the Timing Closure Floorplan Editor 12-49
Using Drag and Drop to Place Logic 12-50
Rubber Banding 12-50
Show Connection Count 12-50
Analyzing LogicLock Region Connectivity Using the Timing Closure Floorplan 12-50
Using LogicLock Methodology for Older Device Families 12-52
The Quartus II LogicLock Methodology 12-52
Improving Design Performance 12-53
LogicLock Restrictions 12-53
Preserving Timing Results Using the LogicLock Flow 12-54
Back-Annotating Routing Information 12-55
Back-Annotating LogicLock Regions 12-56
Scripting Support 12-57
Initializing and Uninitializing a LogicLock Region 12-57
Creating or Modifying LogicLock Regions 12-57
Obtaining LogicLock Region Properties 12-58
Assigning LogicLock Region Content 12-58
Prevent Further Netlist Optimization 12-58
Save a Node-Level Netlist for the Entire Design into a Persistent Source File 12-58
Setting LogicLock Assignment Priority 12-59
Assigning Virtual Pins 12-59
Back-Annotating LogicLock Regions 12-59
Conclusion 12-59
Referenced Documents 12-60
Document Revision History 12-60

Chapter 13. Netlist Optimizations and Physical Synthesis

Introduction 13-1
WYSIWYG Primitive Resynthesis 13-1
Performing Physical Synthesis Optimizations 13-3
Automatic Asynchronous Signal Pipelining 13-5
Physical Synthesis for Combinational Logic 13-6
Physical Synthesis for Registers—Register Duplication 13-7

Physical Synthesis for Registers—Register Retiming 13-8
Preserving Your Physical Synthesis Results 13-10
Physical Synthesis Options for Fitting 13-11
Applying Netlist Optimization Options 13-12
Scripting Support 13-12
Synthesis Netlist Optimizations 13-13
Physical Synthesis Optimizations 13-13
Incremental Compilation 13-14
Back-Annotating Assignments 13-14
Conclusion 13-14
Referenced Documents 13-15
Document Revision History 13-15

Chapter 14. Design Space Explorer

Introduction 14-1
DSE Concepts 14-1
Exploration Space and Exploration Point 14-1
Seed and Seed Sweeping 14-1
DSE Exploration 14-2
General Description 14-2
Timing Analyzer Support 14-3
DSE Flow 14-3
DSE Support for Altera Device Families 14-4
DSE Project Settings 14-4
Setting Up the DSE Work Environment 14-5
Specifying the Revision 14-5
Setting the Initial Seed 14-5
Project Uses Quartus II Integrated Synthesis 14-5
Restructuring LogicLock Regions 14-5
Exploration Settings 14-5
Using DSE to Search for the Best Performance 14-6
Effort Level 14-6
Seed Sweep 14-6
Extra Effort Space 14-7
Physical Synthesis Space 14-7
Physical Synthesis with Retiming Space 14-7
Selective Performance Optimization 14-7
DSE Flow Options 14-8
Create a Revision from the DSE Point 14-8
Stop Flow When Zero Failing Paths are Achieved 14-8
Continue Exploration Even If Base Compilation Fails 14-8
Run Quartus II PowerPlay Power Analyzer During Exploration 14-9
Archive All Compilations 14-9
Stop Flow After Time 14-9
Skip Base Analysis and Compilation If Possible 14-9
DSE Configuration File 14-9
Parallel DSE Information 14-10
Computer Load Sharing Using Parallel DSE 14-10
Parallel DSE Using LSF Resources 14-10
Parallel DSE Using a Quartus II Master Process 14-10
Concurrent Local Compilations 14-11
Referenced Documents 14-12
Document Revision History 14-12

Section IV. Engineering Change Management

Chapter 15. Engineering Change Management with the Chip Planner

Introduction . . . 15-1
Engineering Change Orders . . . 15-2
Performance . . . 15-2
Compilation Time . . . 15-3
Verification . . . 15-3
Documentation . . . 15-3
ECO Design Flow . . . 15-4
The Chip Planner Overview . . . 15-5
Opening the Chip Planner . . . 15-5
The Chip Planner Toolbar . . . 15-6
The Chip Planner Tasks and Layers . . . 15-7
The Chip Planner Floorplan Views . . . 15-8
First-Level View . . . 15-8
Second-Level View . . . 15-9
Third-Level View . . . 15-9
Bird's Eye View . . . 15-10
Performing ECOs with the Chip Planner (Floorplan View) . . . 15-12
Creating Atoms . . . 15-12
Creating ALM Atoms . . . 15-12
Creating Logic Element Atoms . . . 15-14
Deleting Atoms . . . 15-16
Moving Atoms . . . 15-16
Check and Save Netlist Changes . . . 15-16
Resource Property Editor . . . 15-16
Logic Element . . . 15-17
Logic Element Schematic View . . . 15-17
LE Properties . . . 15-18
Modes of Operation . . . 15-18
Sum and Carry Equations . . . 15-18
sload and sclear Signals . . . 15-18
Register Cascade Mode . . . 15-19
Cell Delay Table . . . 15-19
LE Connections . . . 15-19
Delete an LE . . . 15-19
Adaptive Logic Module . . . 15-19
ALM Schematic . . . 15-20
ALM Properties . . . 15-21
ALM Connections . . . 15-21
FPGA IOEs . . . 15-21
Arria GX, Stratix II, Stratix, and Stratix GX IOEs . . . 15-21
Stratix III IOE . . . 15-23
Cyclone II and Cyclone IOEs . . . 15-24
Cyclone III IOEs . . . 15-25
MAX II IOEs . . . 15-27
FPGA RAM Blocks . . . 15-27
FPGA DSP Blocks . . . 15-28
Change Manager . . . 15-29
Complex Changes in the Change Manager . . . 15-31
Managing SignalProbe Signals . . . 15-31
Exporting Changes . . . 15-31

Using Incremental Compilation in the ECO Flow 15-32
ECO Flow without Quartus II Incremental Compilation 15-33
Scripting Support 15-33
Common ECO Applications 15-33
Adjust the Drive Strength of an I/O Using the Chip Planner 15-33
Modify the PLL Properties Using the Chip Planner 15-34
PLL Properties 15-35
Adjusting the Duty Cycle 15-36
Adjusting the Phase Shift 15-36
Adjusting the Output Clock Frequency 15-37
Adjusting the Spread Spectrum 15-37
Modify the Connectivity between Resource Atoms 15-37
Post ECO Steps 15-38
Performing Static Timing Analysis 15-38
Conclusion 15-38
Referenced Documents 15-39
Document Revision History 15-40

Additional Information

About this Handbook Info-1
How to Contact Altera Info-1
Third-Party Software Product Information Info-1
Typographic Conventions Info-2

Chapter Revision Dates

The chapters in this book, *Quartus II Handbook Version 9.0 Volume 2: Design Implementation and Optimization*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1 Assignment Editor
Revised: *March 2009*
Part Number: *QII52001-9.0.0*

Chapter 2 Command-Line Scripting
Revised: *March 2009*
Part Number: *QII52002-9.0.0*

Chapter 3 Tcl Scripting
Revised: *March 2009*
Part Number: *QII52003-9.0.0*

Chapter 4 Managing Quartus II Projects
Revised: *March 2009*
Part Number: *QII52012-9.0.0*

Chapter 5 I/O Management
Revised: *March 2009*
Part Number: *QII52013-9.0.0*

Chapter 6 Simultaneous Switching Noise (SSN) Analysis and Optimizations
Revised: *March 2009*
Part Number: *QII52018-9.0.0*

Chapter 7 Signal Integrity Analysis with Third-Party Tools
Revised: *March 2009*
Part Number: *QII53020-9.0.0*

Chapter 8 Mentor Graphics PCB Design Tools Support
Revised: *March 2009*
Part Number: *QII52015-9.0.0*

Chapter 9 Cadence PCB Design Tools Support
Revised: *March 2009*
Part Number: *QII52014-9.0.0*

Chapter 10 Area and Timing Optimization
Revised: *March 2009*
Part Number: *QII52005-9.0.0*

Chapter 11 Power Optimization
Revised: *March 2009*
Part Number: *QII52016-9.0.0*

Chapter 12 Analyzing and Optimizing the Design Floorplan
Revised: *March 2009*
Part Number: *QII52006-9.0.0*

Chapter 13 Netlist Optimizations and Physical Synthesis
Revised: *March 2009*
Part Number: *QII52007-9.0.0*

Chapter 14 Design Space Explorer
Revised: *March 2009*
Part Number: *QII52008-9.0.0*

Chapter 15 Engineering Change Management with the Chip Planner
Revised: *March 2009*
Part Number: *QII52017-9.0.0*

Section I. Scripting and Constraint Entry

As a result of the increasing complexity of today's FPGA designs and the demand for higher performance, designers must make a large number of complex timing and logic constraints to meet their performance requirements. Once you have created a project and your design, you can use the Quartus® II software Assignment Editor and other GUI features to specify your initial design constraints, such as pin assignments, device options, logic options, and timing constraints.

This section describes how to enter constraints in the Quartus II software, how to take advantage of Quartus II modular executables, how to develop and run Tcl scripts to perform a wide range of functions, and how to manage the Quartus II project for your design.

This section includes the following chapters:

- Chapter 1, Assignment Editor
- Chapter 2, Command-Line Scripting
- Chapter 3, Tcl Scripting
- Chapter 4, Managing Quartus II Projects

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

1. Assignment Editor

QII52001-9.0.0

Introduction

The complexity of today's FPGA designs is compounded by the increasing density and associated pin counts of current FPGAs. It requires you to make a large number of pin assignments that include the pin locations and I/O standards to successfully implement a complex design in the latest generation of FPGAs.

To facilitate the process of entering these assignments, Altera has developed an intuitive, spreadsheet interface called the Assignment Editor (sometimes referred to as the editor in this chapter). The Assignment Editor is designed to make the process of creating, changing, and managing a large number of assignments as easy as possible.

This chapter discusses the following topics:

- "Overview of the Assignment Editor"
- "User Interface" on page 1–3
- "Navigating the Assignment Editor Spreadsheet" on page 1–5
- "Exporting and Importing Assignments" on page 1–7
- "Creating Timing Constraints Using the Assignment Editor" on page 1–9
- "Tcl Interface" on page 1–10
- "Probing to Source Design Files and Other Quartus II Windows" on page 1–10

Overview of the Assignment Editor

You can use the Assignment Editor throughout the design cycle, for making either timing or logic assignments. Altera recommends making assignments using the Assignment Editor to help reduce mistakes while making assignments throughout the design cycle. With the editor's dynamic syntax-checking capability, illegal assignments or incorrect settings can be avoided. You can also use the Assignment Editor to view, filter, and sort assignments based on node name or assignment type.

Even though the Assignment Editor allows you to make pin assignments, Altera recommends using the Pin Planner instead. You can use the Assignment Editor to view and verify the pin assignments that you have made.

The Assignment Editor is a resizable window. This scalability makes it easy to view or edit your assignments right next to your design files. To open the Assignment Editor, click the Assignment Editor icon in the toolbar, or on the Assignments menu, click **Assignment Editor**.

You can also launch the Assignment Editor by pressing **Ctrl**+**Shift**+**A** in the Quartus® II software.

The assignments made in the Assignment Editor are saved in the Quartus II Settings File (**.qsf**), which is located in the project directory. A separate **.qsf** file exists for each individual revision. Every new assignment is placed on a new line at the end of the **.qsf** file. Refer to the Quartus II Help for the syntax of the **.qsf** file. Refer to the "Viewing and Saving Assignments in the Assignment Editor" section for more details about how Assignment Editor handles your assignments.

Every time an assignment is created or updated, the Quartus II software displays the equivalent Tcl command in the **System** tab of the Messages window. You can use the displayed messages as references when making assignments using Tcl commands.

For information about exporting your assignments to a Tcl file, refer to "Tcl Interface" on page 1–10.

Dynamic Syntax Checking

As you enter assignments, the Assignment Editor checks for basic legality and syntax. This checking is not as thorough as the checks performed during compilation, but it rejects incorrect settings.

If there are incomplete assignments when you click **Save** on the File menu, a prompt gives you the choice to either save the file and lose incomplete assignments or cancel the save operation.

The legality status of the assignments is color coded. The color of the text in each row indicates if the assignment is incomplete, disabled, non-editable, or contains errors or warnings (Table 1–1). To customize the colors used in the Assignment Editor, on the Tools menu, click **Options**.

Table 1–1. Description of the Color Codes in the Spreadsheet

Text Color	Description
Green	A new assignment can be created
Dark Yellow	The assignment contains warnings, such as unknown node name
Dark Red	The assignment is incomplete
Red	The assignment has an error, such as an illegal value
Light Gray	The assignment is disabled or turned off
Gray	The assignment is non-editable or read-only

Viewing and Saving Assignments in the Assignment Editor

Although the Assignment Editor is the most common method of entering and modifying assignments, there are other methods you can use to make and edit assignments. For this reason, you can refresh the Assignment Editor after you add, remove, or change an assignment outside the Assignment Editor.

By default, all assignments made in the Quartus II software are first stored in memory, then to the **.qsf** file on the disk after you start a processing task, or if you save or close your project. Saving assignments to memory avoids reading and writing to your disk drive and improves the performance of the software.

After making assignments in the Assignment Editor, on the File menu, click **Save** to save your assignments and update the **.qsf** file.

© March 2009 Altera Corporation

Starting with the Quartus II software version 5.1, you can force all assignments to be written to a disk drive. On the Tools menu, click **Options**. In the **Options** dialog box, open the **Processing** page and turn off **Update assignments to disk during design processing only**.

For more information about how the Quartus II software writes to the **.qsf** file, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

You can refresh the Assignment Editor window by clicking **Refresh** on the View menu. If you make an assignment using the Tcl console, Pin Planner, or directly modifying the **.qsf** file outside the Assignment Editor, you must click **Refresh** to update the Assignment Editor spreadsheet.

If the **.qsf** file is edited while the project is open, on the File menu, click **Save Project** to ensure that you are editing the latest **.qsf** file.

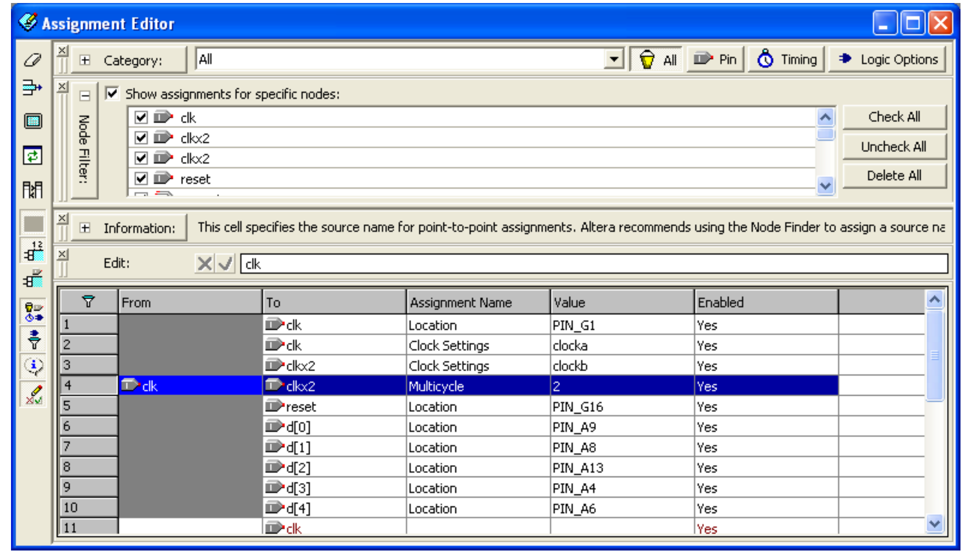
Each time the Assignment Editor is refreshed, the following message is displayed in the **System** tab of the Messages window:

```
Info: Assignments reloaded -- assignments updated outside Assignment
Editor
```

User Interface

The Assignment Editor window consists of four bars and a spreadsheet (Figure 1–1). The spreadsheet is where you view and edit the assignments. You can use the bars to filter, edit, or get detailed information about the assignments.

Figure 1–1. The Assignment Editor Window



You can hide all four bars in the View menu if desired, or you can collapse all the bars for a better view of the spreadsheet. Table 1–2 provides a brief description of each bar.

Table 1–2. Assignment Editor Bar Descriptions

Bar Name	Description
Category	Lists the types of available assignments
Node Filter	Lists a selection of design nodes to be viewed or assigned
Information	Displays a description of the selected assignment
Edit	Allows you to edit the text in the selected cell(s)

Category Bar

The **Category** bar lists all assignment categories available for the selected device. You can use the **Category** bar to select a particular type of assignment and filter out all other assignments, making the spreadsheet show only the applicable assignments. For example, to view all t_{SU} (setup time) assignments in your project, select **tsu** in the **Category** list. If you select **All** in the **Category** list, the Assignment Editor displays all assignments.

When you collapse the **Category** bar, three buttons are displayed that allow you to select from various preset categories (Figure 1–2). For example, clicking the **Timing** button changes the spreadsheet to show only the timing-related assignments of your project.

Figure 1–2. Collapsed Category Bar

Node Filter Bar

The **Node Filter** bar is used when you want the spreadsheet to show only assignments for specific nodes based on the list of selected node filters. The bar provides flexibility in how you view and make your settings.

For example, when **Show assignments for specific nodes** is turned on, the spreadsheet shows only assignments for nodes that match the selected node filter. You can create a new node filter by using the Node Finder to select a node name or by typing the node name. The node name filter can be a node name or assignment group, and can include wildcard symbols (* and ?). The wildcards symbols are used to filter for a selection of nodes with only one entry in the Node Filter. For more information about wildcard symbols and assignment groups, refer to "Navigating the Assignment Editor Spreadsheet" on page 1–5.

Information Bar

The **Information** bar provides a brief description of the currently selected cell and what information you should enter into the cell. For example, the **Information** bar describes whether a cell should contain a node name or a number value.

Edit Bar

The **Edit** bar is an efficient way to enter a value into one or more spreadsheet cells. To change the contents of multiple cells at the same time, select the cells in the spreadsheet, then type or choose the new value in the **Edit** field in the **Edit** bar.

© March 2009 Altera Corporation

Assignment Spreadsheet

The spreadsheet displays the assignments of your project. You can sort columns, use pull-down list boxes to view available options, and copy and paste multiple cells into the Assignment Editor. Refer to "Entering Values into the Spreadsheet" for details about ways to make assignments in the spreadsheet.

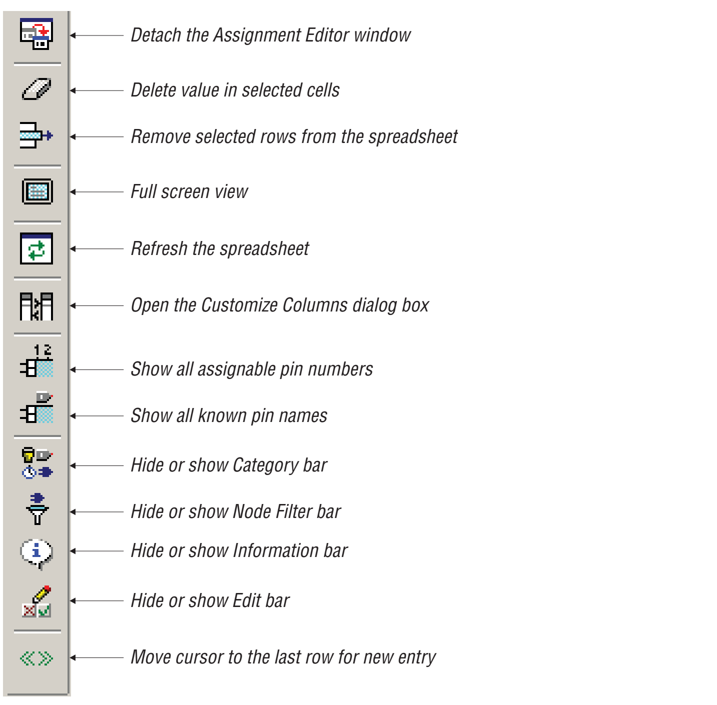
When you enter an assignment, the font color of the row changes to indicate the status of the assignment. For more information, refer to "Dynamic Syntax Checking" on page 1–2.

The spreadsheet also supports customizable columns that allow you to show, hide, and arrange columns. For more information, refer to "Customizing the Spreadsheet Columns" on page 1–7.

Toolbar

The Assignment Editor's toolbar contains shortcut buttons for easy access to the editor's features. Figure 1–3 describes the buttons in the toolbar.

Figure 1–3. Description of Icons in the Toolbar



Navigating the Assignment Editor Spreadsheet

This section describes methods for navigating through the spreadsheet.

Entering Values into the Spreadsheet

There are many ways to select or enter nodes into the spreadsheet, including the Node Finder, the Node Filter bar, the Edit bar, or by directly typing the node name into a cell in the spreadsheet.

To open the Node Finder, click **Node Finder** in the Edit menu, or right-click any cell in the spreadsheet or the Node Filter bar and click **Node Finder**. Alternatively, you can click **Node Finder** in the pull-down menu of the **To** and **From** columns in the spreadsheet.

A node type icon is shown beside each node name and filter to identify its type. Table 1–3 lists the different ions and describes its type.

Table 1–3. Node Type Icons

Node Type Icon	Description
a[0] a	Input pin
result[0] result	Output pin
taps_bidir taps_bidir	Bidirectional pin
taps:inst|xn[0] taps:inst|xn	Register
taps:inst|xn[1] taps:inst|xn	Combinational logic
Asg_1	Assignment group
d*	Node name or filter that contains wildcard symbols
taps:inst	Instance
ae	Missing information—probably caused by incorrect node name, or Analysis and Synthesis has not been performed

Wildcards

To simplify the tasks of making many node assignments, the Quartus II software accepts the `*` and `?` wildcard characters. Use these wildcard characters to reduce the number of individual assignments you need to make for your design.

The `*` wildcard character matches any string. For example, given an assignment made to a node specified as `reg*`, the Assignment Editor applies the assignment to all design nodes that match the prefix `reg` with none, one, or several characters following the prefix, such as `reg`, `reg1`, `reg[2]`, `regbank`, and `reg12bank`.

The `?` wildcard character matches any single character. For example, given an assignment made to a node specified as `reg?`, the Assignment Editor applies the assignment to all design nodes that match the prefix `reg` and any single character following, such as `reg1`, `rega`, and `reg4`.

All assignments that support wildcards are shown in the pull-down list under the **Assignment Name** column of the Assignment Editor with "(Accepts wildcards/groups)" displayed beside it.

Assignment Groups

An assignment group, also known as a time group, is a collection of design nodes grouped together and represented as a single unit for the purpose of making assignments to the collection. Using assignment groups with the Assignment Editor provides the flexibility required for making complex assignments to a large number of nodes. You can also exclude specific nodes, wildcards, and assignments from an assignment group.

To create an assignment group, on the Assignments menu, click **Assignment (Time) Groups**. The **Assignment Groups** dialog box appears. You can add or delete members of each assignment group using wildcards in the Node Finder.

For more information about using Assignment Groups for timing analysis, refer to the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Customizing the Spreadsheet Columns

To provide more control over the display of information in the spreadsheet, the Assignment Editor supports customizable columns.

You can move columns, sort them in ascending or descending order, show or hide individual columns, and align the content of the column left, center, or right for improved readability.

When the Quartus II software starts for the first time, you see a pre-selected set of columns. For example, when the Quartus II software is first started, the Comment column is hidden. To show or hide any of the available columns, on the View menu, click **Customize Columns**. When you restart the Quartus II software, your column settings are maintained.

You can use the **Comments** column to document the purpose of an assignment, or to explain why you applied a timing or logic constraint. You can use the **Enabled** column to disable any assignment without deleting it. This feature is useful when performing multiple compilations with different timing constraints or logic optimizations.

Exporting and Importing Assignments

Designs that use the LogicLock™ hierarchical design methodology use the **Import Assignments** command to import assignments into the current project. You can also use the **Export Assignments** command to save all the assignments in your project to a file to be used for archiving or to transfer assignments from one project to another.

On the Assignments menu, click **Export Assignments** or **Import Assignments** to do the following:

- Export your Quartus II assignments to a **.qsf** file.

- Import assignments from a Quartus II Entity Settings File (**.esf**), a MAX+PLUS® II Assignment and Configuration File (**.acf**), a Synopsys Design Constraint (**.sdc**) file, a text file (**.txt**), or a Comma Separated Value (**.csv**) file.

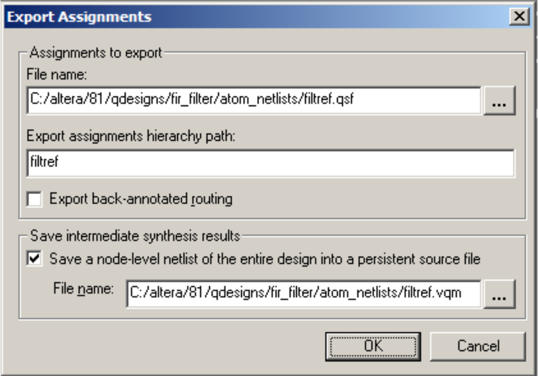
In addition to the **Export Assignments** and **Import Assignments** dialog boxes, the **Export** command on the File menu allows you to export your assignments to a **.tcl** file or a **.csv** file.

You can use these file formats for many different aspects of your project. For example, you can use a **.csv** file for documentation purposes or to transfer pin-related information to board layout tools. The **.tcl** file makes it easy to apply assignments in a scripted design flow. The LogicLock design flow uses the **.qsf** file to transfer your LogicLock region settings.

Exporting Assignments

You can use the **Export Assignments** dialog box to export your Quartus II software assignments into a **.qsf** file, generate a node-level netlist file, and export back-annotated routing information as a Routing Constraints File (**.rcf**). See Figure 1–4.

Figure 1–4. Export Assignments Dialog Box



On the Assignments menu, click **Export Assignments** to open the **Export Assignments** dialog box. The LogicLock design flow also uses this dialog box to export LogicLock regions.

For more information about using the **Export Assignments** dialog box to export LogicLock regions, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

On the File menu, click **Export** to export all assignments to a **.tcl** file or export a set of assignments to a **.csv** file. When you export assignments to a **.tcl** file, only user-created assignments are written to the Tcl script file; default assignments are not exported.

When assignments are exported to a **.csv** file, only the assignments displayed in the current view of the Assignment Editor are exported.

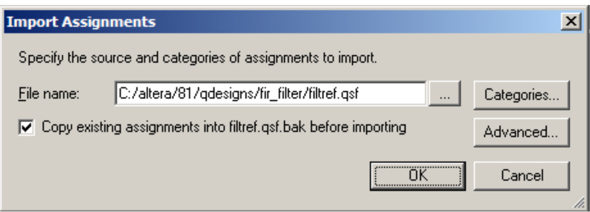
Importing Assignments

The **Import Assignments** dialog box allows you to import Quartus II assignments from a **.qsf** file, an **.esf** file, an **.acf** file, or a **.csv** file (Figure 1–5).

To import assignments from any of the supported assignment files, perform the following steps:

1. On the Assignments menu, click **Import Assignments**. The **Import Assignments** dialog box appears (Figure 1–5).

Figure 1–5. Import Assignments Dialog Box



2. In the **File name** text-entry box, type the file name or browse to the assignment file. The **Select File** dialog box appears.
3. In the **Select File** dialog box, select the file and click **Open**.
4. Click **OK**.

When you import a **.csv** file, the first uncommented row of the file must be in the exact format as it was when exported.

You can create a backup copy of your assignments before importing new assignments by turning on the **Copy existing assignments into <*revision name*>.qsf.bak before importing** option.

When importing assignments from a file, you can choose which assignment categories to import by following these steps:

1. Click **Categories** in the **Import Assignments** dialog box.
2. Turn on the categories you want to import from the **Assignment categories** list.

To select specific types of assignments to import, click **Advanced** in the **Import Assignments** dialog box. The **Advanced Import Settings** dialog box appears. You can choose to import instance, entity, or global assignments and select various assignment types to import.

For more information about these options, refer to the Quartus II Help.

Creating Timing Constraints Using the Assignment Editor

Accurate timing constraints guide the place-and-route engine in the Quartus II software to help optimize your design into the FPGA. After completing a place-and-route, perform a static timing analysis using the Quartus II Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer to analyze slack and critical paths in your design.

If you are using the Quartus II Classic Timing Analyzer, create timing constraints using the Assignment Editor. In the Assignment Editor, select **Timing** in the **Category** list to show all timing-related settings and make the desired timing assignments in the editor's spreadsheet.

For more information about the Quartus II Classic Timing Analyzer, refer to the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

If you are using the Quartus II TimeQuest Timing Analyzer, the TimeQuest Timing Analyzer uses timing assignments from a Synopsys Design Constraint (**.sdc**) file. Therefore, you must convert assignments from the **.qsf** format to the **.sdc** format before you can proceed with the timing analysis.

For information about converting the timing assignments in your **.qsf** file to an **.sdc** file, refer to the *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Tcl Interface

Whether you use the Assignment Editor or another feature to create your design assignments, you can export them to a Tcl (**.tcl)** file. You can then use the **.tcl** file to reapply the settings or to archive your assignments. On the File menu, click **Export** to export your saved assignments to a **.tcl** file.

You can also generate a **.tcl** file that sets up your project and applies all the assignments to it. On the Project menu, click **Generate Tcl File for Project** to generate the file.

In addition, as you use the Assignment Editor to enter assignments, the equivalent Tcl commands are shown in the **System** tab of the Messages window. You can reference these Tcl commands to create a customized **.tcl** file. To copy a Tcl command from the **System** tab of the Messages window, right-click the message and click **Copy**.

For more information about Tcl scripting with the Quartus II software, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook.*

Probing to Source Design Files and Other Quartus II Windows

The Assignment Editor lets you cross-probe from the viewer to the source design file and to other windows within the Quartus II software. You can select a cell in the Assignment Editor spreadsheet and locate the corresponding item in another applicable Quartus II software window. To locate an item from the editor in another window, right-click the items of interest in the spreadsheet, point to **Locate**, and click the appropriate command. The following commands are available:

- Assignment Editor
- Pin Planner
- Timing Closure Floorplan
- Chip Planner (Floorplan & Chip Editor)
- Resource Property Editor

- Technology Map Viewer
- RTL Viewer
- Design File

Probing to the Assignment Editor from Other Quartus II Windows

You can cross-probe to the Assignment Editor from other windows within the Quartus II software. You can select one or more nodes or nets in another window and locate them in the Assignment Editor spreadsheet. This is useful when you want to see all assignments related to specific nodes.

You can locate nodes in the Assignment Editor from all windows within the Quartus II software. To locate assignments related to an element in the editor from other Quartus II windows, select the node or nodes in the appropriate window. For example, select an entity in the **Entity** list on the **Hierarchy** tab in the Project Navigator, or select nodes in the Timing Closure Floorplan. Next, right-click the selected object, point to **Locate**, and click **Locate in Assignment Editor**. The Assignment Editor opens, or it is brought to the foreground if it is already open.

Conclusion

As FPGAs continue to increase in density and pin count, it is essential to be able to quickly create and view design assignments. The Assignment Editor provides an intuitive and effective way of making assignments. With the spreadsheet interface and the **Category**, **Node Filter**, **Information**, and **Edit** bars, the Assignment Editor provides an efficient assignment entry solution for FPGA designers.

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 1–4 shows the revision history for this chapter.

Table 1–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0	■ Revised and reorganized the entire chapter. ■ Added section "Probing to Source Design Files and Other Quartus II Windows" on page 1–10. ■ Added description of node type icons (Table 1–3). ■ Added explanation of wildcard characters.	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1	Changed to 8½" × 11" page size. No change to content.	Updated for the Quartus II software version 8.1.
May 2008 v8.0	Updated Quartus II software 8.0 revision and date.	Updated references.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

© March 2009 Altera Corporation

2. Command-Line Scripting

QII52002-9.0.0

Introduction

FPGA design software that easily integrates into your design flow saves time and improves productivity. The Altera® Quartus® II software provides you with a command-line executable for each step of the FPGA design flow to make the design process customizable and flexible.

The benefits provided by command-line executables include:

- Command-line control over each step of the design flow
- Easy integration with scripted design flows including makefiles
- Reduced memory requirements
- Improved performance

The command-line executables are also completely compatible with the Quartus II GUI, allowing you to use the exact combination of tools that you prefer.

This chapter describes how to take advantage of Quartus II command-line executables, and provides several examples of scripts that automate different segments of the FPGA design flow, including the following topics:

- "The Benefits of Command-Line Executables"
- "Introductory Example" on page 2–2
- "Command-Line Executables" on page 2–3
- "Design Flow" on page 2–9
- "The MegaWizard Plug-In Manager" on page 2–13
- "Command-Line Scripting Examples" on page 2–19

The Benefits of Command-Line Executables

The Quartus II command-line executables provide command-line control over each step of the design flow. Each executable includes options to control commonly used software settings. Each executable also provides detailed, built-in help describing its function, available options, and settings.

Command-line executables allow for easy integration with scripted design flows. You can easily create scripts in any language with a series of commands. These scripts can be batch-processed, allowing for integration with distributed computing in server farms. You can also integrate the Quartus II command-line executables in makefile-based design flows. All of these features enhance the ease of integration between the Quartus II software and other EDA synthesis, simulation, and verification software.

Command-line executables add integration and scripting flexibility without sacrificing the ease-of-use of the Quartus II GUI. You can use the Quartus II GUI and command-line executables at different stages in the design flow. For example, you might use the Quartus II GUI to edit the floorplan for the design, use the command-line executables to perform place-and-route, and return to the Quartus II GUI to perform debugging with the Chip Editor.

Command-line executables reduce the amount of memory required during each step in the design flow. Because each executable targets only one step in the design flow, it is relatively compact, both in file size and the amount of memory used when running. This memory reduction improves performance, and is particularly beneficial in design environments where computer networks or workstations are heavily used with reduced memory.

Introductory Example

The following introduction to design flow with command-line executables shows how to create a project, fit the design, perform timing analysis, and generate programming files.

The tutorial design included with the Quartus II software is used to demonstrate this functionality. If installed, the tutorial design is found in the <*Quartus II directory*>**/qdesigns/fir_filter** directory.

Before making changes, copy the tutorial directory and type the four commands shown in Example 2–1 at a command prompt in the new project directory:

The <*Quartus II directory*>**/quartus/bin** directory must be in your `PATH` environment variable.

Example 2–1. Introductory Example

```
quartus_map filtref --source=filtref.bdf --family=CYCLONE ↵
quartus_fit filtref --part=EP1C12F256C6 --fmax=80MHz --tsu=8ns ↵
quartus_asm filtref ↵
quartus_tan filtref ↵
```

The `quartus_map filtref --source=filtref.bdf --family=CYCLONE` command creates a new Quartus II project called **filtref** with the **filtref.bdf** file as the top-level file. It targets the Cyclone® device family and performs logic synthesis and technology mapping on the design files.

The `quartus_fit filtref --part=EP1C12Q240C6 --fmax=80MHz --tsu=8ns` command performs fitting on the **filtref** project. This command specifies an EP1C12Q240C6 device and the Fitter attempts to meet a global f_{MAX} requirement of 80 MHz and a global t_{SU} requirement of 8 ns.

The `quartus_asm filtref` command creates programming files for the **filtref** project.

The `quartus_tan filtref` command performs timing analysis on the **filtref** project to determine whether the design meets the timing requirements that were specified to the `quartus_fit` executable.

You can put the four commands from Example 2–1 into a batch file or script file, and run them. For example, you can create a simple UNIX shell script called **compile.sh**, which includes the code shown in Example 2–2.

Example 2–2. UNIX Shell Script: compile.sh

```
#!/bin/sh
PROJECT=filtref
TOP_LEVEL_FILE=filtref.bdf
FAMILY=Cyclone
PART=EP1C12F256C6
FMAX=80MHz
--TSU=8ns
quartus_map $PROJECT --source=$TOP_LEVEL_FILE --family=$FAMILY
quartus_fit $PROJECT --part=$PART --fmax=$FMAX --tsu=$TSU
quartus_asm $PROJECT
quartus_tan $PROJECT
```

Edit the script as necessary and compile your project.

Command-Line Executables

Table 2–1 lists the command-line executables and their respective descriptions.

Table 2–1. Quartus II Command-Line Executables and Descriptions (Part 1 of 3)

Executable	Description
Analysis and Synthesis `quartus_map`	Quartus II Analysis and Synthesis builds a single project database that integrates all the design files in a design entity or project hierarchy, performs logic synthesis to minimize the logic of the design, and performs technology mapping to implement the design logic using device resources such as logic elements.
Fitter `quartus_fit`	The Quartus II Fitter performs place-and-route by fitting the logic of a design into a device. The Fitter selects appropriate interconnection paths, pin assignments, and logic cell assignments. Quartus II Analysis and Synthesis must be run successfully before running the Fitter.
Assembler `quartus_asm`	The Quartus II Assembler generates a device programming image, in the form of one or more of the following from a successful fit (that is, place-and-route). ■ Programmer Object Files (**.pof**) ■ SRAM Object Files (**.sof**) ■ Hexadecimal (Intel-Format) Output Files (**.hexout**) ■ Tabular Text Files (**.ttf**) ■ Raw Binary Files (**.rbf**) The **.pof** and **.sof** files are then processed by the Quartus II Programmer and downloaded to the device with the MasterBlaster™ or the ByteBlaster™ II download cable, or the Altera Programming Unit (APU). The Hexadecimal (Intel-Format) Output Files, Tabular Text Files, and Raw Binary Files can be used by other programming hardware manufacturers that provide support for Altera devices. The Quartus II Fitter must be run successfully before running the Assembler.

Table 2–1. Quartus II Command-Line Executables and Descriptions (Part 2 of 3)

Executable	Description
Classic Timing Analyzer `quartus_tan`	The Quartus II Classic Timing Analyzer computes delays for the given design and device, and annotates them on the netlist. Then, the Classic Timing Analyzer performs timing analysis, allowing you to analyze the performance of all logic in your design. The `quartus_tan` executable includes Tcl support. Quartus II Analysis and Synthesis or the Fitter must be run successfully before running the Classic Timing Analyzer.
TimeQuest Timing Analyzer `quartus_sta`	The Quartus II TimeQuest Timing Analyzer computes delays for the given design and device, and annotates them on the netlist. Next, the TimeQuest Timing Analyzer performs timing analysis, allowing you to analyze the performance of all logic in your design. The `quartus_sta` executable includes Tcl support and SDC support. Quartus II Analysis and Synthesis or the Fitter must be run successfully before running the TimeQuest Timing Analyzer.
Design Assistant `quartus_drc`	The Quartus II Design Assistant checks the reliability of a design based on a set of design rules. The Design Assistant is especially useful for checking the reliability of a design before converting the design for HardCopy® devices. The Design Assistant supports designs that target any Altera device supported by the Quartus II software, except MAX® 3000 and MAX 7000 devices. Quartus II Analysis and Synthesis or the Fitter must be run successfully before running the Design Assistant.
Compiler Database Interface `quartus_cdb`	The Quartus II Compiler Database Interface generates incremental netlists for use with LogicLock™ back-annotation, or back-annotates device and resource assignments to preserve the fit for future compilations. The `quartus_cdb` executable includes Tcl support. Analysis and Synthesis must be run successfully before running the Compiler Database Interface.
EDA Netlist Writer `quartus_eda`	The Quartus II EDA Netlist Writer generates netlist and other output files for use with other EDA tools. Analysis and Synthesis, the Fitter, or Timing Analyzer must be run successfully before running the EDA Netlist Writer, depending on the arguments used.
Simulator `quartus_sim`	The Quartus II Simulator tests and debugs the logical operation and internal timing of the design entities in a project. The Simulator can perform two types of simulation: functional simulation and timing simulation. The `quartus_sim` executable includes Tcl support. Quartus II Analysis and Synthesis must be run successfully before running a functional simulation. The Timing Analyzer must be run successfully before running a timing simulation.
Power Analyzer `quartus_pow`	The Quartus II PowerPlay Power Analyzer estimates the thermal dynamic power and the thermal static power consumed by the design. For newer families such as Stratix® II and MAX II, the power drawn from each power supply is also estimated. Quartus II Analysis and Synthesis or the Fitter must be run successfully before running the PowerPlay Power Analyzer.

Table 2–1. Quartus II Command-Line Executables and Descriptions (Part 3 of 3)

Executable	Description
Programmer `quartus_pgm`	The Quartus II Programmer programs Altera devices. The Programmer uses one of the supported file formats: ■ **.pof** file ■ **.sof** file ■ Jam File (**.jam**) ■ Jam Byte-Code File (**.jbc**) Make sure you specify a valid programming mode, programming cable, and operation for a specified device.
Convert Programming File `quartus_cpf`	The Quartus II Convert Programming File module converts one programing file format to a different possible format. Make sure you specify valid options and an input programming file to generate the new requested programming file format.
Quartus Shell `quartus_sh`	The Quartus II Shell acts as a simple Quartus II Tcl interpreter. The Shell has a smaller memory footprint than the other command-line executables that support Tcl. The Shell may be started as an interactive Tcl interpreter (shell), used to run a Tcl script, or used as a quick Tcl command evaluator, evaluating the remaining command-line arguments as one or more Tcl commands.
TimeQuest Timing Analyzer GUI `quartus_staw`	This executable opens the Quartus II TimeQuest Timing Analyzer GUI. This is helpful because you don't have to open the entire Quartus II GUI for certain operations.
Programmer GUI `quartus_pgmw`	This executable opens up the programmer—a GUI to the `quartus_pgm` executable. This is helpful because users don't have to open the entire Quartus II GUI for certain operations

Command-Line Scripting Help

Help on command-line executables is available through different methods. You can access help built in to the executables with command-line options. You can use the Quartus II Command-Line and Tcl API Help browser for an easy graphical view of the help information. Additionally, you can refer to the *Quartus II Scripting Reference Manual* on the Quartus II literature page of Altera's website, which has the same information in PDF format.

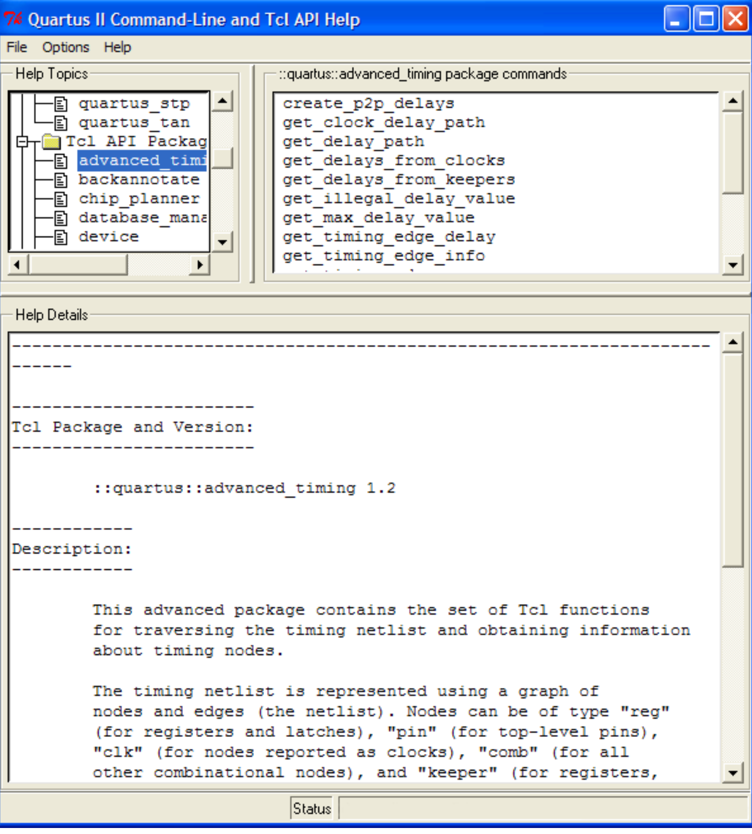
To use the Quartus II Command-Line and Tcl API Help browser, type the following:

```
quartus_sh --qhelp ↵
```

This command starts the Quartus II Command-Line and Tcl API Help browser, a viewer for information about the Quartus II Command-Line executables and Tcl API (Figure 2–1).

Use the `-h` option with any of the Quartus II Command-Line executables to get a description and list of supported options. Use the `--help=`*<option name>* option for detailed information about each option.

Figure 2–1. Quartus II Command-Line and Tcl API Help Browser



Command-Line Option Details

Command-line options are provided for many common global project settings and performing common tasks. You can use either of two methods to make assignments to an individual entity. If the project exists, open the project in the Quartus II GUI, change the assignment, and close the project. The changed assignment is updated in the Quartus II Settings File. Any command-line executables that are run after this update use the updated assignment. Refer to “Option Precedence” on page 2–7 for more information. You can also make assignments using the Quartus II Tcl scripting API. If you want to completely script the creation of a Quartus II project, choose this method.

Refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Scripting information for all Quartus II project settings and assignments is located in the *QSF Reference Manual*.

© March 2009 Altera Corporation

Option Precedence

If you use command-line executables, you should be aware of the precedence of various project assignments and how to control the precedence. Assignments for a particular project exist in the Quartus II Settings File for the project. Assignments for a project can also be made with command-line options. Project assignments are reflected in compiler database files that hold intermediate compilation results and reflect assignments made in the previous project compilation.

All command-line options override any conflicting assignments found in the Quartus II Settings File or the compiler database files. There are two command-line options to specify whether Quartus II Settings File or compiler database files take precedence for any assignments not specified as command-line options.

Any assignment not specified as a command-line option or found in the Quartus II Settings File or compiler database file is set to its default value.

The file precedence command-line options are `--read_settings_files` and `--write_settings_files`.

By default, the `--read_settings_files` and `--write_settings_files` options are turned on. Turning on the `--read_settings_files` option causes a command-line executable to read assignments from the Quartus II Settings File instead of from the compiler database files. Turning on the `--write_settings_files` option causes a command-line executable to update the Quartus II Settings File to reflect any specified options, as happens when you close a project in the Quartus II GUI.

If you use command-line executables, you should be aware of the precedence of various project assignments and how to control the precedence. Assignments for a particular project can exist in three places:

- The Quartus II Settings File for the project
- The result of the last compilation, in the **\db** directory, reflects the assignments that existed when the project was compiled.
- Command-line options

Table 2–2 lists the precedence for reading assignments depending on the value of the `--read_settings_files` option.

Table 2–2. Precedence for Reading Assignments

Option Specified	Precedence for Reading Assignments
`--read_settings_files = on` (Default)	1. Command-line options 2. Quartus II Settings File 3. Project database (db directory, if it exists) 4. Quartus II software defaults
`--read_settings_files = off`	1. Command-line options 2. Project database (db directory, if it exists) 3. Quartus II software defaults

Table 2–3 lists the locations to which assignments are written, depending on the value of the `--write_settings_files` command-line option.

Table 2–3. Location for Writing Assignments

Option Specified	Location for Writing Assignments
`--write_settings_files = on` (Default)	Quartus II Settings File and compiler database
`--write_settings_files = off`	Compiler database

Example 2–3 assumes that a project named **fir_filter** exists, and that the analysis and synthesis step has been performed (using the `quartus_map` executable).

Example 2–3. Write Settings Files

```
quartus_fit fir_filter --fmax=80MHz ↵
quartus_tan fir_filter ↵
quartus_tan fir_filter --fmax=100MHz --tao=timing_result-100.tao
   --write_settings_files=off ↵
```

The first command, `quartus_fit fir_filter --fmax=80MHz`, runs the `quartus_fit` executable and specifies a global f_{MAX} requirement of 80 MHz.

The second command, `quartus_tan fir_filter`, runs Quartus II timing analysis for the results of the previous fit.

The third command reruns Quartus II timing analysis with a global f_{MAX} requirement of 100 MHz and saves the result in a file called **timing_result-100.tao**. By specifying the `--write_settings_files=off` option, the command-line executable does not update the Quartus II Settings File to reflect the changed f_{MAX} requirement. The compiler database files reflect the changed f_{MAX} requirement. If the `--write_settings_files=off` option is not specified, the command-line executable updates the Quartus II Settings File to reflect the 100-MHz global f_{MAX} requirement.

Use the options `--read_settings_files=off` and `--write_settings_files=off` (where appropriate) to optimize the way that the Quartus II software reads and updates settings files. Example 2–4 shows how to avoid unnecessary reading and writing.

Example 2–4. Avoiding Unnecessary Reading and Writing

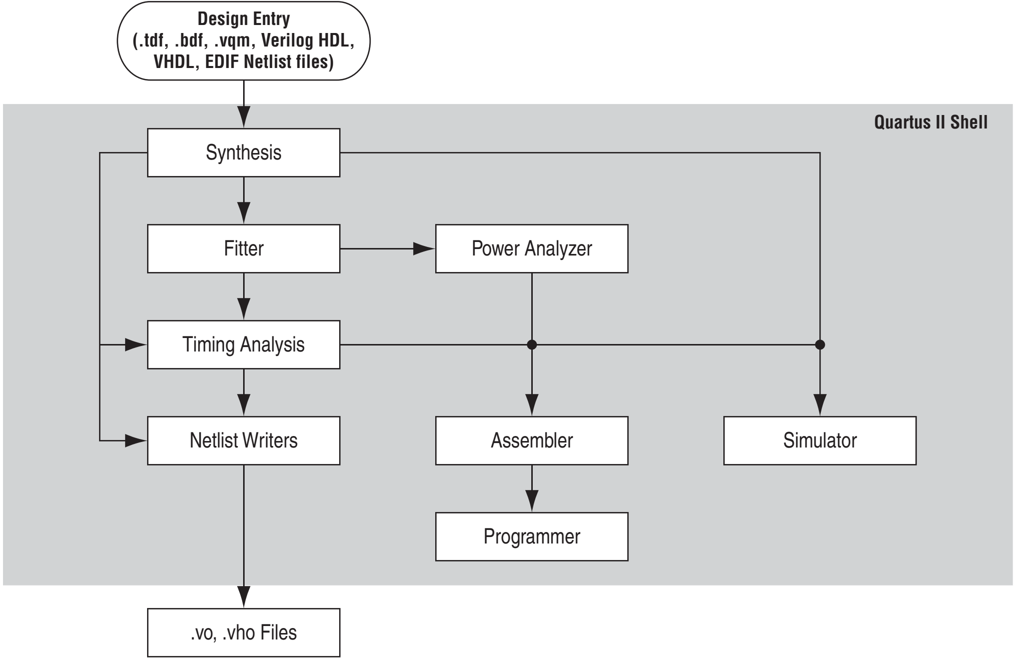
```
quartus_map filtref --source=filtref --part=EP1C12F256C6 ↵
quartus_fit filtref --fmax=100MHz --read_settings_files=off ↵
quartus_tan filtref --read_settings_files=off --write_settings_files=off ↵
quartus_asm filtref --read_settings_files=off --write_settings_files=off ↵
```

The `quartus_tan` and `quartus_asm` executables do not read or write settings files because they do not change any settings in the project.

Design Flow

Figure 2–2 shows a typical design flow.

Figure 2–2. Typical Design Flow



Compilation with quartus_sh --flow

Use the `quartus_sh` executable with the `--flow` option to perform a complete compilation flow with a single command. (For information about specialized flows, type `quartus_sh --help=flow` ↵ at a command prompt.) The `--flow` option supports the smart recompile feature and efficiently sets command-line arguments for each executable in the flow.

If you used the `quartus_cmd` executable to perform command-line compilations in earlier versions of the Quartus II software, you should use the `quartus_sh --flow` command beginning with the Quartus II software version 3.0.

The following example runs compilation, timing analysis, and programming file generation with a single command:

```
quartus_sh --flow compile filtref ↵
```

Text-Based Report Files

Each command-line executable creates a text report file when it is run. These files report success or failure, and contain information about the processing performed by the executable.

Report file names contain the revision name and the short-form name of the executable that generated the report file: <*revision*>.<*executable*>**.rpt**. For example, using the `quartus_fit` executable to place and route a project with the revision name **design_top** generates a report file named **design_top.fit.rpt**. Similarly, using the `quartus_tan` executable to perform timing analysis on a project with the revision name **fir_filter** generates a report file named **fir_filter.tan.rpt**.

As an alternative to parsing text-based report files, you can use the Tcl package called **::quartus::report**. For more information about this package, refer to "Command-Line Scripting Help" on page 2–5.

You can use Quartus II command-line executables in scripts that control a design flow that uses other software in addition to the Quartus II software. For example, if your design flow uses other synthesis or simulation software, and you can run the other software at a system command prompt, you can include it in a single script. The Quartus II command-line executables include options for common global project settings and operations, but you must use a Tcl script or the Quartus II GUI to set up a new project and apply individual constraints, such as pin location assignments and timing requirements. Command-line executables are very useful for working with existing projects, for making common global settings, and for performing common operations. For more flexibility in a flow, use a Tcl script, which makes it easier to pass data between different stages of the design flow and have more control during the flow.

For more information about Tcl scripts, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*, or the *Quartus II Scripting Reference Manual*.

For example, your script could run other synthesis software, then place-and-route the design in the Quartus II software, then generate output netlists for other simulation software. Example 2–5 shows how to do this with a UNIX shell script for a design that targets a Cyclone II device.

Example 2–5. Script for End-to-End Flow (Part 1 of 2)

```
#!/bin/sh
# Run synthesis first.
# This example assumes you use Synplify software
synplify -batch synthesize.tcl

# If your Quartus II project exists already, you can just
# recompile the design.
# You can also use the script described in a later example to
# create a new project from scratch
quartus_sh --flow compile myproject

# Use the quartus_tan executable to do best and worst case
# timing analysis
quartus_tan myproject --tao=worst_case
quartus_tan myproject --fast_model --tao=best_case

# Use the quartus_eda executable to write out a gate-level
# Verilog simulation netlist for ModelSim
quartus_eda my_project --simulation --tool=modelsim --format=verilog
```

Example 2–5. Script for End-to-End Flow (Part 2 of 2)

```
# Perform the simulation with the ModelSim software
vlib cycloneii_ver
vlog -work cycloneii_ver /opt/quartusii/eda/sim_lib/cycloneii_atoms.v
vlib work
vlog -work work my_project.vo
vsim -L cycloneii_ver -t 1ps work.my_project
```

Makefile Implementation

You can use the Quartus II command-line executables in conjunction with the `make` utility to automatically update files when other files they depend on change. The file dependencies and commands used to update files are specified in a text file called a makefile.

To facilitate easier development of efficient makefiles, the following “smart action” scripting command is provided with the Quartus II software:

`quartus_sh --determine_smart_action` ↵

Because assignments for a Quartus II project are stored in the Quartus II Settings File (**.qsf**), including it in every rule results in unnecessary processing steps. For example, updating a setting related to programming file generation (which requires re-running only `quartus_asm`) modifies the Quartus II Settings File, requiring a complete recompilation if the Quartus II Settings File is included in every rule.

The smart action command determines the earliest command-line executable in the compilation flow that must be run based on the current Quartus II Settings File, and generates a change file corresponding to that executable. For a given command-line executable named `quartus_`*<executable>*, the change file is named with the format *<executable>***.chg**. For example, if `quartus_map` must be re-run, the smart action command creates or updates a file named **map.chg**. Thus, rather than including the Quartus II Settings File in each makefile rule, include only the appropriate change file.

Example 2–6 uses change files and the smart action command. You can copy and modify it for your own use. A copy of this example is included in the help for the makefile option, which is available by typing:

`quartus_sh --help=makefiles` ↵

Example 2–6. Sample Makefile (Part 1 of 2)

```
###################################################################
# Project Configuration:
#
# Specify the name of the design (project), the Quartus II Settings
# File (.qsf), and the list of source files used.
###################################################################

PROJECT = chiptrip
SOURCE_FILES = auto_max.v chiptrip.v speed_ch.v tick_cnt.v time_cnt.v
ASSIGNMENT_FILES = chiptrip.qpf chiptrip.qsf

###################################################################
# Main Targets
#
# all: build everything
# clean: remove output files and database
###################################################################
```

Example 2–6. Sample Makefile (Part 2 of 2)

```
all: smart.log $(PROJECT).asm.rpt $(PROJECT).tan.rpt

clean:
	rm -rf *.rpt *.chg smart.log *.htm *.eqn *.pin *.sof *.pof db

map: smart.log $(PROJECT).map.rpt
fit: smart.log $(PROJECT).fit.rpt
asm: smart.log $(PROJECT).asm.rpt
tan: smart.log $(PROJECT).tan.rpt
smart: smart.log

###################################################################
# Executable Configuration
###################################################################

MAP_ARGS = --family=Stratix
FIT_ARGS = --part=EP1S20F484C6
ASM_ARGS =
TAN_ARGS =

###################################################################
# Target implementations
###################################################################

STAMP = echo done >

$(PROJECT).map.rpt: map.chg $(SOURCE_FILES)
	quartus_map $(MAP_ARGS) $(PROJECT)
	$(STAMP) fit.chg

$(PROJECT).fit.rpt: fit.chg $(PROJECT).map.rpt
	quartus_fit $(FIT_ARGS) $(PROJECT)
	$(STAMP) asm.chg
	$(STAMP) tan.chg

$(PROJECT).asm.rpt: asm.chg $(PROJECT).fit.rpt
	quartus_asm $(ASM_ARGS) $(PROJECT)

$(PROJECT).tan.rpt: tan.chg $(PROJECT).fit.rpt
	quartus_tan $(TAN_ARGS) $(PROJECT)
smart.log: $(ASSIGNMENT_FILES)
	quartus_sh --determine_smart_action $(PROJECT) > smart.log

###################################################################
# Project initialization
###################################################################

$(ASSIGNMENT_FILES):
	quartus_sh --prepare $(PROJECT)

map.chg:
	$(STAMP) map.chg
fit.chg:
	$(STAMP) fit.chg
tan.chg:
	$(STAMP) tan.chg
asm.chg:
	$(STAMP) asm.chg
```

A Tcl script is provided with the Quartus II software to create or modify files that can be specified as dependencies in the make rules, assisting you in makefile development. Complete information about this Tcl script and how to integrate it with makefiles is available by running the following command:

```
quartus_sh --help=determine_smart_action ↵
```

The MegaWizard Plug-In Manager

The MegaWizard™ Plug-In Manager and associated MegaWizard Plug-Ins provide a GUI-based flow to configure variation files. However, you can use command-line options to modify, update, or create variation files without using the GUI. This capability is useful in a fully scripted design flow, or in cases where you want to generate variation files without using the wizard GUI flow.

The MegaWizard Plug-In Manager has three functions:

- Providing an interface for you to select the output file or files
- Running a specific MegaWizard Plug-In
- Creating output files (such as variation files, symbol files, and simulation netlist files)

Each MegaWizard Plug-In provides a user interface you use to configure the Plug-In variation, and performs validation and error-checking of your selected ports and parameters.

When you create or update a Plug-In variation in the GUI, the parameters and values are entered through the GUI provided by the Plug-In. The he Plug-In Manager then generates the required files. When you create a Plug-In variation with the command line, you provide the parameters and values as command-line options.

The MegaWizard Plug-In Manager command-line executable is `qmegawiz`. Example 2–7 shows how to create a new variation file.

Example 2–7. MegaWizard Plug-In Manager Command-Line Executable

```
qmegawiz [options] [module=<module name>|wizard=<wizard name>] [<param>=<value> ...
    <port>=<used|unused> ...] [OPTIONAL_FILES=<optional files>] <variation file name>
```

When you use `qmegawiz` to update an existing variation file, the module or wizard name is not required.

If a megafunction changes between software version, the variation files must be updated. To do this, run `qmegawiz -silent` *<variation file name>*.

Table 2–4 describes the supported options.

Table 2–4. qmegawiz Options

Option	Description
`-silent`	Run the MegaWizard Plug-In Manager in command-line mode, without displaying the GUI.
`-f:`*<param file>*	A file that contains all options for the `qmegawiz` command. Refer to "Parameter File" on page 2–18.
`-p:`*<working directory>*	Sets the default working directory. Refer to"Working Directory" on page 2–18.

Refer to “Module and Wizard Names” on page 2–15 for information about specifying the module name or wizard name.

Refer to “Ports and Parameters” on page 2–15 for information about specifying ports and parameters.

Refer to “Optional Files” on page 2–16 for information about generating optional files.

Refer to “Variation File Name” on page 2–18 for information about specifying the variation file name.

Command-Line Support

Only the MegaWizard Plug-Ins listed in Table 2–5 support creation and update with the command-line mode. For Plug-Ins not listed in the table, you must use the MegaWizard Plug-In Manager GUI for creation and update.

Table 2–5. MegaWizard Plug-Ins with Command Line Support (Part 1 of 2)

MegaWizard Plug-In	Wizard Name	Module Name
alt2gxb	ALT2GXB	alt2gxb
alt4gxb	ALTGX	alt4gxb
altclkctrl	ALTCLKCTRL	altclkctrl
altddio_bidir	ALTDDIO_BIDIR	altddio_bidir
altddio_in	ALTDDIO_IN	altddio_in
altddio_out	ALTDDIO_OUT	altddio_out
altecc_decoder	ALTECC	altecc_decoder
altecc_encoder		altecc_encoder
altfp_add_sub	ALTFP_ADD_SUB	altfp_add_sub
altfp_compare	ALTFP_COMPARE	altfp_compare
altfp_convert	ALTFP_CONVERT	altfp_convert
altfp_div	ALTFP_DIV	altfp_div
altfp_mult	ALTFP_MULT	altfp_mult
altfp_sqrt	ALTFP_SQRT	altfp_sqrt
altiobuf_bidir	ALTIOBUF	altiobuf_bidir
altiobuf_in		altiobuf_in
altiobuf_out		altiobuf_out
altlvds_rx	ALTLVDS	altlvds_rx
altlvds_tx		altlvds_tx
altmult_accum	ALTMULT_ACCUM (MAC)	altmult_accum
altmult_complex	ALTMULT_COMPLEX	altmult_complex
altpll_reconfig	ALTPLL_RECONFIG	altpll_reconfig
altpll	ALTPLL	altpll
altsyncram	RAM: 2-PORT	altsyncram
	RAM: 1-PORT	
	ROM: 1-PORT	

Table 2–5. MegaWizard Plug-Ins with Command Line Support (Part 2 of 2)

MegaWizard Plug-In	Wizard Name	Module Name
dcfifo	FIFO	dcfifo
scfifo		scfifo

Module and Wizard Names

You must specify the wizard or module name, shown in Table 2–5, as a command-line option when you create a variation file. Use the option `module=<module name>` to specify the module, or use the option `wizard=<wizard name>` to specify the wizard. If there are spaces in the wizard or module name, enclose the name in double quotes, for example:

```
wizard="RAM: 2-PORT"
```

When there is a one-to-one mapping between the MegaWizard Plug-In and the wizard name and the module name, you can use either the wizard option or the module option.

When there are multiple wizard names that correspond to one module name, you should use the wizard option to specify one wizard.

When there are multiple module names that correspond to one wizard name, you should use the module option to specify one module. For example, use the module option if you create a FIFO because one wizard is common to both modules. However, you should use the wizard option if you create a RAM, because one module is common to three wizards.

If you edit or update an existing variation file, the wizard or module option is not necessary, because information about the wizard or module is already in the variation file.

Ports and Parameters

Ports and parameters for each MegaWizard Plug-In are described in Quartus II Help, and in the Megafunction User Guides on Altera's website. You should use these references to determine appropriate values for each port and parameter required for a particular variation configuration. Refer to "Strategies to Determine Port and Parameter Values" for more information. You do not have to specify every port and parameter supported by a Plug-In. The MegaWizard Plug-In Manager uses default values for any port or parameter you do not specify.

Specify ports as used or unused, for example:

```
<port>=used
<port>=unused
```

You can specify port names in any order. Grouping does not matter. Separate port configuration options from each other with spaces.

Specify a value for a parameter with the equal sign, for example:

```
<parameter>=<value>
```

You can specify parameters in any order. Grouping does not matter. Separate parameter configuration options from each other with spaces. You can specify port names and parameter names in upper or lower case; case does not matter.

All MegaWizard Plug-Ins allow you to specify the target device family with the INTENDED_DEVICE_FAMILY parameter as shown in the following example:

```
qmegawiz wizard=<wizard> INTENDED_DEVICE_FAMILY="Cyclone III" <file>
```

You must specify enough ports and parameters to create a legal configuration of the Plug-In. When you use the GUI flow, each MegaWizard Plug-In performs validation and error checking for the particular ports and parameters you choose. When you use command-line options to specify ports and parameters, you must ensure that the ports and parameters you use are complete and valid for your particular configuration.

For example, when you use a RAM Plug-In to configure a RAM to be 32 words deep, the Plug-In automatically configures an address port that is five bits wide. If you use the command-line flow to configure a RAM that is 32 words deep, you must use one option to specify the depth of the RAM, then calculate the width of the address port and specify that width with another option.

Invalid Configurations

If the combination of default and specified ports and parameters is not complete to create a legal configuration of the Plug-In, qmegawiz generates an error message that indicates what is missing and what values are supported. If the combination of default and specified ports and parameters results in an illegal configuration of the Plug-In, qmegawiz generates an error message that indicates what is illegal, and displays the legal values.

Strategies to Determine Port and Parameter Values

For simple Plug-In variations, it is often easy to determine appropriate port and parameter values with the information in Help and the Megafunction User Guides. Determining that a 32-word-deep RAM requires an address port that is five bits wide is straightforward. For complex Plug-In variations, an option in the GUI may affect multiple port and parameter settings, so it can be difficult to determine a complete set of ports and parameters. In this case, you should use the GUI to generate a variation file which includes the ports and parameters for your desired configuration. Open the variation file in a text editor and use the port and parameter values in the variation file as command-line options.

Optional Files

In addition to the variation file, the MegaWizard Plug-In Manager can generate other files, such as instantiation templates, simulation netlists, and symbols for graphic design entry. Use the OPTIONAL_FILES parameter to control whether the MegaWizard Plug-In Manager generates optional files. Table 2–6 lists valid arguments for the OPTIONAL_FILES parameter.

Table 2–6. Arguments for the OPTIONAL_FILES Parameter (Part 1 of 2)

Argument	Description
INST	Controls the generation of the <*variation*>**_inst.v** file.
INC	Controls the generation of the <*variation*>**.inc** file.
CMP	Controls the generation of the <*variation*>**.cmp** file.
BSF	Controls the generation of the <*variation*>**.bsf** file.

Table 2–6. Arguments for the OPTIONAL_FILES Parameter (Part 2 of 2)

Argument	Description
BB	Controls the generation of the <*variation*>**_bb.v** file.
SIM_NETLIST	Controls the generation of simulation netlist file, wherever there is wizard support.
SYNTH_NETLIST	Controls the generation of the synthesis netlist file, wherever there is wizard support.
ALL	Generates all applicable optional files.
NONE	Disables the generation of all optional files.

Specify a single optional file, for example:

OPTIONAL_FILES=<*argument*>

Specify multiple optional files separated by a vertical bar character, for example:

OPTIONAL_FILES=<*argument 1*>|...|<*argument n*>

If you prefix an argument with a dash (for example, -BB), it is excluded from the generated optional files. If any of the optional files exist when you run qmegawiz and they are excluded in the OPTIONAL_FILES parameter (with the NONE argument, or prefixed with a dash), they are deleted.

You can combine the ALL argument with other excluded arguments to generate "all files except <*excluded files*>." You can combine the NONE argument with other included arguments to generate "no files except <*files*>."

When you combine multiple arguments, they are processed from left to right, and arguments evaluated later have precedence over arguments evaluated earlier. Therefore, the ALL or NONE argument should be the first one in a combination of multiple arguments. When ALL is the first argument, all optional files are generated before exclusions are processed (deleted). When NONE is the first argument, none of the optional files are generated (in other words, any that exist are deleted), then any you specify are generated.

Table 2–7 shows examples for the OPTIONAL_FILES parameter and describes the result of each example.

Table 2–7. Examples of Different Optional File Arguments

Example Values for OPTIONAL_FILES	Description
BB	The optional file <*variation*>**_bb.v** is generated, and no optional files are deleted
BB|INST	The optional file <*variation*>**_bb.v** is generated, then the optional file <*variation*>**_inst.v** is generated, and no optional files are deleted.
NONE	No optional files are generated, and any existing optional files are deleted.
NONE|INC|BSF	Any existing optional files are deleted, then the optional file <*variation*>**.inc** is generated, then the optional file <*variation*>**.bsf** is generated.
ALL|-INST	All optional files are generated, then <*variation*>**_inst.v** is deleted if it exists.
-BB	The optional file <*variation*>**_bb.v** is deleted if it exists
-BB|INST	The optional file <*variation*>**_bb.v** is deleted if it exists, then the optional file <*variation*>**_inst.v** is generated.

The `qmegawiz` command accepts the `ALL` argument combined with other included file arguments, for example, `ALL|BB`, but that combination is equivalent to `ALL` because first all optional files are generated, then the file **<*variation*>_bb.v** is generated (again). Additionally, the software accepts the `NONE` argument combined with other excluded file arguments, for example, `NONE|-BB`, but that combination is equivalent to `NONE` because no optional files are generated (any that exist are deleted), then the file **<*variation*>_bb.v** is deleted if it exists.

Parameter File

You can put all the parameter values and port values in a file, and pass the file name as an argument to `qmegawiz` with the `-f:`<*parameter file*> option. For example, you could run the following command:

```
qmegawiz -silent module=altsyncram -f:rom_params.txt myrom.v ↵
```

The following options are in a file called **rom_params.txt**:

```
RAM_BLOCK_TYPE=M4K DEVICE_FAMILY=Stratix WIDTH_A=5 WIDTHAD_A=5
   NUMWORDS_A=32 INIT_FILE=rom.hex OPERATION_MODE=ROM
```

Working Directory

You can change the working directory that `qmegawiz` uses when it generates files. By default, the working directory is the current directory when you execute the `qmegawiz` command. Use the `-p` option to specify a different working directory, for example:

```
-p:<working directory>
```

You can specify the working directory with an absolute or relative path. Specify an alternative working directory any time you do not want files generated in the current directory. The alternative working directory can be useful if you generate multiple variations in a batch script, and keep generated files for the different Plug-In variations in separate directories.

Variation File Name

The language used for a variation file depends on the file extension of the variation file name. The MegaWizard Plug-In Manager creates HDL output files in a language based on the file name extension. Therefore, you must always specify a complete file name, including file extension, as the last argument to the `qmegawiz` command. Table 2–8 shows the file extension that corresponds to supported HDL types.

Table 2–8. Variation File Extensions

Variation File HDL Type	Required File Extension
Verilog HDL	**.v**
VHDL	**.vhd**
AHDL	**.tdf**

Command-Line Scripting Examples

This section of the chapter presents various examples of command-line executable use.

Create a Project and Apply Constraints

The command-line executables include options for common global project settings and commands. To apply constraints such as pin locations and timing assignments, run a Tcl script with the constraints in it. You can write a Tcl constraint file from scratch, or generate one for an existing project. From the Project menu, click **Generate Tcl File for Project**.

Example 2–8 creates a project with a Tcl script and applies project constraints using the tutorial design files in the <*Quartus II installation directory*>**/qdesigns/fir_filter/** directory.

Example 2–8. Tcl Script to Create Project and Apply Constraints

```
project_new filtref -overwrite
# Assign family, device, and top-level file
set_global_assignment -name FAMILY Cyclone
set_global_assignment -name DEVICE EP1C12F256C6
set_global_assignment -name BDF_FILE filtref.bdf
# Assign pins
set_location_assignment -to clk Pin_28
set_location_assignment -to clkx2 Pin_29
set_location_assignment -to d[0] Pin_139
set_location_assignment -to d[1] Pin_140
# Other pin assignments could follow
# Create timing assignments
create_base_clock -fmax "100 MHz" -target clk clocka
create_relative_clock -base_clock clocka -divide 2 -offset "500 ps" -target clkx2 clockb
set_multicycle_assignment -from clk -to clkx2 2
# Other timing assignments could follow
project_close
```

Save the script in a file called **setup_proj.tcl** and type the commands illustrated in Example 2–9 at a command prompt to create the design, apply constraints, compile the design, and perform fast-corner and slow-corner timing analysis. Timing analysis results are saved in two files.

Example 2–9. Script to Create and Compile a Project

```
quartus_sh -t setup_proj.tcl ↵
quartus_map filtref ↵
quartus_fit filtref ↵
quartus_asm filtref ↵
quartus_tan filtref --fast_model --tao=min.tao --export_settings=off ↵
quartus_tan filtref --tao=max.tao --export_settings=off ↵
```

You can use the following two commands to create the design, apply constraints, and compile the design:

```
quartus_sh -t setup_proj.tcl ↵
quartus_sh --flow compile filtref ↵
```

The `quartus_sh --flow compile` command performs a full compilation, and is equivalent to clicking the Start Compilation button in the toolbar.

Check Design File Syntax

The UNIX shell script example shown in Example 2–10 assumes that the Quartus II software **fir_filter** tutorial project exists in the current directory. (You can find the **fir_filter** project in the <*Quartus II directory*>**/qdesigns/fir_filter** directory unless the Quartus II software tutorial files are not installed.)

The `--analyze_file` option performs a syntax check on each file. The script checks the exit code of the `quartus_map` executable to determine whether there is an error during the syntax check. Files with syntax errors are added to the `FILES_WITH_ERRORS` variable, and when all files are checked, the script prints a message indicating syntax errors. When options are not specified, the executable uses the project database values. If not specified in the project database, the executable uses the Quartus II software default values. For example, the **fir_filter** project is set to target the Cyclone device family, so it is not necessary to specify the `--family` option.

Example 2–10. Shell Script to Check Design File Syntax

```
#!/bin/sh
FILES_WITH_ERRORS=""
# Iterate over each file with a .bdf or .v extension
for filename in `ls *.bdf *.v`
do
# Perform a syntax check on the specified file
    quartus_map fir_filter --analyze_file=$filename
  # If the exit code is non-zero, the file has a syntax error
    if [ $? -ne 0 ]
    then
       FILES_WITH_ERRORS="$FILES_WITH_ERRORS $filename"
    fi
done
if [ -z "$FILES_WITH_ERRORS" ]
then
    echo "All files passed the syntax check"
    exit 0
else
    echo "There were syntax errors in the following file(s)"
    echo $FILES_WITH_ERRORS
    exit 1
fi
```

Create a Project and Synthesize a Netlist Using Netlist Optimizations

This example creates a new Quartus II project with a file **top.edf** as the top-level entity. The `--enable_register_retiming=on` and `--enable_wysiwyg_resynthesis=on` options allow the technology mapper to optimize the design using gate-level register retiming and technology remapping.

For more details about register retiming, WYSIWYG primitive resynthesis, and other netlist optimization options, refer to the Quartus II Help.

The `--part` option tells the technology mapper to target an EP20K600EBC652-1X device. To create the project and synthesize it using the netlist optimizations described above, type the command shown in Example 2–11 at a command prompt.

© March 2009 Altera Corporation

Example 2–11. Creating a Project and Synthesizing a Netlist Using Netlist Optimizations

```
quartus_map top --source=top.edf --enable_register_retiming=on
   --enable_wysiwyg_resynthesis=on --part=EP20K600EBC652-1X ↵
```

Archive and Restore Projects

You can archive or restore a Quartus II project with a single command. This makes it easy to take snapshots of projects when you use batch files or shell scripts for compilation and project management. Use the `--archive` or `--restore` options for `quartus_sh` as appropriate. Type the command shown in Example 2–12 at a system command prompt to archive your project.

Example 2–12. Archiving a Project

```
quartus_sh --archive <project name> ↵
```

The archive file is automatically named <*project name*>**.qar.** If you want to use a different name, rename the archive after it has been created. This command overwrites any existing archive with the same name.

To restore a project archive, type the command shown in Example 2–13 at a system command prompt.

Example 2–13. Restoring a Project Archive

```
quartus_sh --restore <archive name> ↵
```

The command restores the project archive to the current directory and overwrites existing files.

Perform I/O Assignment Analysis

You can perform I/O assignment analysis with a single command. I/O assignment analysis checks pin assignments to ensure they do not violate board layout guidelines. I/O assignment analysis does not require a complete place and route, so it is a quick way to ensure your pin assignments are correct. The command shown in Example 2–14 performs I/O assignment analysis for the specified project and revision.

Example 2–14. Performing I/O Assignment Analysis

```
quartus_fit --check_ios <project name> --rev=<revision name> ↵
```

Update Memory Contents without Recompiling

You can use two simple commands to update the contents of memory blocks in your design without recompiling. Use the `quartus_cdb` executable with the `--update_mif` option to update memory contents from Memory Initialization Files or Hexadecimal (Intel-Format) Files. Then re-run the assembler with the `quartus_asm` executable to regenerate the SOF, POF, and any other programming files.

Example 2–15 shows these two commands.

Example 2–15. Commands to Update Memory Contents without Recompiling

```
quartus_cdb --update_mif <project name> [--rev=<revision name>]↵
quartus_asm <project name> [--rev=<revision name>]↵
```

Example 2–16 shows the commands for a DOS batch file for this example. You can paste the following lines into a DOS batch file called **update_memory.bat.**

Example 2–16. Batch file to Update Memory Contents without Recompiling

```
quartus_cdb --update_mif %1 --rev=%2
quartus_asm %1 --rev=%2
```

Type the following command at a system command prompt:

```
update_memory.bat <project name> <revision name> ↵
```

Create a Compressed Configuration File

You can create a compressed configuration file in two ways. The first way is to run `quartus_cpf` with an option file that turns on compression. The second way is to run `quartus_cpf` with a Conversion Setup File (**.cof**).

To create an option file that turns on compression, type

```
quartus_cpf -w <filename>.opt ↵
```

This interactive command walks you through some questions, including compression, then creates an option file based on your answers. Use the `--option` option to `quartus_cpf` to specify the option file you just created. For example, the following command creates a compressed Programming Object File (**.pof**) that targets an EPCS64 device:

```
quartus_cpf --convert --option=<filename>.opt --device=EPCS64 <file>.sof <file>.pof ↵
```

Alternatively, you can use the Convert Programming Files utility in the Quartus II software to create a conversion setup file. Configure any options you want, including compression, then save the conversion setup. Use the following command to run the conversion setup you specified:

```
quartus_cpf <file>.cof ↵
```

Fit a Design as Quickly as Possible

This example assumes that a project called **top** exists in the current directory, and that the name of the top-level entity is **top**. The `--effort=fast` option causes the Fitter to use the fast fit algorithm to increase compilation speed, possibly at the expense of reduced f_{MAX} performance. The `--one_fit_attempt=on` option restricts the Fitter to only one fitting attempt for the design.

To attempt to fit the project called **top** as quickly as possible, type the command shown in Example 2–17 at a command prompt.

Example 2–17. Fitting a Project Quickly

```
quartus_fit top --effort=fast --one_fit_attempt=on ↵
```

Fit a Design Using Multiple Seeds

This shell script example assumes that the Quartus II software tutorial project called **fir_filter** exists in the current directory (defined in the file **fir_filter.qpf**). If the tutorial files are installed on your system, this project exists in the <*Quartus II directory*>/qdesigns<*quartus_version_number*> /**fir_filter** directory. Because the top-level entity in the project does not have the same name as the project, you must specify the revision name for the top-level entity with the `--rev` option. The `--seed` option specifies the seeds to use for fitting.

A seed is a parameter that affects the random initial placement of the Quartus II Fitter. Varying the seed can result in better performance for some designs.

After each fitting attempt, the script creates new directories for the results of each fitting attempt and copies the complete project to the new directory so that the results are available for viewing and debugging after the script has completed.

Example 2–18 is designed for use on UNIX systems using **sh** (the shell).

Example 2–18. Shell Script to Fit a Design Using Multiple Seeds

```
#!/bin/sh
ERROR_SEEDS=""
quartus_map fir_filter --rev=filtref
# Iterate over a number of seeds
for seed in 1 2 3 4 5
do
echo "Starting fit with seed=$seed"
# Perform a fitting attempt with the specified seed
quartus_fit fir_filter --seed=$seed --rev=filtref
# If the exit-code is non-zero, the fitting attempt was
# successful, so copy the project to a new directory
if [ $? -eq 0 ]
then
          mkdir ../fir_filter-seed_$seed
          mkdir ../fir_filter-seed_$seed/db
          cp * ../fir_filter-seed_$seed
          cp db/* ../fir_filter-seed_$seed/db
else
          ERROR_SEEDS="$ERROR_SEEDS $seed"
fi
done
if [ -z "$ERROR_SEEDS" ]
then
echo "Seed sweeping was successful"
exit 0
else
echo "There were errors with the following seed(s)"
echo $ERROR_SEEDS
exit 1
fi
```

Use the Design Space Explorer (DSE) included with the Quartus II software script (by typing `quartus_sh --dse` ↵ at a command prompt) to improve design performance by performing automated seed sweeping.

For more information about the DSE, type `quartus_sh --help=dse` ↵ at the command prompt, or refer to the *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*, or see the Quartus II Help.

The QFlow Script

A Tcl/Tk-based graphical interface called QFlow is included with the command-line executables. You can use the QFlow interface to open projects, launch some of the command-line executables, view report files, and make some global project assignments. The QFlow interface can run the following command-line executables:

- `quartus_map` (Analysis and Synthesis)
- `quartus_fit` (Fitter)
- `quartus_tan` (Timing Analysis)
- `quartus_asm` (Assembler)
- `quartus_eda` (EDA Netlist Writer)

To view floorplans or perform other GUI-intensive tasks, launch the Quartus II software.

Start QFlow by typing the following command at a command prompt:

```
quartus_sh -g ↵
```

Figure 2–3 shows the QFlow interface.

Figure 2–3. QFlow Interface

The QFlow script is located in the <*Quartus II directory*>**/common/tcl/apps/qflow/** directory.

Referenced Documents

This chapter references the following documents:

- *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Reference Manual*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 2–9 shows the revision history for this chapter.

Table 2–9. Document Revision History

Date / Version	Changes Made	Summary of Changes
March 2009, v9.0.0	No change to content.	Updated for the Quartus II software version 9.0 release.

November 2008, v8.1.0	Added the following sections:
	■ "The MegaWizard Plug-In Manager" on page 2–13
	  ■ "Command-Line Support" on page 2–14
	  ■ "Module and Wizard Names" on page 2–15
	  ■ "Ports and Parameters" on page 2–15
	  ■ "Invalid Configurations" on page 2–16
	  ■ "Strategies to Determine Port and Parameter Values" on page 2–16
	  ■ "Optional Files" on page 2–16
	  ■ "Parameter File" on page 2–18
	  ■ "Working Directory" on page 2–18
	  ■ "Variation File Name" on page 2–18
	■ "Create a Compressed Configuration File" on page 2–22
	Updated "Option Precedence" on page 2–7 to clarify how to control precedence
	Corrected Example 2–5 on page 2–10
	Changed Example 2–1, Example 2–2, Example 2–4, and Example 2–7 to use the EP1C12F256C6 device
	Minor editorial updates
	Updated entire chapter using 8½" × 11" chapter template
		Updated for the Quartus II software version 8.1 release.

May 2008, v8.0.0	Updated "Referenced Documents" on page 2–20. Updated references in document.	Updated for the Quartus II software version 8.0.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

3. Tcl Scripting

QII52003-9.0.0

Introduction

Developing and running Tcl scripts to control the Altera® Quartus® II software allows you to perform a wide range of functions, such as compiling a design or writing procedures to automate common tasks.

You can use Tcl scripts to manage a Quartus II project, make assignments, define design constraints, make device assignments, run compilations, perform timing analysis, import LogicLock™ region assignments, use the Quartus II Chip Editor, and access reports. You can automate your Quartus II assignments using Tcl scripts so that you do not have to create them individually. Tcl scripts also facilitate project or assignment migration. For example, when using the same prototype or development board for different projects, you can automate reassignment of pin locations in each new project. The Quartus II software can also generate a Tcl script based on all the current assignments in the project, which aids in switching assignments to another project.

The Quartus II software Tcl commands follow the EDA industry Tcl application programming interface (API) standards for using command-line options to specify arguments. This simplifies learning and using Tcl commands. If you encounter an error using a command argument, the Tcl interpreter gives help information showing correct usage.

This chapter includes sample Tcl scripts for automating the Quartus II software. You can modify these example scripts for use with your own designs. You can find more Tcl scripts in the Design Examples section of the Support area of Altera's website.

This chapter includes the following topics:

- "Quartus II Tcl Packages" on page 3–2
- "Quartus II Tcl API Help" on page 3–4
- "Executables Supporting Tcl" on page 3–7
- "End-to-End Design Flows" on page 3–9
- "Creating Projects and Making Assignments" on page 3–10
- "Compiling Designs" on page 3–14
- "Reporting" on page 3–15
- "Timing Analysis" on page 3–18
- "Automating Script Execution" on page 3–22
- "Other Scripting Features" on page 3–25
- "Using the Quartus II Tcl Shell in Interactive Mode" on page 3–29
- "Quartus II Legacy Tcl Support" on page 3–32
- "Using the tclsh Shell" on page 3–32
- "Tcl Scripting Basics" on page 3–33

What is Tcl?

Tcl (pronounced "tickle") is a popular scripting language that is similar to many shell scripting and high-level programming languages. It provides support for control structures, variables, network socket access, and APIs. Tcl is the EDA industry-standard scripting language used by Synopsys, Mentor Graphics®, and Altera software. It allows you to create custom commands and works seamlessly across most development platforms. For a list of recommended literature on Tcl, refer to "External References" on page 3–39.

You can create your own procedures by writing scripts containing basic Tcl commands, user-defined procedures, and Quartus II API functions. You can then automate your design flow, run the Quartus II software in batch mode, or execute the individual Tcl commands interactively in the Quartus II Tcl interactive shell.

If you're unfamiliar with Tcl scripting, or are a Tcl beginner, refer to "Tcl Scripting Basics" on page 3–33 for an introduction to Tcl scripting.

The Quartus II software, beginning with version 4.1, supports Tcl/Tk version 8.4, supplied by the Tcl DeveloperXchange at tcl.activestate.com.

Quartus II Tcl Packages

The Quartus II Tcl commands are grouped in packages by function. Table 3–1 describes each Tcl package.

Table 3–1. Tcl Packages (Part 1 of 2)

Package Name	Package Description
advanced_timing	Traverse the timing netlist and get information about timing nodes
backannotate	Back annotate assignments
chip_planner	Identify and modify resource usage and routing with the Chip Editor
database_manager	Manage version-compatible database files
device	Get device and family information from the device database
flow	Compile a project, run command-line executables and other common flows
insystem_memory_edit	Read and edit memory contents in Altera devices
jtag	Control the JTAG chain
logic_analyzer_interface	Query and modify the logic analyzer interface output pin state
logiclock	Create and manage LogicLock regions
misc	Perform miscellaneous tasks
project	Create and manage projects and revisions, make any project assignments including timing assignments
report	Get information from report tables, create custom reports
sdc	Specifies constraints and exceptions to the TimeQuest Timing Analyzer
simulator	Configure and perform simulations
sta	Contains the set of Tcl functions for obtaining advanced information from the Quartus II TimeQuest Timing Analyzer
stp	Run the SignalTap® II Logic Analyzer
timing	Annotate timing netlist with delay information, compute and report timing paths

Table 3–1. Tcl Packages (Part 2 of 2)

Package Name	Package Description
timing_assignment	Contains the set of Tcl functions for making project-wide timing assignments, including clock assignments; all Tcl commands designed to process Quartus II Classic Timing Analyzer assignments have been moved to this package
timing_report	List timing paths
sdc_ext	Altera-specific SDC commands

By default, only the minimum number of packages is loaded automatically with each Quartus II executable. This keeps the memory requirement for each executable as low as possible. Because the minimum number of packages is automatically loaded, you must load other packages before you can run commands in those packages.

Table 3–2 lists the Quartus II Tcl packages available with Quartus II executables and indicates whether a package is loaded by default (●) or is available to be loaded as necessary (◓). A white circle (○) means that the package is not available in that executable.

Table 3–2. Tcl Package Availability by Quartus II Executable (Part 1 of 2)

	Quartus II Executable						
Packages	**Quartus_sh**	**Quartus_tan**	**Quartus_cdb**	**Quartus_sim**	**Quartus_stp**	**Quartus_sta Quartus_staw**	**Tcl Console**
advanced_timing	○	◓	○	○	○	○	○
backannotate	○	○	◓	○	○	○	◓
chip_planner	○	○	◓	○	○	○	○
device	●	◓	●	●	○	●	◓
flow	◓	◓	◓	◓	○	◓	◓
insystem_memory_edit	○	○	○	○	●	○	○
jtag	○	○	○	○	●	○	○
logic_analyzer_interface	○	○	○	○	●	○	○
logiclock	○	◓	◓	○	○	○	◓
misc	●	●	●	●	●	●	●
old_api	○	○	○	○	○	○	●
project	●	●	●	●	●	●	●
report	◓	◓	◓	●	○	●	◓
sdc	○	○	○	○	○	●	○
sdc_ext	○	○	○	○	○	●	○
simulator	○	○	○	●	○	○	○

Table 3–2. Tcl Package Availability by Quartus II Executable (Part 2 of 2)

Packages	Quartus II Executable						
	Quartus_sh	Quartus_tan	Quartus_cdb	Quartus_sim	Quartus_stp	Quartus_sta Quartus_staw	Tcl Console
sta	○	○	○	○	○	●	○
stp	○	○	○	○	●	○	○
timing	○	●	○	○	○	○	○
timing_assignmen t	●	●	●	●	●	○	○
timing_report	○	◓	○	○	●	○	●

Notes to Table 3–2:

(1) A dark circle (●) indicates that the package is loaded automatically.

(2) A half-circle (◓) means that the package is available but not loaded automatically.

(3) A white circle (○) means that the package is not available for that executable.

Because different packages are available in different executables, you must run your scripts with executables that include the packages you use in the scripts. For example, if you use commands in the **timing** package, you must use the `quartus_tan` executable to run the script because the `quartus_tan` executable is the only one with support for the **timing** package.

Loading Packages

To load a Quartus II Tcl package, use the `load_package` command as follows:

```
load_package [-version <version number>] <package name>
```

This command is similar to the `package require` Tcl command (described in Table 3–3 on page 3–5), but you can easily alternate between different versions of a Quartus II Tcl package with the `load_package` command.

For additional information about these and other Quartus II command-line executables, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook.*

Quartus II Tcl API Help

Access the Quartus II Tcl API Help reference by typing the following command at a system command prompt:

```
quartus_sh --qhelp ↵
```

This command runs the Quartus II Command-Line and Tcl API help browser, which documents all commands and options in the Quartus II Tcl API. It includes detailed descriptions and examples for each command.

In addition, the information in the Tcl API help is available in the *Quartus II Scripting Reference Manual*.

© March 2009 Altera Corporation

Quartus II Tcl help allows easy access to information about the Quartus II Tcl commands. To access the help information, type `help` at a Tcl prompt, as shown in Example 3–1.

Example 3–1. Help Output

```
tcl> help
----------------------------------------------------------------------
---

----------------------------------
Available Quartus II Tcl Packages:
----------------------------------

Loaded                          Not Loaded
----------------------------    -----------------------
::quartus::misc                 ::quartus::device
::quartus::old_api              ::quartus::backannotate
::quartus::project              ::quartus::flow
::quartus::timing_assignment    ::quartus::logiclock
::quartus::timing_report        ::quartus::report


* Type "help -tcl"
to get an overview on Quartus II Tcl usages.
```

Using the `-tcl` option with `help` displays an introduction to the Quartus II Tcl API that focuses on how to get help for Tcl commands (short help and long help) and Tcl packages.

The Tcl API help is also available in Quartus II online help. Search for the command or package name to find details about that command or package.

Table 3–3 summarizes the help options available in the Tcl environment.

Table 3–3. Help Options Available in the Quartus II Tcl Environment (Part 1 of 2)

Help Command	Description
`help`	To view a list of available Quartus II Tcl packages, loaded and not loaded.
`help -tcl`	To view a list of commands used to load Tcl packages and access command-line help.
`help -pkg` *`<package_name>`* `[-version` *`<version number>`*`]`	To view help for a specified Quartus II package that includes the list of available Tcl commands. For convenience, you can omit the `::quartus::` package prefix, and type `help -pkg` *<package name>* ↵. If you do not specify the `-version` option, help for the currently loaded package is displayed by default. If the package for which you want help is not loaded, help for the latest version of the package is displayed by default. Examples: `help -pkg ::quartus::project` ↵ `help -pkg project` ↵ `help -pkg project -version 1.0` ↵

Table 3–3. Help Options Available in the Quartus II Tcl Environment (Part 2 of 2)

Help Command	Description

`<command_name> -h`
or
`<command_name> -help`
	To view short help for a Quartus II Tcl command for which the package is loaded.
	Examples:
	`project_open -h` ↵
	`project_open -help` ↵

`package require ::quartus::<package name> [<version>]`	To load a Quartus II Tcl package with the specified version. If <version> is not specified, the latest version of the package is loaded by default. Example: `package require ::quartus::project 1.0` ↵ This command is similar to the **load_package** command. The advantage of using **load_package** is that you can alternate freely between different versions of the same package. Type `<`*package name*`> [-version <`*version number*`>]` ↵ to load a Quartus II Tcl package with the specified version. If the `-version` option is not specified, the latest version of the package is loaded by default. Example: `load_package ::quartus::project -version 1.0` ↵

`help -cmd <command name> [-version <version number>]`
or
`<command name> -long_help`
	To view long help for a Quartus II Tcl command. Only `<`*command name*`> -long_help` requires that the associated Tcl package is loaded.
	If you do not specify the `-version` option, help for the currently loaded package is displayed by default.
	If the package for which you want help is not loaded, help for the latest version of the package is displayed by default.
	Examples:
	`project_open -long_help` ↵
	`help -cmd project_open` ↵
	`help -cmd project_open -version 1.0` ↵

`help -examples`	To view examples of Quartus II Tcl usage.
`help -quartus`	To view help on the predefined global Tcl array that can be accessed to view information about the Quartus II executable that is currently running.
`quartus_sh --qhelp`	To launch the Tk viewer for Quartus II command-line help and display help for the command-line executables and Tcl API packages. For more information about this utility, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Executables Supporting Tcl

Some of the Quartus II command-line executables support Tcl scripting (refer to Table 3–4). Each executable supports different sets of Tcl packages. Refer to Table 3–4 to determine the appropriate executable to run your script.

Table 3–4. Command-Line Executables Supporting Tcl Scripting

Executable Name	Executable Description
`quartus_sh`	The Quartus II Shell is a simple Tcl scripting shell, useful for making assignments, general reporting, and compiling.
`quartus_tan`	Use the Quartus II Classic Timing Analyzer to perform simple timing reporting and advanced timing analysis.
`quartus_cdb`	The Quartus II Compiler Database supports back annotation, LogicLock region operations, and Chip Editor functions.
`quartus_sim`	The Quartus II Simulator supports the automation of design simulation.
`quartus_sta` `quartus_staw`	The Quartus II TimeQuest Timing Analyzer supports SDC terminology for constraint entry and reporting.
`quartus_stp`	The Quartus II SignalTap II executable supports in-system debugging tools.

The `quartus_tan` and `quartus_cdb` executables support supersets of the packages supported by the `quartus_sh` executable. Use the `quartus_sh` executable if you run Tcl scripts with only project management and assignment commands, or if you require a Quartus II command-line executable with a small memory footprint.

For more information about these command-line executables, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Command-Line Options: -t, -s, and --tcl_eval

Table 3–5 lists three command-line options you can use with executables that support Tcl.

Table 3–5. Command-Line Options Supporting Tcl Scripting

Command-Line Option	Description
`-t <script file> [<script args>]`	Run the specified Tcl script with optional arguments.
`-s`	Open the executable in the interactive Tcl shell mode.
`--tcl_eval <tcl command>`	Evaluate the remaining command-line arguments as Tcl commands. For example, the following command displays help for the project package: `quartus_sh --tcl_eval help -pkg project`

Run a Tcl Script

Running an executable with the `-t` option runs the specified Tcl script. You can also specify arguments to the script. Access the arguments through the `argv` variable, or use a package such as cmdline, which supports arguments of the following form:

```
-<argument name> <argument value>
```

The **cmdline** package is included in the <*Quartus II directory*>**/common/tcl/packages** directory.

For example, to run a script called **myscript.tcl** with one argument, `Stratix`, type the following command at a system command prompt:

```
quartus_sh -t myscript.tcl Stratix ↵
```

Beginning with version 4.1, the Quartus II software supports the `argv` variable. In previous software versions, script arguments are accessed in the `quartus(args)` global variable.

Refer to “Accessing Command-Line Arguments” on page 3–27 for more information.

Interactive Shell Mode

Running an executable with the `-s` option starts an interactive Tcl shell that displays a `tcl>` prompt. For example, type `quartus_tan -s` ↵ at a system command prompt to open the Quartus II Classic Timing Analyzer executable in interactive shell mode. Commands you type in the Tcl shell are interpreted when you click **Enter**. You can run a Tcl script in the interactive shell with the following command:

```
source <script name> ↵
```

If a command is not recognized by the shell, it is assumed to be an external command and executed with the `exec` command.

Evaluate as Tcl

Running an executable with the `--tcl_eval` option causes the executable to immediately evaluate the remaining command-line arguments as Tcl commands. This can be useful if you want to run simple Tcl commands from other scripting languages.

For example, the following command runs the Tcl command that prints out the commands available in the project package.

```
quartus_sh --tcl_eval help -pkg project ↵
```

Using the Quartus II Tcl Console Window

You can run Tcl commands directly in the Quartus II Tcl Console window. On the View menu, click **Utility Windows.** By default, the Tcl Console window is docked in the bottom-right corner of the Quartus II GUI. Everything typed in the Tcl Console is interpreted by the Quartus II Tcl shell.

The Quartus II Tcl Console window supports the Tcl API used in the Quartus II software version 3.0 and earlier for backward compatibility with older designs and EDA tools.

Tcl messages appear in the **System** tab (Messages window). Errors and messages written to `stdout` and `stderr` also are shown in the Quartus II Tcl Console window.

You can do limited timing analysis in the Tcl console in the Quartus II GUI. With the **timing_report** package, you can use the `list_path` command to get details on paths listed in the timing report. However, if you want to get information about timing paths that are not listed in the timing report, you must use the `quartus_tan` executable in shell mode or run a script that reports on the paths in which you are interested.

© March 2009 Altera Corporation

If your design uses the Quartus II TimeQuest Timing Analyzer, you should perform scripted timing analysis in the TimeQuest Tcl console.

As Table 3–2 shows, the Tcl console in the Quartus II GUI does not include support for every package, so you cannot run scripts that use commands in packages that are not supported.

End-to-End Design Flows

You can use Tcl scripts to control all aspects of the design flow, including controlling other software if it includes a scripting interface.

Typically, EDA tools include their own script interpreters that extend core language functionality with tool-specific commands. For example, the Quartus II Tcl interpreter supports all core Tcl commands, and adds numerous commands specific to the Quartus II software. You can include commands in one Tcl script to run another script, which allows you to combine or chain together scripts to control different tools. Because scripts for different tools must be executed with different Tcl interpreters, it is difficult to pass information between the scripts unless one script writes information into a file and another script reads it.

Within the Quartus II software, you can perform many different operations in a design flow (such as synthesis, fitting, and timing analysis) from a single script, making it easy to maintain global state information and pass data between the operations. However, there are some limitations on the operations you can perform in a single script due to the various packages supported by each executable. For example, you cannot write a single script that performs simulation with commands in the **simulator** package while using commands in the **advanced_timing** package; those two packages are not available in the same executable. In a case where you wanted to include Tcl simulation and advanced timing analysis commands, you must write two scripts.

There are no limitations on running flows from any executable. Flows include operations found in the Start section of the Processing menu in the Quartus II GUI, and are also documented with the `execute_flow` Tcl command. If you can make settings in the Quartus II software and run a flow to get your desired result, you can make the same settings and run the same flow in any command-line executable.

To revisit the example with simulation and timing analysis, you could write one script that includes settings that configure a simulation, with settings that configure timing analysis. Then, run the simulation and timing analysis flows with the `execute_flow` command.

Configuring a simulation includes specifying settings such as name and location of the stimulus file, the duration of the simulation, whether to perform glitch detection or not, and more. Configuring timing analysis includes specifying settings such as the required clock frequency, the number of paths to report, and which timing model to use. You can make the settings, then run the flows with the `execute_flow` command, in any Quartus II command-line executable.

Creating Projects and Making Assignments

A benefit of the Tcl scripting API is that you can easily create a script that makes all the assignments for an existing project. You can use the script at any time to restore your project settings to a known state. From the Project menu, click **Generate Tcl File for Project** to automatically generate a **.tcl** file with all of your assignments. You can source this file to recreate your project, and you can edit the file to add other commands, such as compiling the design. The file is a good starting point to learn about project management commands and assignment commands.

Refer to "Interactive Shell Mode" on page 3–8 for information about sourcing a script. Scripting information for all Quartus II project settings and assignments is located in the *QSF Reference Manual*.

Example 3–2 shows how to create a project, make assignments, and compile the project. It uses the **fir_filter** tutorial design files in the **qdesigns** installation directory. Use the `quartus_sh` executable to run this Tcl script.

Example 3–2. Create and Compile a Project

```
load_package flow

# Create the project and overwrite any settings
# files that exist
project_new fir_filter -revision filtref -overwrite
# Set the device, the name of the top-level BDF,
# and the name of the top level entity
set_global_assignment -name FAMILY Cyclone
set_global_assignment -name DEVICE EP1C6F256C6
set_global_assignment -name BDF_FILE filtref.bdf
set_global_assignment -name TOP_LEVEL_ENTITY filtref
# Add other pin assignments here
set_location_assignment -to clk Pin_G1
# Create a base clock and a derived clock
create_base_clock -fmax "100 MHz" -target clk clocka
create_relative_clock -base_clock clocka -divide 2 \
   -offset "500 ps" -target clkx2 clockb
# Create a multicycle assignment of 2 between
# the two clock domains in the design.
set_multicycle_assignment -from clk -to clkx2 2
execute_flow -compile
project_close
```

The assignments created or modified while a project is open are not committed to the Quartus II Settings Files (**.qsf**) unless you explicitly call `export_assignments` or `project_close` (unless `-dont_export_assignments` is specified). In some cases, such as when running `execute_flow`, the Quartus II software automatically commits the changes.

HardCopy Device Design

For information about using a scripted design flow for HardCopy II designs, refer to the *Script-Based Design for HardCopy II Devices* chapter of the *HardCopy Handbook*. It contains sample scripts and recommendations to make your HardCopy II design flow easy.

A separate chapter in the *HardCopy Handbook* called *Timing Constraints for HardCopy II Devices* also contains information about script-based design for HardCopy II devices, with an emphasis on timing constraints.

Using LogicLock Regions

You can use Tcl commands to work with LogicLock™ regions. The following examples show how to export and import LogicLock regions for use in other designs. The examples use the files in the LogicLock tutorial design.

For additional information about the LogicLock design methodology, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

To compile a design and export LogicLock regions, follow these steps:

1. Create a project and add assignments.
2. Assign virtual pins.
3. Create a LogicLock region.
4. Assign a design entity to the region.
5. Compile the project.
6. Back-annotate the region.
7. Export the region.

Example 3–3 shows a script that creates a project called **lockmult**, and makes all the required assignments to compile the project. Next, the script compiles the project, back-annotates the design, and exports the LogicLock region. The script uses a procedure called `assign_virtual_pins`, which is described after the example. The example assumes that a design file named **pipemult.bdf** exists, and that it is a low-level module that would be instantiated multiple times in a higher-level module. Use the `quartus_cdb` executable to run this script.

Example 3–3. LogicLock Export Script

```
load_package flow
load_package logiclock
load_package backannotate

project_new lockmult -overwrite
set_global_assignment -name BDF_FILE pipemult.bdf
set_global_assignment -name FAMILY Cyclone
set_global_assignment -name DEVICE EP1C6F256C6
set_global_assignment -name TOP_LEVEL_ENTITY pipemult

# These two assignments cause the Quartus II software
# to generate a VQM file for the logic in the LogicLock
# region. The VQM file is imported into the top-level
# design.
set_global_assignment -name \
   LOGICLOCK_INCREMENTAL_COMPILE_FILE pipemult.vqm
set_global_assignment -name \
   LOGICLOCK_INCREMENTAL_COMPILE_ASSIGNMENT ON

create_base_clock -fmax 200MHz -target clk clk_200
assign_virtual_pins { clk }
#Prepare LogicLock data structures before
#LogicLock-related commands.
initialize_logiclock

# Create a region named lockmult and assign pipemult
# to it.
# The region is auto-sized and floating.
set_logiclock -region lockmult -auto_size true \
-floating true
set_logiclock_contents -region lockmult -to pipemult
execute_flow -compile

# Back annotate the LogicLock Region and export a QSF
logiclock_back_annotate -region lockmult -lock
logiclock_export -file_name pipemult.qsf


uninitialize_logiclock
project_close
```

The `assign_virtual_pins` command is a procedure that makes virtual pin assignments to all bottom-level design pins, except for signals specified as arguments to the procedure. The procedure is defined in Example 3–4.

Example 3–4. assign_virtual_pins Procedure

```
proc assign_virtual_pins { skips } {

    # Analysis and elaboration must be run first to get the pin names
    execute_flow -analysis_and_elaboration

    # Get all pin names as a collection.

    set name_ids [get_names -filter * -node_type pin]
    foreach_in_collection name_id $name_ids {
        # Get the hierarchical path name of the pin.
        set hname [get_name_info -info full_path $name_id]
        #Skip the virtual pin assignment if the
        #pin is in the list of signals to be skipped.
        if {[lsearch -exact $skips $hname] == -1} {
            post_message "Setting VIRTUAL_PIN on $hname"
            set_instance_assignment -to $hname -name VIRTUAL_PIN ON
        } else {
            post_message "Skipping VIRTUAL_PIN for $hname"
        }
    }
}
```

When the script runs, it generates a netlist file named **pipemult.vqm** and a **.qsf** file named **pipemult.qsf,** which contains the back-annotated assignments. You can then import the LogicLock region in another design. This example uses the top-level design in the **topmult** directory.

To import it four times in the top-level LogicLock tutorial design, follow these steps:

1. Create the top-level project.
2. Add assignments.
3. Elaborate the design.
4. Import the LogicLock constraints.
5. Compile the project.

Example 3–5 shows a script that demonstrates these steps. The example assumes that a top-level file named **topmult.bdf** exists, and that it instantiates the **pipemult** entity. Use the `quartus_cdb` executable to run this script example.

Example 3–5. LogicLock Import Script

```
load_package flow
load_package logiclock

project_new topmult -overwrite
set_global_assignment -name BDF_FILE topmult.bdf
set_global_assignment -name VQM_FILE pipemult.vqm
set_global_assignment -name FAMILY Cyclone
set_global_assignment -name DEVICE EP1C6F256C6
create_base_clock -fmax 200MHz -target clk clk_200

# The LogicLock region will be used four times
# in the top-level design. These assignments
# specify that the back-annotated assignments in
# the QSF will be applied to the four entities
# in the top-level design.
set_instance_assignment -name LL_IMPORT_FILE pipemult.qsf \
    -to pipemult:inst
set_instance_assignment -name LL_IMPORT_FILE pipemult.qsf \
    -to pipemult:inst1
set_instance_assignment -name LL_IMPORT_FILE pipemult.qsf \
    -to pipemult:inst2
set_instance_assignment -name LL_IMPORT_FILE pipemult.qsf \
    -to pipemult:inst3

execute_flow -analysis_and_elaboration
initialize_logiclock
logiclock_import
uninitialize_logiclock
execute_flow -compile
project_close
```

For additional information about the LogicLock design methodology, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Compiling Designs

You can run the Quartus II command-line executables from Tcl scripts. Use the included **flow** package to run various Quartus II compilation flows, or run each executable directly.

The flow Package

The **flow** package includes two commands for running Quartus II command-line executables, either individually or together in standard compilation sequence. The `execute_module` command allows you to run an individual Quartus II command-line executable. The `execute_flow` command allows you to run some or all of the modules in commonly-used combinations.

Altera recommends using the **flow** package instead of using system calls to run compiler executables.

© March 2009 Altera Corporation

Another way to run a Quartus II command-line executable from the Tcl environment is by using the `qexec` Tcl command, a Quartus II implementation of the Tcl `exec` command. For example, to run the Quartus II technology mapper on a given project, type:

```
qexec "quartus_map <project_name>" ↵
```

When you use the `qexec` command to compile a design, assignments made in the Tcl script (or from the Tcl shell) are not exported to the project database and settings file before compilation. Use the `export_assignments` command or compile the project with commands in the **flow** package to ensure assignments are exported to the project database and settings file.

You should use the `qexec` command to make system calls.

You can also run executables directly in a Tcl shell, using the same syntax as at the system command prompt. For example, to run the Quartus II technology mapper on a given project, type the following at the Tcl shell prompt:

```
quartus_map <project_name> ↵
```

Compile All Revisions

You can use a simple Tcl script to compile all revisions in your project. Save the script shown in Example 3–6 in a file called **compile_revisions.tcl** and type the following to run it:

```
quartus_sh -t compile_revisions.tcl <project name> ↵
```

Example 3–6. Compile All Revisions

```
load_package flow
project_open [lindex $quartus(args) 0]
set original_revision [get_current_revision]
foreach revision [get_project_revisions] {
    set_current_revision $revision
    execute flow -compile
}
set_current_revision $original_revision
project_close
```

Reporting

It is sometimes necessary to extract information from the report to evaluate the results when compilation is complete. For example, you might have to know how many device resources the design uses, or whether it meets your performance requirements. The Quartus II Tcl API provides easy access to report data so you do not have to write scripts to parse the text report files.

Use the commands that access report data one row at a time, or one cell at a time. If you know the exact cell or cells you want to access, use the `get_report_panel_data` command and specify the row and column names (or x and y coordinates) and the name of the appropriate report panel. You can often search for data in a report panel. To do this, use a loop that reads the report one row at a time with the `get_report_panel_row` command.

Column headings in report panels are in row 0. If you use a loop that reads the report one row at a time, you can start with row 1 to skip the row with column headings. The `get_number_of_rows` command returns the number of rows in the report panel, including the column heading row. Because the number of rows includes the column heading row, your loop should continue as long as the loop index is less than the number of rows, as illustrated in Example 3–8.

Report panels are hierarchically arranged and each level of hierarchy is denoted by the string "||" in the panel name. For example, the name of the Fitter Settings report panel is **Fitter||Fitter Settings** because it is in the Fitter folder. Panels at the highest hierarchy level do not use the "||" string. For example, the Flow Settings report panel is named **Flow Settings**.

The code in Example 3–7 prints a list of all report panel names in your project. You can run this code with any executable that includes support for the report package.

Example 3–7. Print All Report Panel Names

```
load_package report
project_open myproject
load_report
set panel_names [get_report_panel_names]
foreach panel_name $panel_names {
post_message "$panel_name"
}
```

Example 3–8 prints the number of failing paths in each clock domain in your design. It uses a loop to access each row of the Timing Analyzer Summary report panel. Clock domains are listed in the column named **Type** with the format `Clock Setup:'`*<clock name>*`'`. Other summary information is listed in the **Type** column as well. If the **Type** column matches the pattern "`Clock Setup*`", the script prints the number of failing paths listed in the column named **Failed Paths**. You can run this script example with any executable that supports the report package.

Example 3–8. Print Number of Failing Paths per Clock

```
load_package report
project_open my-project
load_report
set report_panel_name "Timing Analyzer||Timing Analyzer Summary"
set num_rows [get_number_of_rows -name $report_panel_name]

# Get the column indices for the Type and Failed Paths columns
set type_column [get_report_panel_column_index -name \
    $report_panel_name "Type"]
set failed_paths_column [get_report_panel_column_index -name \
    $report_panel_name "Failed Paths"]

# Go through each line in the report panel
for {set i 1} {$i < $num_rows} {incr i} {

    # Get the row of data, then the type of summary
    # information in the row, and the number of failed paths
    set report_row [get_report_panel_row -name \
    $report_panel_name -row $i]
    set row_type [lindex $report_row $type_column]
    set failed_paths [lindex $report_row $failed_paths_column]
    if { [string match "Clock Setup*" $row_type] } {
        puts "$row_type has $failed_paths failing paths"
    }
}
unload_report
```

Creating .csv Files for Excel

The Microsoft Excel software is sometimes used to view or manipulate timing analysis results. You can create a **.csv** file to import into Excel with data from any Quartus II report. Example 3–9 shows a simple way to create a **.csv** file with data from a timing analysis panel in the report. You could modify the script to use command-line arguments to pass in the name of the project, report panel, and output file to use. You can run this script example with any executable that supports the report package.

Example 3–9. Create .csv Files from Reports

```
load_package report
project_open my-project

load_report

# This is the name of the report panel to save as a CSV file
set panel_name "Timing Analyzer||Clock Setup: 'clk'"
set csv_file "output.csv"

set fh [open $csv_file w]
set num_rows [get_number_of_rows -name $panel_name]

# Go through all the rows in the report file, including the
# row with headings, and write out the comma-separated data
for { set i 0 } { $i < $num_rows } { incr i } {
    set row_data [get_report_panel_row -name $panel_name \
        -row $i]
    puts $fh [join $row_data ","]
}

close $fh
unload_report
```

Timing Analysis

The Quartus II software includes comprehensive Tcl APIs for both the Classic and TimeQuest Timing Analyzers. This section includes simple and advanced script examples for the Classic Timing Analyzer and introductory scripting information about the TimeQuest Tcl API.

Classic Timing Analysis

The following example script uses the `quartus_tan` executable to perform a timing analysis on the **fir_filter** tutorial design.

The **fir_filter** design is a two-clock design that requires a base clock and a relative clock relationship for timing analysis. This script first does an analysis of the two-clock relationship and checks for t_{SU} slack between `clk` and `clkx2`. The first pass of timing analysis discovers a negative slack for one of the clocks. The second part of the script adds a multicycle assignment from `clk` to `clkx2` and re-analyzes the design as a multi-clock, multicycle design.

The script does not recompile the design with the new timing assignments, and the timing-driven compilation is not used in the synthesis and placement of this design. New timing assignments are added only for the timing analyzer to analyze the design with the `create_timing_netlist` and `report_timing` Tcl commands.

You must compile the project before running the script example shown in Example 3–10.

Example 3–10. Classic Timing Analysis

```
# This Tcl file is to be used with quartus_tan.exe
# This Tcl file will do the Quartus II tutorial fir_filter design
# timing analysis portion by making a global timing assignment and
# creating multi-clock assignments and run timing analysis
# for a multi-clock, multi-cycle design
# set the project_name to fir_filter
# set the revision_name to filtref
set project_name fir_filter
set revision_name filtref
# open the project
# project_name is the project name
project_open -revision $revision_name $project_name;
# Doing TAN tutorial steps this section re-runs the timing
# analysis module with multi-clock and multi-cycle settings
#------ Make timing assignments ------#
#Specifying a global FMAX requirement (tan tutorial)
set_global_assignment -name FMAX_REQUIREMENT 45.0MHz
set_global_assignment -name CUT_OFF_IO_PIN_FEEDBACK ON
# create a base reference clock "clocka" and specifies the
# following:
#  BASED_ON_CLOCK_SETTINGS = clocka;
#  INCLUDE_EXTERNAL_PIN_DELAYS_IN_FMAX_CALCULATIONS = OFF;
#  FMAX_REQUIREMENT = 50MHZ;
#  DUTY_CYCLE = 50;
# Assigns the reference clocka to the pin "clk"
create_base_clock -fmax 50MHZ -duty_cycle 50 clocka -target clk
# creates a relative clock "clockb" based on reference clock
# "clocka" with the following settings:
#  BASED_ON_CLOCK_SETTINGS = clocka;
#  MULTIPLY_BASE_CLOCK_PERIOD_BY = 1;
#  DIVIDE_BASE_CLOCK_PERIOD_BY = 2;clock period is half the base clk
#  DUTY_CYCLE = 50;
#  OFFSET_FROM_BASE_CLOCK = 500ps;The offset is .5 ns (or 500 ps)
#  INVERT_BASE_CLOCK = OFF;
# Assigns the reference clock to pin "clkx2"
create_relative_clock -base_clock clocka -duty_cycle 50\
-divide 2 -offset 500ps -target clkx2 clockb
# create new timing netlist based on new timing settings
create_timing_netlist
# does an analysis for clkx2
# Limits path listing to 1 path
# Does clock setup analysis for clkx2
report_timing -npaths 1 -clock_setup -file setup_multiclock.tao
# The output file will show a negative slack for clkx2 when only
# specifying a multi-clock design. The negative slack was created
# by the 500 ps offset from the base clock
# removes old timing netlist to allow for creation of a new timing
# netlist for analysis by report_timing
delete_timing_netlist
# adding a multi-cycle setting corrects the negative slack by adding a
# multicycle assignment to clkx2 to allow for more set-up time
set_multicycle_assignment 2 -from clk -to clkx2
# create a new timing netlist based on additional timing
# assignments create_timing_netlist
# outputs the result to a file for clkx2 only
report_timing -npaths 1 -clock_setup -clock_filter clkx2 \
-file clkx2_setup_multicycle.tao
# The new output file will show a positive slack for the clkx2
project_close
```

Advanced Classic Timing Analysis

There might be times when the commands available for timing analysis reporting do not provide access to specific data you require. The **advanced_timing** package provides commands to access the data structures representing the timing netlist for your design. These commands provide low-level details about timing delays, node fan-in and fan-out, and timing data. Writing procedures to traverse the timing netlist and extract information gives you the most control to get exactly the data you require.

The timing netlist is represented with a graph, which is a collection of nodes and edges. Nodes represent elements in your design such as registers, combinational nodes, pins, and clocks. Edges connect the nodes and represent the connections between the logic in your design. Edges and nodes have unique positive integer IDs that identify them in the timing netlist. All the commands for getting information about the timing netlist use node and edge IDs instead of text-based names.

As an example of how to use the **advanced_timing** package, Example 3–11 shows one way to show the register-to-pin delays from all registers that drive the pins of an output bus. Specify the name of an output bus (for example, `address`), and the script prints out the names of all registers driving the pins of the bus and the delays from registers to pins. Use the `quartus_tan` executable to run this example.

Example 3–11. Report Register-to-Pin Delays

```
load_package advanced_timing
package require cmdline

# This procedure returns a list of IDs for pins with names
# that match the bus name passed in
proc find { bus_name } {

    set to_return [list]

    foreach_in_collection node_id [get_timing_nodes -type pin] {
        set node_name [get_timing_node_info -info name $node_id]
        if { [string match $bus_name* $node_name] } {
            lappend to_return $node_id
        }
    }
    return $to_return
}

# Required arguments for the script are the name of the project and
# revision, as well as the name of the bus to analyze
set options {\
    { "project.arg" "" "Project name" } \
    { "revision.arg" "" "Revision name" } \
    { "bus_name.arg" "" "Name of the bus to get timing data for" } \
}
array set opts [::cmdline::getoptions quartus(args) $options]

project_open $opts(project) -revision $opts(revision)

# The timing netlist must be created before accessing it.
create_timing_netlist

# This creates a data structure with additional timing data
create_p2p_delays

# Walk through each pin in the specified bus
foreach pin_id [find $opts(bus_name)] {
    set pin_name [get_timing_node_info -info name $pin_id]
    puts "$pin_name source registers and delays"
    # The get_delays_from_keepers command returns a list of all the
    # non-combinational nodes in the design that fan in to the
    # specified timing node, with the associated delays.
    foreach data [get_delays_from_keepers $pin_id] {
        set source_node [lindex $data 0]
        set max_delay [lindex $data 1]
        set source_node_name \
[get_timing_node_info -info name $source_node]
        puts " $source_node_name $max_delay"
    }
}
project_close
```

Type the command shown in Example 3–12 at a system command prompt to run this script.

Example 3–12.

```
quartus_tan -t script.tcl -project fir_filter -revision filtref -bus_name yn_out ↵
```

TimeQuest Timing Analysis

The Quartus II TimeQuest Timing Analyzer includes support for SDC commands in the **::quartus::sdc** package.

Refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* for detailed information about how to perform timing analysis with the Quartus II TimeQuest Timing Analyzer.

TimeQuest Scripting

In versions of the Quartus II software before 6.0, the **::quartus::project** Tcl package contained the following SDC-like commands for making timing assignments:

- `create_base_clock`
- `create_relative_clock`
- `get_clocks`
- `set_clock_latency`
- `set_clock_uncertainty`
- `set_input_delay`
- `set_multicycle_assignment`
- `set_output_delay`
- `set_timing_cut_assignment`

These commands are not SDC-compliant. Beginning with version 6.0, these commands are in a new package named **::quartus::timing_assignment**. To ensure backwards compatibility with existing Tcl scripts, the **::quartus::timing_assignment** package is loaded by default in the following executables:

- `quartus`
- `quartus_sh`
- `quartus_cdb`
- `quartus_sim`
- `quartus_stp`
- `quartus_tan`

The **::quartus::timing_assignment** package is not loaded by default in the `quartus_sta` executable. The **::quartus::sdc** Tcl package includes SDC-compliant versions of the commands listed above. The package is available only in the `quartus_sta` executable and it is loaded by default.

Automating Script Execution

You can configure scripts to run automatically at various points during compilation (Beginning with the Quartus II software version 4.0). Use this capability to automatically run scripts that perform custom reporting, make specific assignments, and perform many other tasks.

© March 2009 Altera Corporation

The following three global assignments control when a script is run automatically:

- `PRE_FLOW_SCRIPT_FILE` —before a flow starts
- `POST_MODULE_SCRIPT_FILE` —after a module finishes
- `POST_FLOW_SCRIPT_FILE` —after a flow finishes

The `POST_FLOW_SCRIPT_FILE` and `POST_MODULE_SCRIPT_FILE` assignments are supported beginning in version 4.0, and the `PRE_FLOW_SCRIPT_FILE` assignment is supported beginning in version 4.1.

A module is a Quartus II executable that performs one step in a flow. For example, two modules are Analysis and Synthesis (`quartus_map`), and timing analysis (`quartus_tan`).

A flow is a series of modules that the Quartus II software runs with predefined options. For example, compiling a design is a flow that typically consists of the following steps (performed by the indicated module):

1. Analysis and synthesis (`quartus_map`)
2. Fitter (`quartus_fit`)
3. Assembler (`quartus_asm`)
4. Timing Analyzer (`quartus_tan` or `quartus_sta`)

Other flows are described in the help for the `execute_flow` Tcl command. In addition, many commands in the Processing menu of the Quartus II GUI correspond to this design flow.

Making the Assignment

To make an assignment automatically run a script, add an assignment with the following form to your project's **.qsf** file:

```
set_global_assignment -name <assignment name> <executable>:<script name>
```

The assignment name is one of the following:

- `PRE_FLOW_SCRIPT_FILE`
- `POST_MODULE_SCRIPT_FILE`
- `POST_FLOW_SCRIPT_FILE`

The executable is the name of a Quartus II command-line executable that includes a Tcl interpreter.

- `quartus_cdb`
- `quartus_sh`
- `quartus_sim`
- `quartus_sta`
- `quartus_stp`
- `quartus_tan`

The script name is the name of your Tcl script.

Script Execution

The Quartus II software runs the scripts as shown in Example 3–13.

Example 3–13.

```
<executable> -t <script name> <flow or module name> <project name> <revision name>
```

The first argument passed in the argv variable (or quartus(args) variable) is the name of the flow or module being executed, depending on the assignment you use. The second argument is the name of the project and the third argument is the name of the revision.

When you use the POST_MODULE_SCRIPT_FILE assignment, the specified script is automatically run after every executable in a flow. You can use a string comparison with the module name (the first argument passed in to the script) to isolate script processing to certain modules.

Execution Example

Example 3–14 illustrates how automatic script execution works in a complete flow, assuming you have a project called **top** with a current revision called **rev_1**, and you have the following assignments in the **.qsf** file for your project.

Example 3–14.

```
set_global_assignment -name PRE_FLOW_SCRIPT_FILE quartus_sh:first.tcl
set_global_assignment -name POST_MODULE_SCRIPT_FILE quartus_sh:next.tcl
set_global_assignment -name POST_FLOW_SCRIPT_FILE quartus_sh:last.tcl
```

When you compile your project, the PRE_FLOW_SCRIPT_FILE assignment causes the following command to be run before compilation begins:

```
quartus_sh -t first.tcl compile top rev_1
```

Next, the Quartus II software starts compilation with analysis and synthesis, performed by the quartus_map executable. After the analysis and synthesis finishes, the POST_MODULE_SCRIPT_FILE assignment causes the following command to run:

```
quartus_sh -t next.tcl quartus_map top rev_1
```

Then, the Quartus II software continues compilation with the Fitter, performed by the quartus_fit executable. After the Fitter finishes, the POST_MODULE_SCRIPT_FILE assignment runs the following command:

```
quartus_sh -t next.tcl quartus_fit top rev_1
```

Corresponding commands are run after the other stages of the compilation. When the compilation is over, the POST_FLOW_SCRIPT_FILE assignment runs the following command:

```
quartus_sh -t last.tcl compile top rev_1
```


© March 2009 Altera Corporation

Controlling Processing

The `POST_MODULE_SCRIPT_FILE` assignment causes a script to run after every module. Because the same script is run after every module, you might have to include some conditional statements that restrict processing in your script to certain modules.

For example, if you want a script to run only after timing analysis, you should include a conditional test like the one shown in Example 3–15. It checks the flow or module name passed as the first argument to the script and executes code when the module is `quartus_tan`.

Example 3–15. Restrict Processing to a Single Module

```
set module [lindex $quartus(args) 0]

if [string match "quartus_tan" $module] {

    # Include commands here that are run
    # after timing analysis
    # Use the post-message command to display
    # messages
    post_message "Running after timing analysis"
}
```

Displaying Messages

Because of the way the Quartus II software runs the scripts automatically, you must use the `post_message` command to display messages, instead of the `puts` command. This requirement applies only to scripts that are run by the three assignments listed in "Automating Script Execution" on page 3–22.

Refer to "Using the post_message Command" on page 3–27 for more information about this command.

Other Scripting Features

The Quartus II Tcl API includes other general-purpose commands and features described in this section.

Natural Bus Naming

Beginning with version 4.2, the Quartus II software supports natural bus naming. Natural bus naming means that square brackets used to specify bus indexes in HDL do not have to be escaped to prevent Tcl from interpreting them as commands. For example, one signal in a bus named address can be identified as `address[0]` instead of `address\[0\]`. You can take advantage of natural bus naming when making assignments, as in Example 3–16.

Example 3–16. Natural Bus Naming

```
set_location_assignment -to address[10] Pin_M20
```

The Quartus II software defaults to natural bus naming. You can turn off natural bus naming with the `disable_natural_bus_naming` command. For more information about natural bus naming, type the following at a Quartus II Tcl prompt:

```
enable_natural_bus_naming -h ↵
```

Short Option Names

Beginning with version 6.0 of the Quartus II software, you can use short versions of command options, as long as they distinguish between the command's options. For example, the `project_open` command supports two options: `-current_revision` and `-revision`. You can use any of the following shortened versions of the `-revision` option: `-r`, `-re`, `-rev`, `-revi`, `-revis`, and `-revisio`. You can use an option as short as `-r` because no other option starts with the same letters as revision. However, the `report_timing` command includes the options `-recovery` and `-removal`. You cannot use `-r` or `-re` to shorten either of those options, because they do not uniquely distinguish between the two options. You could use `-rec` or `-rem`.

Using Collection Commands

Some Quartus II Tcl functions return very large sets of data that would be inefficient as Tcl lists. These data structures are referred to as collections and the Quartus II Tcl API uses a collection ID to access the collection. There are two Quartus II Tcl commands for working with collections, `foreach_in_collection` and `get_collection_size`. Use the `set` command to assign a collection ID to a variable.

For information about which Quartus II Tcl commands return collection IDs, see the Quartus II Help and search for the `foreach_in_collection` command.

The foreach_in_collection Command

The `foreach_in_collection` command is similar to the `foreach` Tcl command. Use it to iterate through all elements in a collection. Example 3–17 prints all instance assignments in an open project.

Example 3–17. Using Collection Commands

```
set all_instance_assignments [get_all_instance_assignments -name *]
foreach_in_collection asgn $all_instance_assignments {
    # Information about each assignment is
    # returned in a list. For information
    # about the list elements, refer to Help
    # for the get-all-instance-assignments command.
    set to [lindex $asgn 2]
    set name [lindex $asgn 3]
    set value [lindex $asgn 4]
    puts "Assignment to $to: $name = $value"
}
```

The get_collection_size Command

Use the `get_collection_size` command to get the number of elements in a collection. Example 3–18 prints the number of global assignments in an open project.

Example 3–18. get_collection_size Command

```
set all_global_assignments [get_all_global_assignments -name *]
set num_global_assignments [get_collection_size $all_global_assignments]
puts "There are $num_global_assignments global assignments in your project"
```

Using the post_message Command

To print messages that are formatted like Quartus II software messages, use the `post_message` command. Messages printed by the `post_message` command appear in the **System** tab of the Messages window in the Quartus II GUI, and are written to standard at when scripts are run. Arguments for the `post_message` command include an optional message type and a required message string.

The message type can be one of the following:

- `info` (default)
- `extra_info`
- `warning`
- `critical_warning`
- `error`

If you do not specify a type, Quartus II software defaults to `info`.

When you are using the Quartus II software in Windows, you can color code messages displayed at the system command prompt with the `post_message` command. Add the following line to your **quartus2.ini** file:

```
DISPLAY_COMMAND_LINE_MESSAGES_IN_COLOR = on
```

Example 3–19 shows how to use the `post_message` command.

Example 3–19. post_message command

```
post_message -type warning "Design has gated clocks"
```

Accessing Command-Line Arguments

Many Tcl scripts are designed to accept command-line arguments, such as the name of a project or revision. The global variable `quartus(args)` is a list of the arguments typed on the command-line following the name of the Tcl script. Example 3–20 shows code that prints all of the arguments in the `quartus(args)` variable.

Example 3–20. Simple Command-Line Argument Access

```
set i 0
foreach arg $quartus(args) {
   puts "The value at index $i is $arg"
   incr i
}
```

If you copy the script in the previous example to a file named **print_args.tcl**, it displays the following output when you type the command shown in Example 3–21 at a command prompt.

Example 3–21. Passing Command-Line Arguments to Scripts

```
quartus_sh -t print_args.tcl my_project 100MHz ↵
The value at index 0 is <my_project>
The value at index 1 is 100MHz
```

Beginning with version 4.1, the Quartus II software supports the Tcl variables `argv`, `argc`, and `argv0` for command-line argument access. Table 3–6 shows equivalent information for earlier versions of the software.

Table 3–6. Quartus II Support for Tcl Variables

Beginning with Version 4.1	Equivalent Support in Previous Software Versions
`argc`	`llength $quartus(args)`
`argv`	`quartus(args)`
`argv0`	`info nameofexecutable`

Using the cmdline Package

You can use the **cmdline** package included with the Quartus II software for more robust and self-documenting command-line argument passing. The **cmdline** package supports command-line arguments with the form `-`*<option>* *<value>*.

Example 3–22 uses the **cmdline** package.

Example 3–22. cmdline Package

```
package require cmdline
variable ::argv0 $::quartus(args)
set options {
   { "project.arg" "" "Project name" }
   { "frequency.arg" "" "Frequency" }
}
set usage "You need to specify options and values"

array set optshash [::cmdline::getoptions ::argv $options $usage]
puts "The project name is $optshash(project)"
puts "The frequency is $optshash(frequency)"
```

If you save those commands in a Tcl script called **print_cmd_args.tcl** you see the following output when you type the command shown in Example 3–32 at a command prompt.

Example 3–23. Passing Command-Line Arguments for Scripts

```
quartus_sh -t print_cmd_args.tcl -project my_project -frequency 100MHz ↵
The project name is my_project
The frequency is 100MHz
```

Virtually all Quartus II Tcl scripts must open a project. Example 3–24 opens a project, and you can optionally specify a revision name. The example checks whether the specified project exists. If it does, the example opens the current revision, or the revision you specify.

Example 3–24. Full-Featured Method to Open Projects

```
package require cmdline
variable ::argv0 $::quartus(args)
set options { \
{ "project.arg" "" "Project Name" } \
{ "revision.arg" "" "Revision Name" } \
}
array set optshash [::cmdline::getoptions ::argv0 $options]

# Ensure the project exists before trying to open it
if {[project_exists $optshash(project)]} {

    if {[string equal "" $optshash(revision)]} {

        # There is no revision name specified, so default
        # to the current revision
        project_open $optshash(project) -current_revision
    } else {

        # There is a revision name specified, so open the
        # project with that revision
        project_open $optshash(project) -revision \
            $optshash(revision)
    }
} else {
    puts "Project $optshash(project) does not exist"
    exit 1
}
# The rest of your script goes here
```

If you do not require this flexibility or error checking, you can use just the `project_open` command, as shown in Example 3–25.

Example 3–25. Simple Method to Open Projects

```
set proj_name [lindex $argv 0]
project_open $proj_name
```

For more information about the **cmdline** package, refer to the documentation for the package at <*Quartus II installation directory*> **/common/tcl/packages.**

Using the Quartus II Tcl Shell in Interactive Mode

This section presents an example of using the `quartus_sh` interactive shell to make some project assignments and compile the finite impulse response (FIR) filter tutorial project. This example assumes that you already have the FIR filter tutorial design files in a project directory.

To begin, run the interactive Tcl shell. The command and initial output are shown in Example 3–26.

Example 3–26. Interactive Tcl Shell

```
tcl> quartus_sh -s
tcl> Info:
*******************************************************************
Info: Running Quartus II Shell
Info: Version 8.1 Full Version
Info: Copyright (C) 1991-2007 Altera Corporation. All rights reserved.
Info: Your use of Altera Corporation's design tools, logic functions
Info: and other software and tools, and its AMPP partner logic
Info: functions, and any output files any of the foregoing
Info: (including device programming or simulation files), and any
Info: associated documentation or information are expressly subject
Info: to the terms and conditions of the Altera Program License
Info: Subscription Agreement, Altera MegaCore Function License
Info: Agreement, or other applicable license agreement, including,
Info: without limitation, that your use is for the sole purpose of
Info: programming logic devices manufactured by Altera and sold by
Info: Altera or its authorized distributors. Please refer to the
Info: applicable agreement for further details.
Info: Processing started: <date>
Info:
*******************************************************************
Info: The Quartus II Shell supports all TCL commands in addition
Info: to Quartus II Tcl commands. All unrecognized commands are
Info: assumed to be external and are run using Tcl's "exec"
Info: command.
Info: - Type "exit" to exit.
Info: - Type "help" to view a list of Quartus II Tcl packages.
Info: - Type "help <package name>" to view a list of Tcl commands
Info:   available for the specified Quartus II Tcl package.
Info: - Type "help -tcl" to get an overview on Quartus II Tcl usages.
Info:
*******************************************************************

tcl>
```

Create a new project called **fir_filter**, with a revision called **filtref** by typing the following command at a Tcl prompt:

```
project_new -revision filtref fir_filter ↵
```

If the project file and project name are the same, the Quartus II software gives the revision the same name as the project.

Because the revision named **filtref** matches the top-level file, all design files are automatically picked up from the hierarchy tree.

Next, set a global assignment for the device with the following command:

```
set_global_assignment -name family Cyclone ↵
```

To learn more about assignment names that you can use with the `-name` option, refer to the Quartus II Help.

For assignment values that contain spaces, the value should be enclosed in quotation marks.

To quickly compile a design, use the `::quartus::flow` package, which properly exports the new project assignments and compiles the design using the proper sequence of the command-line executables. First, load the package:

```
load_package flow ↵
```

It returns the following:

```
1.0
```

For additional help on the `::quartus::flow` package, view the command-line help at the Tcl prompt by typing:

```
help -pkg ::quartus::flow ↵
```

Example 3–27 shows an alternative command and the resulting output.

Example 3–27. Help Output

```
tcl> help -pkg flow
----------------------------------------------------------------
------------------------
Tcl Package and Version:
------------------------
::quartus::flow 1.0
------------
Description:
------------
This package contains the set of Tcl functions
for running flows or command-line executables.
-------------
Tcl Commands:
-------------
execute_flow
execute_module
----------------------------------------------------------------
```

This help display gives information about the flow package and the commands that are available with the package.To learn about the options available for the `execute_flow` Tcl command, type the following command at a Tcl prompt:

```
execute_flow -h ↵
```

To view additional information and example usage type the following command at a Tcl prompt:

```
execute_flow -long_help ↵
```

or

```
help -cmd execute_flow ↵
```

To perform a full compilation of the FIR filter design, use the `execute_flow` command with the `-compile` option, as shown in Example 3–28.

Example 3–28.

```
tcl> execute_flow -compile ↵
Info:*******************************************************
Info: Running Quartus II Analysis & Synthesis
Info: Version 6.0 SJ Full Version
Info: Processing started: <date><time>
Info: Command: quartus_map --import_settings_files=on --export_settings_files=of
fir_filter -c filtref
.
.
.
Info: Writing report file filtref.tan.rpt
tcl>
```

This script compiles the FIR filter tutorial project, exporting the project assignments and running `quartus_map`, `quartus_fit`, `quartus_asm`, and `quartus_tan`. This sequence of events is the same as selecting **Start Compilation** from the Processing menu in the Quartus II GUI.

When you are finished with a project, close it using the `project_close` command as shown in Example 3–29.

Example 3–29.

```
project_close ↵
```

Exit the interactive Tcl shell, type `exit` ↵ at a Tcl prompt.

Quartus II Legacy Tcl Support

Beginning with the Quartus II software version 3.0, command-line executables do not support the Tcl commands used in previous versions of the Quartus II software. These commands are supported in the GUI with the Quartus II Tcl console or by using the `quartus_cmd` executable at the system command prompt. If you source Tcl scripts developed for an earlier version of the Quartus II software using either of these methods, the project assignments are ported to the project database and settings file. You can then use the command-line executables to process the resulting project. This might be necessary if you create a **.tcl** file using EDA tools that do not generate Tcl scripts for the most recent version of the Quartus II software.

You should create all new projects and Tcl scripts with the latest version of the Quartus II Tcl API.

Using the tclsh Shell

On the UNIX and Linux operating systems, the tclsh shell included with the Quartus II software is initialized with a minimal PATH environment variable. As a result, system commands may not be available within the tclsh shell because certain directories are not in the PATH environment variable. To include other directories in the path searched by the tclsh shell, set the QUARTUS_INIT_PATH environment variable before running the tclsh shell. Directories in the QUARTUS_INIT_PATH environment variable are searched by the tclsh shell when you execute a system command.

Tcl Scripting Basics

The core Tcl commands support variables, control structures, and procedures. Additionally, there are commands for accessing the file system and network sockets, and running other programs. You can create platform-independent graphical interfaces with the Tk widget set.

Tcl commands are executed immediately as they are typed in an interactive Tcl shell. You can also create scripts (including this chapter's examples) as files and run them with a Tcl interpreter. A Tcl script does not require any special headers.

To start an interactive Tcl interpreter, type `quartus_sh -s` ↵ at a command prompt. The commands you type are executed immediately at the interpreter prompt. If you save a series of Tcl commands in a file, you can run it with a Tcl interpreter. To run a script named **myscript.tcl**, type `quartus_sh -t myscript.tcl` ↵ at a command prompt.

Hello World Example

The following shows the basic "Hello world" example in Tcl:

```
puts "Hello world"
```

Use double quotation marks to group the words `hello` and `world` as one argument. Double quotation marks allow substitutions to occur in the group. Substitutions can be simple variable substitutions, or the result of running a nested command, described in "Substitutions" on page 3–33. Use curly braces { } for grouping when you want to prevent substitutions.

Variables

Use the `set` command to assign a value to a variable. You do not have to declare a variable before using it. Tcl variable names are case-sensitive. Example 3–30 assigns the value `1` to the variable named `a`.

Example 3–30. Assigning Variables

```
set a 1
```

To access the contents of a variable, use a dollar sign before the variable name. Example 3–31 prints `"Hello world"` in a different way.

Example 3–31. Accessing Variables

```
set a Hello
set b world
puts "$a $b"
```

Substitutions

Tcl performs three types of substitution:

- Variable value substitution
- Nested command substitution
- Backslash substitution

Variable Value Substitution

Variable value substitution, as shown in Example 3–31, refers to accessing the value stored in a variable by using a dollar sign ("$") before the variable name.

Nested Command Substitution

Nested command substitution refers to how the Tcl interpreter evaluates Tcl code in square brackets. The Tcl interpreter evaluates nested commands, starting with the innermost nested command, and commands nested at the same level from left to right. Each nested command result is substituted in the outer command. Example 3–32 sets a to the length of the string foo.

Example 3–32. Command Substitution

```
set a [string length foo]
```

Backlash Substitution

Backslash substitution allows you to quote reserved characters in Tcl, such as dollar signs ("$") and braces ("[ ]"). You can also specify other special ASCII characters like tabs and new lines with backslash substitutions. The backslash character is the Tcl line continuation character, used when a Tcl command wraps to more than one line. Example 3–33 shows how to use the backslash character for line continuation.

Example 3–33. Backslash Substitution

```
set this_is_a_long_variable_name [string length "Hello \
    world."]
```

Arithmetic

Use the expr command to perform arithmetic calculations. Using curly braces ("{ }") to group the arguments of this command makes arithmetic calculations more efficient and preserves numeric precision. Example 3–34 sets b to the sum of the value in the variable a and the square root of 2.

Example 3–34. Arithmetic with the expr Command

```
set a 5
set b [expr { $a + sqrt(2) }]
```

Tcl also supports boolean operators such as && (AND), || (OR), ! (NOT), and comparison operators such as < (less than), > (greater than), and == (equal to).

Lists

A Tcl list is a series of values. Supported list operations include creating lists, appending lists, extracting list elements, computing the length of a list, sorting a list, and more. Example 3–35 sets a to a list with three numbers in it.

Example 3–35. Creating Simple Lists

```
set a { 1 2 3 }
```

You can use the `lindex` command to extract information at a specific index in a list. Indexes are zero-based. You can use the index `end` to specify the last element in the list, or the index `end-<n>` to count from the end of the list. Example 3–36 prints the second element (at index 1) in the list stored in `a`.

Example 3–36. Accessing List Elements

```
puts [lindex $a 1]
```

The `llength` command returns the length of a list. Example 3–37 prints the length of the list stored in `a`.

Example 3–37. List Length

```
puts [llength $a]
```

The `lappend` command appends elements to a list. If a list does not already exist, the list you specify is created. The list variable name is not specified with a dollar sign. Example 3–38 appends some elements to the list stored in `a`.

Example 3–38. Appending to a List

```
lappend a 4 5 6
```

Arrays

Arrays are similar to lists except that they use a string-based index. Tcl arrays are implemented as hash tables. You can create arrays by setting each element individually or by using the `array set` command. To set an element with an index of `Mon` to a value of `Monday` in an array called `days`, use the following command:

```
set days(Mon) Monday
```

The `array set` command requires a list of index/value pairs. This example sets the array called days:

```
array set days { Sun Sunday Mon Monday Tue Tuesday \
   Wed Wednesday Thu Thursday Fri Friday Sat Saturday }
```

Example 3–39 shows how to access the value for a particular index.

Example 3–39. Accessing Array Elements

```
set day_abbreviation Mon
puts $days($day_abbreviation)
```

Use the `array names` command to get a list of all the indexes in a particular array. The index values are not returned in any specified order. Example 3–40 shows one way to iterate over all the values in an array.

Example 3–40. Iterating Over Arrays

```
foreach day [array names days] {
   puts "The abbreviation $day corresponds to the day \
name $days($day)"
}
```

Arrays are a very flexible way of storing information in a Tcl script and are a good way to build complex data structures.

Control Structures

Tcl supports common control structures, including if-then-else conditions and `for`, `foreach`, and `while` loops. The position of the curly braces as shown in the following examples ensures the control structure commands are executed efficiently and correctly. Example 3–41 prints whether the value of variable `a` positive, negative, or zero.

Example 3–41. If-Then-Else Structure

```
if { $a > 0 } {
    puts "The value is positive"
} elseif { $a < 0 } {
    puts "The value is negative"
} else {
    puts "The value is zero"
}
```

Example 3–42 uses a `for` loop to print each element in a list.

Example 3–42. For Loop

```
set a { 1 2 3 }
for { set i 0 } { $i < [llength $a] } { incr i } {
    puts "The list element at index $i is [lindex $a $i]"
}
```

Example 3–43 uses a `foreach` loop to print each element in a list.

Example 3–43. foreach Loop

```
set a { 1 2 3 }
foreach element $a {
    puts "The list element is $element"
}
```

Example 3–44 uses a `while` loop to print each element in a list.

Example 3–44. while Loop

```
set a { 1 2 3 }
set i 0
while { $i < [llength $a] } {
    puts "The list element at index $i is [lindex $a $i]"
    incr i
}
```

You do not have to use the `expr` command in boolean expressions in control structure commands because they invoke the `expr` command automatically.

Procedures

Use the `proc` command to define a Tcl procedure (known as a subroutine or function in other scripting and programming languages). The scope of variables in a procedure is local to the procedure. If the procedure returns a value, use the `return` command to return the value from the procedure. Example 3–45 defines a procedure that multiplies two numbers and returns the result.

Example 3–45. Simple Procedure

```
proc multiply { x y } {
    set product [expr { $x * $y }]
    return $product
}
```

Example 3–46 shows how to use the `multiply` procedure in your code. You must define a procedure before your script calls it.

Example 3–46. Using a Procedure

```
proc multiply { x y } {
    set product [expr { $x * $y }]
    return $product
}
set a 1
set b 2
puts [multiply $a $b]
```

You should define procedures near the beginning of a script. If you want to access global variables in a procedure, use the `global` command in each procedure that uses a global variable. Example 3–47 defines a procedure that prints an element in a global list of numbers, then calls the procedure.

Example 3–47. Accessing Global Variables

```
proc print_global_list_element { i } {
    global my_data
    puts "The list element at index $i is [lindex $my_data $i]"
}
set my_data { 1 2 3}
print_global_list_element 0
```

File I/O

Tcl includes commands to read from and write to files. You must open a file before you can read from or write to it, and close it when the read and write operations are done. To open a file, use the `open` command; to close a file, use the `close` command. When you open a file, specify its name and the mode in which to open it. If you do not specify a mode, Tcl defaults to read mode. To write to a file, specify `w` for write mode as shown in Example 3–48.

Example 3–48. Open a File for Writing

```
set output [open myfile.txt w]
```

Tcl supports other modes, including appending to existing files and reading from and writing to the same file.

The `open` command returns a file handle to use for read or write access. You can use the `puts` command to write to a file by specifying a filehandle, as shown in Example 3–49.

Example 3–49. Write to a File

```
set output [open myfile.txt w]
puts $output "This text is written to the file."
close $output
```

You can read a file one line at a time with the `gets` command. Example 3–50 uses the `gets` command to read each line of the file and then prints it out with its line number.

Example 3–50. Read from a File

```
set input [open myfile.txt]
set line_num 1
while { [gets $input line] >= 0 } {
    # Process the line of text here
    puts "$line_num: $line"
    incr line_num
}
close $input
```

Syntax and Comments

Arguments to Tcl commands are separated by white space, and Tcl commands are terminated by a newline character or a semicolon. As shown in "Substitutions" on page 3–33, you must use backslashes when a Tcl command extends more than one line.

Tcl uses the hash or pound character (#) to begin comments. The # character must begin a comment. If you prefer to include comments on the same line as a command, be sure to terminate the command with a semicolon before the # character. Example 3–51 is a valid line of code that includes a `set` command and a comment.

Example 3–51. Comments

```
set a 1;# Initializes a
```

Without the semicolon, it would be an invalid command because the `set` command would not terminate until the new line after the comment.

The Tcl interpreter counts curly braces inside comments, which can lead to errors that are difficult to track down. Example 3–52 causes an error because of unbalanced curly braces.

Example 3–52. Unbalanced Braces in Comments

```
# if { $x > 0 } {
if { $y > 0 } {
    # code here
}
```

External References

For more information about using Tcl, refer to the following sources:

- *Practical Programming in Tcl and Tk*, Brent B. Welch
- *Tcl and the TK Toolkit*, John Ousterhout
- *Effective Tcl/TK Programming*, Michael McLennan and Mark Harrison
- Quartus II Tcl example scripts at www.altera.com/support/examples/tcl/tcl.html
- Tcl Developer Xchange at tcl.activestate.com

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Script-Based Design for HardCopy Devices* chapter of the *HardCopy Handbook*
- *Section I. Simulation* in volume 1 of the *Quartus II Handbook*

Document Revision History

Table 3–7 shows the revision history for this chapter.

Table 3–7. Document Revision History

Date / Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Removed the "EDA Tool Assignments" section ■ Added the section "Compile All Revisions" on page 3–15 ■ Added the section "Using the tclsh Shell" on page 3–32	Updated for the Quartus II software version 9.0.
November 2008 v8.1.0	Changed to 8½" × 11" page size. No change to content.	Updated for the Quartus II software version 8.1.
May 2008 v8.0.0	Updated references.	Updated for the Quartus II software version 8.0.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

4. Managing Quartus II Projects

QII52012-9.0.0

Introduction

FPGA designs once required just one or two engineers, but today's larger and more sophisticated FPGA designs are often developed by several engineers and are constantly changing throughout the project. To ensure efficient design coordination, designers must track their project changes.

To help designers manage their FPGA designs, the Quartus® II software provides the following capabilities:

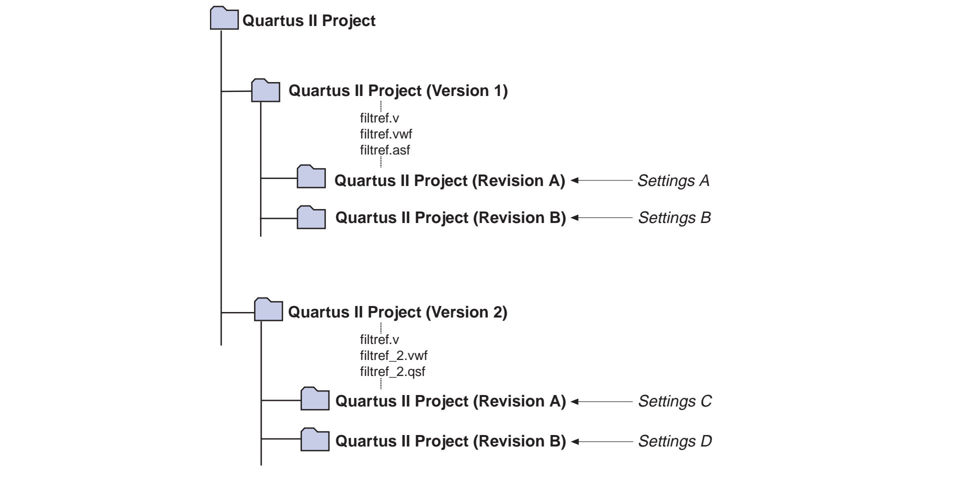
- Create revisions
- Copy and archive projects
- Make a version compatible database
- Control message suppression

In the Quartus II software, a revision is a set of assignments and settings. A project may have multiple revisions, with each revision having its own set of assignments and settings. You can create multiple revisions to change assignments and settings while preserving the previous results.

A version is a Quartus II project that includes one set of design files, and one or more revisions (assignments and settings for your project). Creating multiple versions with the Copy Project feature allows you to edit a copy of your design files while preserving the original functionality of your design in another copy.

The Quartus II software Version Compatible Database feature allows Quartus II databases to be compatible across different versions of the Quartus II software, which saves valuable design time by avoiding unnecessary compilations (Figure 4–1). This chapter also discusses how to migrate your projects from one computing platform to another, as well as message suppression.

Figure 4–1. Quartus II Version Compatible Database Structure



Creating a New Project

A Quartus II project contains all the design files, settings files, and other files necessary for the successful compilation of your design. These files include the following files that contain the project settings:

- Quartus II Project File (**.qpf**)—contains the name of your project and all revisions of your project, described in “Using Revisions with Your Design” on page 4–2.
- Quartus II Settings File (**.qsf**)—contains all assignments applied to your design, including assignments to help fit your design and meet performance requirements. For more information about the **.qsf** file, refer to “Quartus II Settings File” on page 4–21.
- Synopsys Design Constraints (**.sdc**)—contains timing assignments for the Quartus II TimeQuest Timing Analyzer.

For more information about SDC constraints, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

To start a new Quartus II project, use the **New Project Wizard**. On the File menu, click **New Project Wizard,** and add the following project information:

- Project name and directory
- Name of the top-level design entity
- Project files and user libraries
- Target device family and device
- EDA tool settings

For more information about user libraries, refer to “Specifying Libraries” on page 4–11 and “Specifying Libraries Using Scripts” on page 4–25.

Using Revisions with Your Design

The Revisions feature allows you to create a new set of assignments and settings for a set of design files without losing your previous assignments and settings. You can use the Revisions feature in the following ways:

- Create a unique revision which is not based on a previous revision. Creating a unique revision lets you optimize a design for different fundamental reasons such as to optimize by area in one revision, then optimize for f_{MAX} in another revision. When you create a unique revision (a revision that is not based on an existing revision), all default settings are turned on.
- Create a revision based on an existing revision, but try new settings and assignments in the new revision. A new revision already includes all the assignments and settings applied in the previous revision. If you are not satisfied with the results in the new revision, you can revert to the original revision. You can compare revisions manually or with the Compare feature.

Creating and Deleting Revisions

All Quartus II assignments and settings are stored in the **.qsf** file. Each time you create a new revision of a project, the Quartus II software creates a new **.qsf** file and adds the name of the new revision to the list of revisions in the **.qsf** file.

The name of a new **.qsf** file matches the revision name.

You can manage revisions with the **Revisions** dialog box, which allows you to set the current revision, and perform the following tasks:

- "Create a Revision"
- "Delete a Revision"
- "Compare Revisions"

Create a Revision

To create a revision, perform the following steps:

1. If you have not already done so, create a new project or open an existing project. On the File menu, click **New Project Wizard** or **Open Project.**
2. On the Project menu, click **Revisions**.
3. To base the new revision on an existing revision for the current design, select the existing revision in the **Revisions** list.
4. Click **Create**.
5. In the **Create Revision** dialog box, type the name of the new revision in the **Revisions name** box.
6. To base the new revision on an existing revision for the current design, if you did not select the correct revision in Step 3, select the revision in the **Based on revision** list. To create a unique revision that is not based on an existing revision of the current design, select the "blank entry" in the **Based on revision** list.
7. Optionally, edit the description of the revision in the **Description** box.
8. Turn off the **Copy database** option if you do not want the new revision to contain the database information from the existing revision. The **Copy database** option is on by default.

   Copying the database allows you to view the results from the previous compilation while you are making changes to the settings of the new revision.

9. If you do not want to use the new revision immediately, turn off the **Set as current revision** option. The **Set as current revision** option is turned on by default.
10. Click **OK**.

Delete a Revision

To delete a revision, perform the following steps:

1. If you have not already done so, open an existing project by clicking **Open Project** on the File menu and selecting **Quartus II Project File.**
2. On the Project menu, click **Revisions**.
3. In the **Revisions** list, select the revision you want to delete and click **Delete**.

You cannot delete the current revision when it is active; you must first open another revision. For example, if revision A is currently active, you must create (if the revision does not exist) and open revision B before you delete revision A.

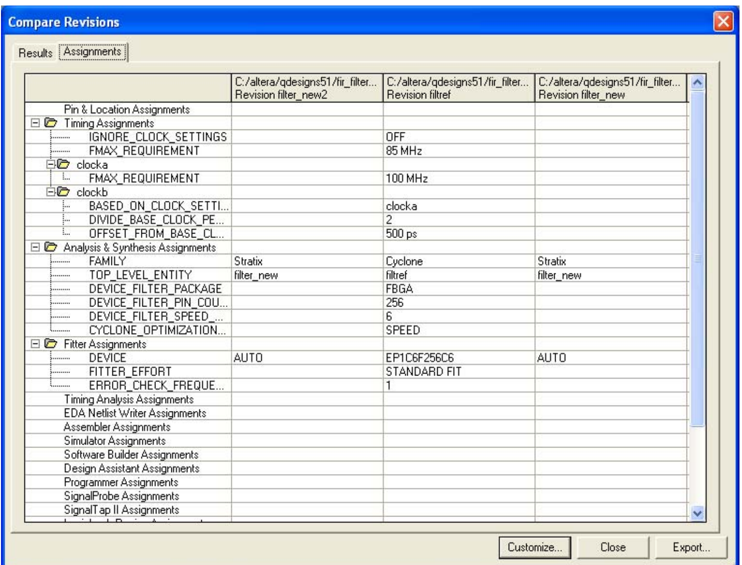
Compare Revisions

You can compare the results of multiple revisions side by side with the **Compare Revisions** dialog box.

1. On the Project menu, click **Revisions**.
2. In the **Revisions** dialog box, click **Compare** to compare all revisions in a single window.

The **Compare Revisions** dialog box (Figure 4–2) compares the results of each revision in three assignment categories: **Analysis & Synthesis**, **Fitter**, and **Timing Analyzer**.

Figure 4–2. Compare Revisions Dialog Box



© March 2009 Altera Corporation

In addition to viewing the results of each revision, you can show the assignments for each revision. Click the **Assignments** tab in the **Compare Revisions** dialog box to view all assignments applied to each revision (Figure 4–2). To export both **Results** and **Assignments** for your revisions, click on **Export.** When the dialog box appears, enter the name of the Comma-Separated Value (.**csv**) file in which the software will export the data when you click **OK**. View the results of each revision and their assignments to gain better understanding of how different optimization options affect your design.

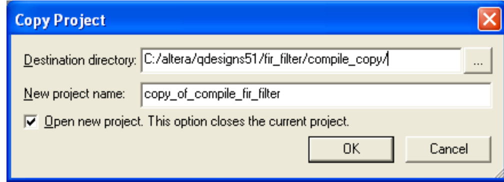
Creating New Copies of Your Design

Managing different copies of design files in a large project can become difficult. To assist you in this task, the Quartus II software provides utilities to copy and save different copies of your project. Creating a copy of your project using the Copy Project feature includes copying all your design files, your **.qsf** file, and all your associated revisions (all assignments and settings).

To create a new copy of your project with the Quartus II software, create a copy of your project and edit your design files (for example, if you have a design that is compatible with a 32-bit data bus and you require a new copy of the design to interface with a 64-bit data bus). To solve this problem, create a new copy of your project and edit the new version of the design files by performing the following steps:

1. On the Project menu, click **Copy Project**. The **Copy Project** dialog box appears (Figure 4–3).

Figure 4–3. Copy Project Dialog Box



2. Specify the path to your new project in the **Destination directory** box.
3. Type the new project name in the **New project name** box.
4. To open the new project immediately, turn on **Open new project. This option closes the current project** option.
5. Click **OK**.

When creating a new copy of your project with an EDIF or Verilog Quartus Mapping (.**vqm**) netlist file from a third-party EDA synthesis tool, create a copy of your project and then replace the previous netlist file with the newly generated netlist file. On the Project menu, click **Copy Project** to create a copy of your design. On the Project menu, click the **Add/Remove Files** command to add and remove design files, if necessary.

Archiving Projects with the Quartus II Archive Project Feature

A single project can contain hundreds of files in many directories, which can make transferring a project between engineers difficult. You can use the Quartus II Archive Project feature to create a single compressed Quartus II Archive File (**.qar**) of your project containing all your design, project, and settings files. The **.qar** file contains all the files, including the Quartus II Default Settings File (**.qdf**), required to perform a full compilation to restore the original results. The **.qdf** file contains all the project and assignment default settings from the current version of the Quartus II software. It is necessary to archive the **.qdf** file to preserve your results when you restore the archive in a different version of the Quartus II software. For more information about the **.qdf** file, refer to "Quartus II Default Settings File" on page 4–22.

You can easily share projects between engineers with the archive file generated by the Archive Project feature.

Archive a Project

To archive a project, perform the following steps:

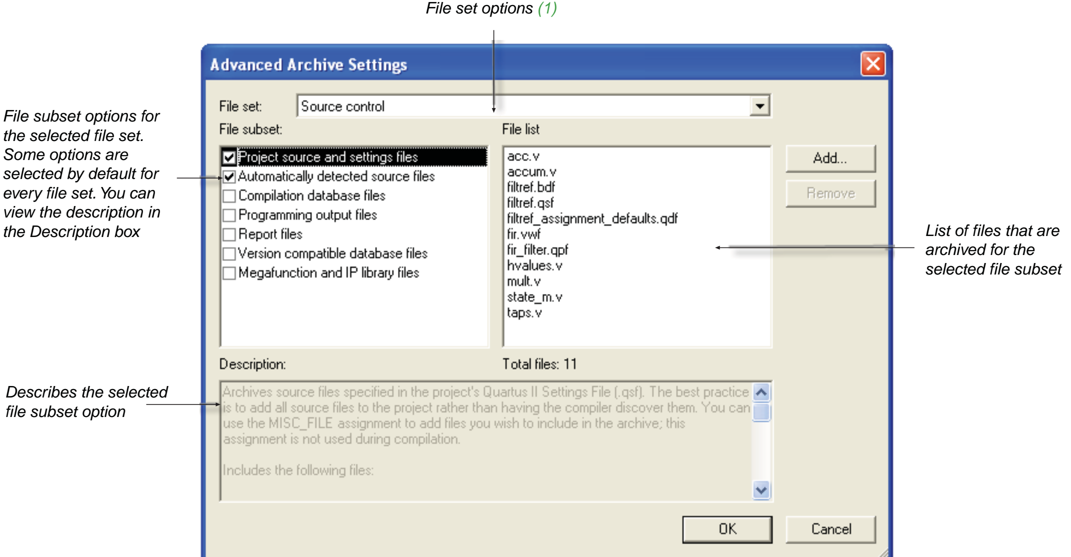
1. Create a new project or open an existing project. On the File menu, click **New Project Wizard** or **Open Project.**
2. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration.**

Altera recommends that you perform analysis and elaboration before archiving a project to ensure that all design files are located and archived.

3. On the Project menu, click **Archive Project**. The **Archive Project** dialog box appears.
4. Under **Archive file name**, type the file name of the **.qar** file that you want to archive, or click **Browse** to select a **.qar** file name.
5. Click **Advanced...** for more archiving options. You can choose the type of files you want to archive by selecting the file set type from the list. The default Source Control file set includes all project source files and settings files. The default archive does not include the compilation results, or post-compilation netlists for incremental compilation. However, the imported Quartus II Exported Partition (**.qxp**) files are included by default for all file set options if you run the Quartus II incremental compilation.

© March 2009 Altera Corporation

Figure 4–4. Advanced Archive Settings Dialog Box



Note to Figure 4–4:

(1) For more information about file set options, refer to Table 4–1 on page 4–7.

Altera recommends that you include all source files in your Quartus II project rather than having the compiler discover them. Specify all your source files in the **.qsf** file, using the **Add/Remove Files in Project** command from the Project menu. If all source files are not specified in your project, you can click **Add...** to manually add files to the archive.

6. Click **Archive**.

Table 4–1 shows the types of file set options.

Table 4–1. File Set Options

File Set	Description
Source control	This is the default setting for the Quartus II software archive feature. This file set includes project source files specified in the **.qsf** and all automatically detected source files. The file set also includes any imported Quartus II Exported Partition (**.qxp**) files for incremental compilation, because top-level projects do not include source files for imported partitions.
Source control with compilation database	This file set includes files from the Source control settings, plus the compilation database files. This file set includes the previous compilation results, so you can analyze or view the results in tools such as the Chip Planner. The compilation database is required to preserve post-synthesis or post-fitter netlists for incremental compilation. However, the archived file for this setting is larger in size.
Service request	This file set includes files from the Source control settings, the report files and programming output files are included. This file set is useful when you must attach your design files to a service request with Altera technical support, and for debugging purposes.

Table 4–1. File Set Options

File Set	Description
Service request with compilation database	This file set is useful when you want to include the database files so that technical support can analyze or view the results in tools such as the Chip Planner without performing a new compilation, or to duplicate the post-synthesis or post-fitter netlists for incremental compilation.
Custom	Use this file set to select the types of files that you want to include in the archive from the file subset list. When you add or remove file subsets from one the predefined file set, the file set selection automatically changes to Custom.

Restore an Archived Project

To restore an archived project, perform the following steps:

1. On the Project menu, click **Restore Archived Project**.
2. In the **Archive file name** box, type the file name of the **.qar** file you wish to restore, or click **Browse** to select a **.qar** file.
3. In the **Destination folder** box, specify the directory path in which you will restore the contents of the **.qar** file, or browse to a directory.
4. Click **Show log** to view the Quartus II Archive Log File (**.qarlog**) for the project you are restoring from the **.qar** file (optional).
5. Click **OK**.
6. If you did not include the compilation database files in the project archive using the Advanced File Sets, you must recompile the project.

   ☞ Altera recommends that you recompile the project if the Project Navigator does not display the correct source-file paths for a restored project.

   ☞ To remove any unwanted restored megafunction files from your project, perform the following steps:

   a. Locate and delete the megafunction files in: *<drive><restored_project>*/**megafunctions**.
   b. Locate and delete the XML file in: *<drive><restored_project>/<revision.name>***.cbx.xml**. The ***_cbx.xml** file is revision-based. Altera uses the XML format to describe data in different file types for revision report, design metadata, component declaration, ports and nodes assignments, device utilization and more. For more information about the XML files, you can refer to the Quartus II Help.

Version Compatible Databases

The Version Compatible Database feature allows you to export a version compatible database and import it into a later version. For example, using one set of design files, you can export a database generated from the Quartus II software version 7.2 and import it into the Quartus II software version 8.0 and later without recompiling your design. Using this feature eliminates unnecessary compilation time.

This feature is not supported immediately for new device families added to the Quartus II software, because some device information might still be preliminary. Version compatible database support is added for each new device family in a later software release after the device family introduction.

Migrate to a New Version

To migrate a design from one Quartus II software version to a newer version, perform the following steps:

1. On the File menu, open the older version of the Quartus II software project by clicking **Open Project** and browse to select the Quartus II project file.
2. On the Project menu, click **Copy Project** to make a new copy of the project. The older version closes and the copied project opens.
3. If you have not run Analysis & Synthesis for the new version, you must do so before exporting the database. On the Project menu, click **Export Database**. By default, the database is exported to the **export_db** directory of the copied project. If desired, a new directory can be created.
4. Open the copied project from the new version of the Quartus II software. The Quartus II software deletes the existing database but not the exported database.
5. On the Project menu, click **Import Database**. By default, the directory of the database you just exported is selected. Select the exported database and the Quartus II software imports the version compatible database files.

Save the Database in a Version Compatible Format

To save the database in a version compatible format during a full compilation, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Compilation Process Settings**. The **Compilation Process Settings** page appears.
3. Turn on the **Export version compatible database** option.
4. Browse to the directory in which you want to save the database.
5. Click **OK**.

Quartus II Project Platform Migration

When moving your project from one computing platform to another, you must think about the following cross-platform issues:

- "Filenames and Hierarchies"
- "Specifying Libraries"
- "Quartus II Search Path Precedence Rules"
- "Quartus II-Generated Files for Third-Party EDA Tools"
- "Migrating Database Files between Platforms"

Filenames and Hierarchies

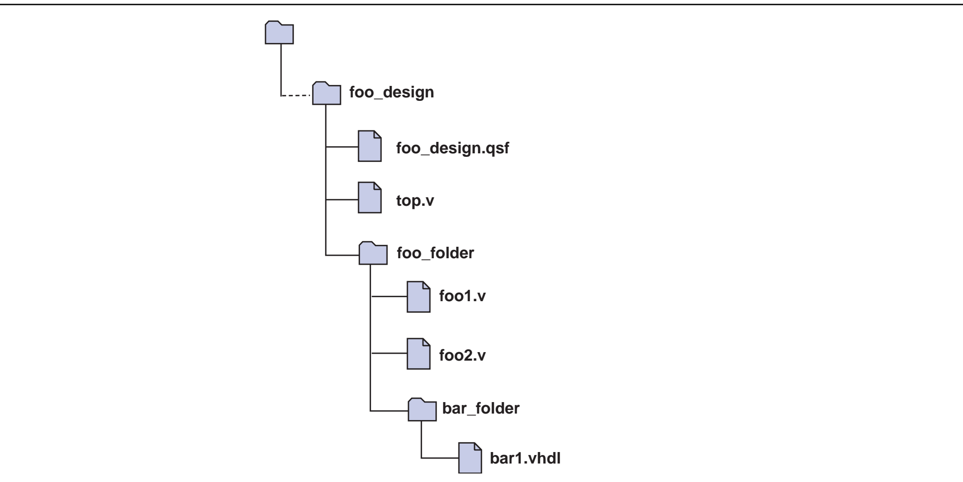
To ensure migration across platforms, you must consider a few basic differences between operating systems when naming source files, especially when interacting with these from the system-command prompt or a Tcl script:

- Some operating system file systems are case-sensitive. When writing scripts, ensure you specify paths exactly, even if the current operating system is not case-sensitive. For best results, use lowercase letters when naming files.
- Use a character set common to all the used platforms.
- Do not convert the forward-slash / and back-slash \ path separators in the **.qsf** file because the Quartus II software changes all back-slash \ path separators to forward-slashes /.
- Observe the shortest file name length limit of the different operating systems you are using.

Altera recommends that you avoid naming the project directory with spaces in between. You can rename the directory using a symbol such as the underscore (_) as a place holder instead of spaces (for example, "my_design" instead of "my design").

You can specify files and directories inside a Quartus II project as paths relative to the project directory. For instance, for a project titled **foo_design** with a directory structure shown in Figure 4–5, specify the source files as: **top.v, foo_folder/foo1.v**, **foo_folder/foo2.v**, and **foo_folder/bar_folder/bar1.vhdl**.

Figure 4–5. All-Inclusive Project Directory Structure



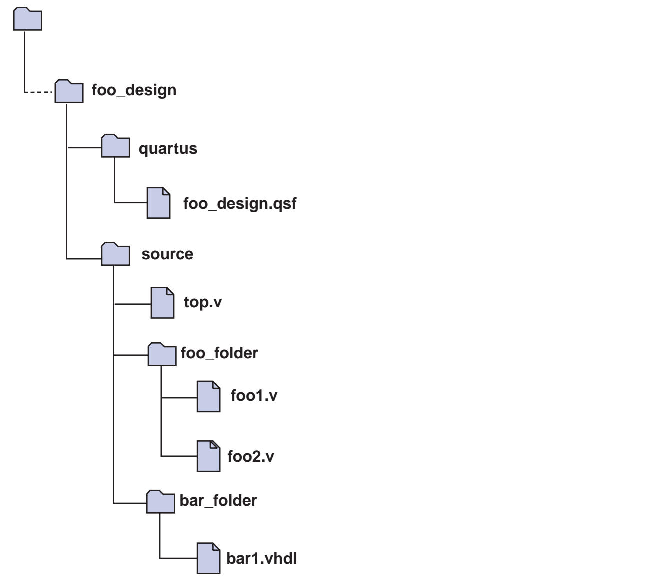
If the **.qsf** file is in a directory that is separate from the source files, you can specify paths using the following options:

- Relative paths
- Absolute paths
- Libraries

Relative Paths

If the source files are very near the Quartus II project directory, you can express relative paths using the `..` notation. For example, given the directory structure shown in Figure 4–6, you can specify **top.v** as **../source/top.v** and **foo1.v** as **../source/foo_folder/foo1.v**.

Figure 4–6. Quartus II Project Directory Separate from Design Files



When you copy a directory structure to a different platform, ensure that any subdirectories are in the same hierarchical structure and relative path as in the original platform.

Specifying Libraries

You also can specify the directory (or directories) containing source files as a library that the Quartus II software searches when you compile your project. A Quartus II library is a directory containing design files used by your Quartus II project. You can specify the following two kinds of libraries in the Quartus II software:

- User libraries, which apply to a specific project
- Global libraries, which all projects use

Use the procedures in this section to specify user or global libraries.

All files in the directories specified as libraries are relative to the libraries. For example, if you want to include the file **/user_lib1/foo1.v** and the **user_lib1** directory is specified as a user library in the project, the **foo1.v** file can be specified in the **.qsf** file as **foo1.v**. The Quartus II software searches directories that are specified as libraries and finds the file.

Specifying User Libraries

To specify user libraries from the GUI, on the Assignments menu, click **Settings**, and select **Libraries.** Type the name of the directory in the **Project Library name** box, or browse to the name of the directory. User libraries are stored in the **.qsf** file of the current revision.

Specifying Global Libraries

To specify global libraries from the GUI, on the Tools menu, click **Options**, and select **Global User Libraries (All Project).** Type the name of the directory in the **Library name** box, or browse to the name of the directory. Global libraries are stored in the **quartus2.ini** file.

The Quartus II software searches for the **quartus2.ini** file in the following order:

- USERPROFILE, for example, **C:\Documents and Settings**<*user name*>
- Directory specified by the **TMP** environmental variable
- Directory specified by **TEMP** environmental variable
- Root directory, for example, **C:**

For UNIX and Linux users, the file is created in the **altera.quartus** directory under the **<*home*>** directory, if the **altera.quartus** directory exists. If the **altera.quartus** directory does not exist, the file is created in the **<*home*>** directory.

Whenever you specify a directory name in the GUI or in Tcl, the name you use is maintained verbatim in the **.qsf** file rather than resolved to an absolute path.

If the directory is outside of the project directory, the path returned in the dialog box is an absolute path.You can use the **Browse** button in either the **Settings** dialog box or the **Options** dialog box to select a directory. You can change the absolute path to a relative path by editing the absolute path displayed in the **library name** field to create a relative path before you click **Add** to put the directory in the **Libraries** list or select from the **Libraries** list and double-click to edit the path.

When copying projects that specify user libraries, you must either copy your user library files along with the project directory or ensure that your user library files exist in the target platform.

Quartus II Search Path Precedence Rules

If two files have the same file name, the file found is determined by the Quartus II software's search path precedence rules. The Quartus II software resolves relative paths by searching for the file in the following directories and order:

1. The project directory, which is the directory containing the **.qsf** file.
2. The project's database (**db)** directory.
3. User libraries are searched in the order specified by the **SEARCH_PATH** setting of the **.qsf** file for the current revision.
4. Global user libraries are searched in the order specified by the **SEARCH_PATH** setting on the **Global User Libraries** page in the **Options** dialog box.

Beginning with the Quartus II software version 9.0, Altera recommends that you use the **SEARCH_PATH** assignment to define the user libraries. You can have multiple **SEARCH_PATH** assignments. However, you can specify only one source directory for each **SEARCH_PATH** assignment. For more information about **SEARCH_PATH** assignments, refer to Example 4–5 on page 4–25.

5. The Quartus II software **libraries** directory, for example, ***<Quartus II Software Installation directory>*\libraries**. For more information about libraries, refer to "Specifying Libraries Using Scripts" on page 4–25.

Quartus II-Generated Files for Third-Party EDA Tools

The project archive and copy feature in the Quartus II software does not include Quartus II generated files for third-party EDA tools such as the Verilog Output Files (**.vo**), VHDL Output Files (**.vho**), Standard Delay Format Output Files (**.sdo**) output netlist files, Stamp model files, PartMiner XML-Format Files (**.xml**), and IBIS Output Files (**.ibs**). When you archive the design project, you can save the database in a version compatible format during a full compilation and include the version compatible database files in your project archive. For more information about saving the database in a version compatible format and archiving projects, refer to "Save the Database in a Version Compatible Format" on page 4–9 and "Archiving Projects" on page 4–23.

If you want to copy your project to another platform, you can regenerate the output netlist or output files by performing the following steps:

1. Import the version compatible database (for more information, refer to "Migrate to a New Version" on page 4–9).
2. You can run the Classic Timing Analyzer or the TimeQuest Timing Analyzer from the Processing menu followed by the EDA Netlist Writer.

If you want to restore your project, you can regenerate the output netlist or output files by performing the following steps:

1. Restore the design project. For more information about restoring an archived project, refer to "Restore an Archived Project" on page 4–8.
2. Import the version compatible database. For more information about migrating to a new version, refer "Migrate to a New Version" on page 4–9.
3. You can run the Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer from the Processing menu followed by the EDA Netlist Writer.

When you create version compatible databases, you do not need to recompile your design as you move across platform.

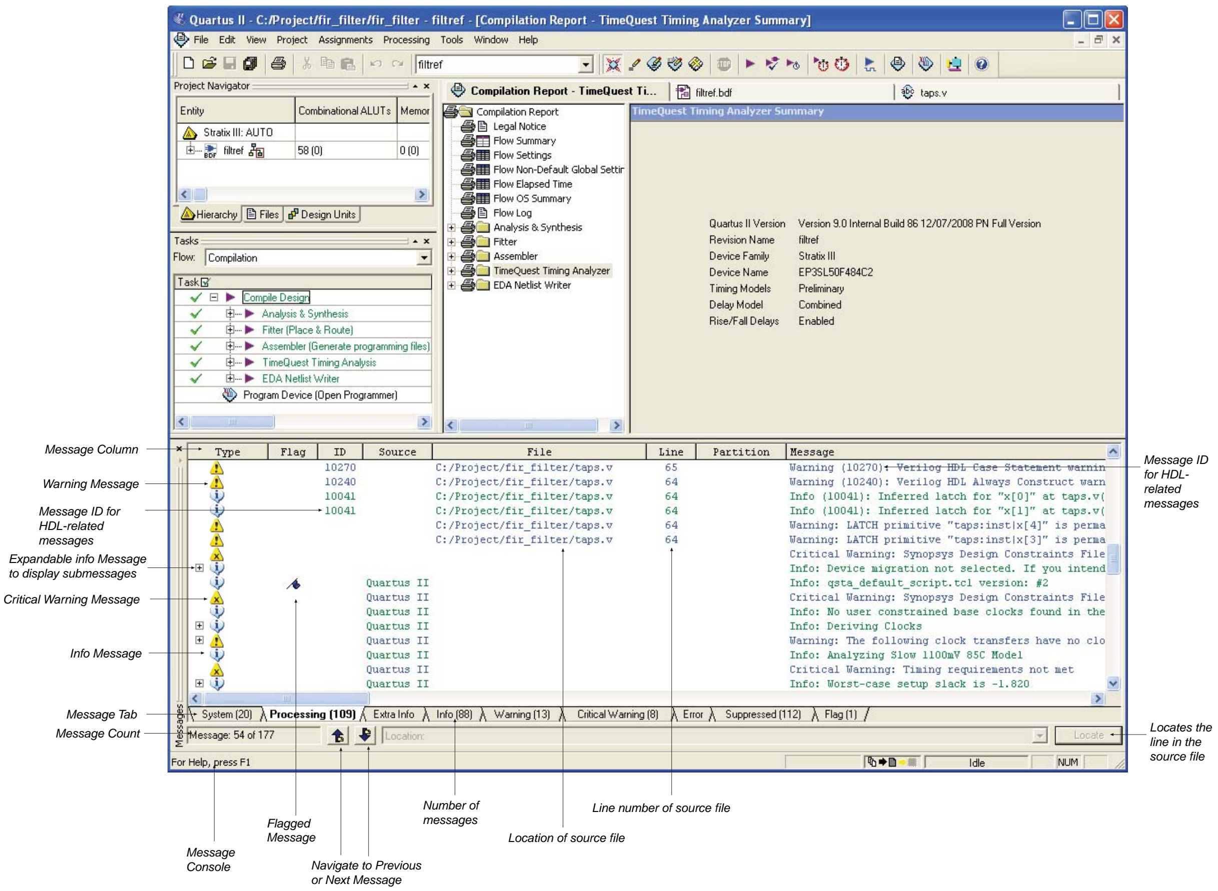
Migrating Database Files between Platforms

There is nothing inherent in the file format and syntax of the exported version compatible database files that might cause problems when migrating the files to other platforms. However, the contents of the database can cause problems for platform migration. For example, using the absolute paths in version compatible database files generated by the Quartus II software can cause problems for migration. Altera recommends that you change the absolute paths to relative paths before migrating files whenever possible.

Working with Messages

The Quartus II software generates various types of messages, including Information, Warning, and Error messages. Some messages include information on software status during a compilation and alert you to possible problems with your design. Messages are displayed in the Messages box in the Quartus II GUI (Figure 4–7), and written to standard out when you use command-line executables. In both cases, messages are written to Quartus II report files.

Figure 4–7. Viewing Quartus II Messages



You can right-click on a message in the Message window to get help for the message, locate the source of the message of your design, and manage messages.

Messages provide useful information if you take time to review them after each compilation. However, it can be tedious if there are thousands of them. The Quartus II software includes features to help you manage messages. These are described in the following subsections.

Messages Window

By default, the Messages window displays eight message tabs (Table 4–2), which makes it easy to review all messages of a certain type.

Table 4–2. Quartus II Message Tabs

Message Tab	Icon	Description
System	—	Displays messages that are unrelated to processing your design. For example: messages generated during programming are displayed in the **System** tab.
Processing	—	Displays messages that are generated when the Quartus II software processes your most recent compilation, simulation, or software build; timing analysis messages appear as part of the compilation messages.
Info	—	Displays general informational messages during a compilation, simulation, or software build. For example: compilation-success messages, and information about your design that might be helpful but is not necessarily a cause for concern such as warning or error messages.
Extra Info	—	Displays detailed informational messages about the operations for designers. For example: extra fitting information messages.
Warning		Displays strong warning messages generated during a compilation, simulation, or software build.
Critical Warning		Displays critical warning messages generated during a compilation, simulation, or software build.
Error		Displays processing and compilation error messages generated during a compilation, simulation, or software build. Error messages can sometimes stop processing and cannot be disabled.
Suppressed		Displays suppressed messages during the last processing operation.
Flag		Displays flagged messaged. In any tab, a flag icon appears in the optional **Flag** column to indicate a flagged message.

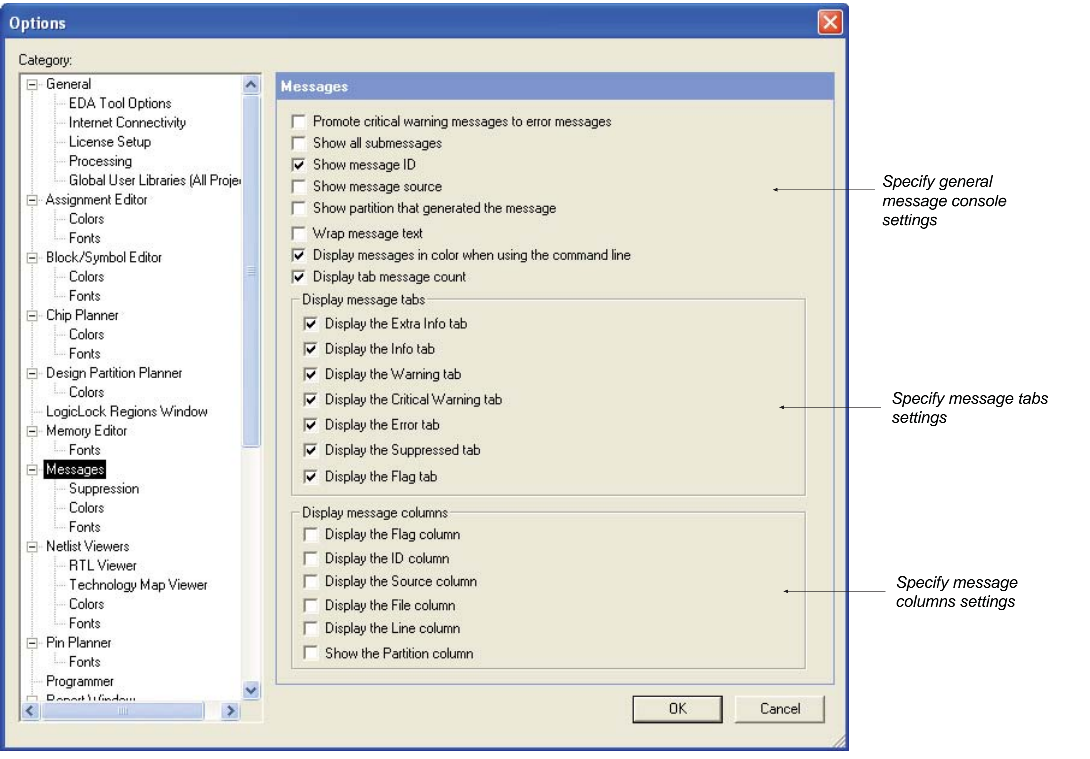
The **Info**, **Extra Info**, **Warning**, **Critical Warning**, and **Error** tabs display messages grouped by type.

A **Message ID** displays in parentheses at the beginning of some HDL-related messages, which you can use to enable or disable the specific messages. The **Message Status** box indicates the number of the message currently selected and the total number of messages. The **Message Location** list indicates the line number in the source file or files in which the message originates. For each compilation, any number of messages can be flagged and you can later choose to export or import flagged messages to or from another Quartus II project.

You can control which tabs are displayed by setting the options on the **Messages** page in the **Options** dialog box on the Tools menu (Figure 4–8). You can select the message suppression settings at the **Messages** page in the **Options** dialog box (see Figure 4–9 on page 4–17) or you can create suppression rules to suppress any set of non-critical messages with the **Message Suppression Manager** dialog box.

For more information about Message Suppression and Message Suppression Manager, refer to "Message Suppression" on page 4–17 and "Message Suppression Manager" on page 4–19.

Figure 4–8. Message Tab Options



Hiding Messages

In the Messages window, you can hide all messages of a particular type. For example, to hide Info messages, perform the following steps:

1. On the **Processing** tab, right-click in the **Processing message** dialog box and click the **Hide** option.
2. Select the **Hide** options type (point to **Hide** and click **Hide Critical Warning Messages**, for example).

All messages of the specified type are removed from the list of messages in the **Processing** tab, although they are still included in the separate tabs corresponding to the message type. For example, if you hide Info messages, no Info messages are shown in the **Processing message** dialog box, but all the Info messages are shown in the Info messages window.

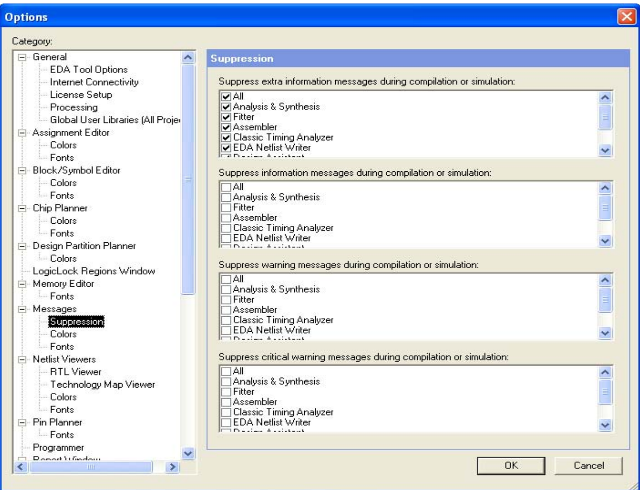
Message Suppression

You can use message suppression to reduce the number of messages to be reviewed after a compilation by preventing individual messages and entire categories of messages from being displayed. For example, if you review a particular message and determine that it is not caused by something in your design that needs to be changed or fixed, you can suppress the message so that it is not displayed during subsequent compilations. This saves time because you see only new messages during subsequent compilations.

Every time you add a message to be suppressed, a suppression rule is created. Suppressing exact selected messages adds patterns that are exact strings to the suppression rules. Suppressing all similar messages adds patterns with wildcards to the suppression rules.

Furthermore, you can suppress all messages of a particular type in a particular stage of the compilation flow. On the Tools menu, click **Options**. In the **Category** list, select **Suppression** from under the **Messages** section (Figure 4–9).

Figure 4–9. Controlling Suppression Messages



Suppressing individual messages is controlled in two locations in the Quartus II GUI. You can right-click on a message in the Messages window and choose commands in the Suppress sub-menu entry. To open the Message Suppression Manager, right-click in the Messages window. From the Suppress sub-menu, click **Message Suppression Manager**. For more information about Message Suppression Manager, refer to "Message Suppression Manager" on page 4–19.

Message Suppression Methods

There are three methods that you can use to create suppression rules:

- **Suppress Exact Selected Messages**
- **Suppress All Similar Messages**
- **Suppress All Flagged Messages**

If you suppress a message with the exact selected messages option, only messages that match the exact text are suppressed during subsequent compilations. The **All Similar Messages** option behaves like a wildcard pattern on variable fields in messages and the **Flagged Messages** option only suppresses flagged messages.

The following message is an example of suppressing all similar messages:

```
Info: Found 1 design units, including 1 entities, in source file mult.v.
```

This type of message is common during synthesis and is displayed for each source file that is processed with varying information about the number of design units, entities, and source file name.

Help for this message shows it is in the form Found <*number*> design units, including <*number*> entities, in source file <*name*>. Choosing to suppress all similar messages effectively replaces the variable parts of that message (<*number*>, <*number*>, and <*name*>) with wildcards, resulting in the following suppression rule:

```
Info: Found * design units, including * entities, in source file *.
```

As a result, all similar messages (ones that match the pattern) are suppressed.

Message Supression Details and Limitations

The following limitations apply to which messages can be suppressed and how they can be suppressed:

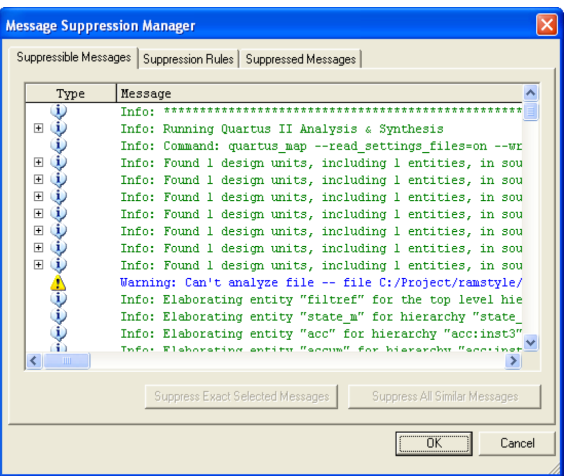
- You cannot suppress error messages or messages with information about Altera legal agreements.
- Suppressing a message also suppresses all its submessages, if any.
- Suppressing a submessage causes matching submessages to be suppressed only if the parent messages are the same.
- You cannot create your own custom wildcards to suppress messages.
- You must use the Quartus II GUI to manage message suppression, including choosing messages to suppress. These messages are suppressed during compilation in the GUI and when using command-line executables.
- Messages are suppressed on a per-revision basis, not for an entire project. Information about which messages to suppress is stored in a file called <*revision*>**.srf**. If you create a revision based on a revision for which messages are suppressed, the suppression rules file is copied to the new revision. You cannot make all revisions in one project use the same suppression rules file.
- You cannot remove messages or modify message suppression rules while a compilation is running.

Message Suppression Manager

You can use Message Suppression Manager to view and suppress messages, view and delete suppression rules, and view suppressed messages.

Right-click anywhere in the Messages window and click **Message Suppression Manager** from the Suppress sub-menu. The Message Suppression Manager has three tabs labeled **Suppression Messages**, **Suppression Rules**, and **Suppressed Messages** (Figure 4–10).

Figure 4–10. Message Suppression Manager Dialog Box



Suppressible Messages

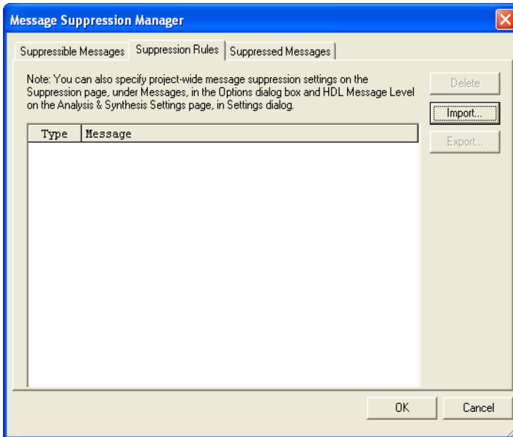
Messages that are listed in the **Suppressible Messages** tab are messages that were not suppressed during the last compilation. These messages can be suppressed. The **Select All Similar Messages** option in the right-click menu selects messages according to the example described in the "Message Suppression Methods" on page 4–18. You can select all similar messages to see which messages are suppressed if you choose to suppress all similar messages.

Suppression Rules

Items listed in the **Suppression Rules** tab are the patterns that the Quartus II software uses to determine whether to suppress a message. Messages matching any of the items listed in the **Suppression Rules** tab are suppressed during compilations (Figure 4–11). At this tab, you can also perform the following actions:

- Delete—Remove selected rules from the **Suppression Rules** tab
- Import—Open the **Import Message Suppression Rule File** dialog box. This allows you to import suppression rules from another project or a revision of the current project with the Quartus II Message Suppression Rule File (**.srf**).
- Export—Open the **Export Message Suppression Rule File** dialog box and create **.srf** file containing the suppression rules listed in the **Suppression Rules** tab for the current revision

Figure 4–11. Message Suppression Manager



An entry in the **Suppression Rules** tab that includes a message with submessages indicates the submessage is suppressed only when all of its parent messages match.

You can stop suppressing messages by deleting the suppression rules that match them (causing them to be suppressed). Merely deleting suppression rules does not cause the formerly suppressed messages to be added to the messages generated during the previous compilation; you must recompile the design for the changed suppression rules to take effect.

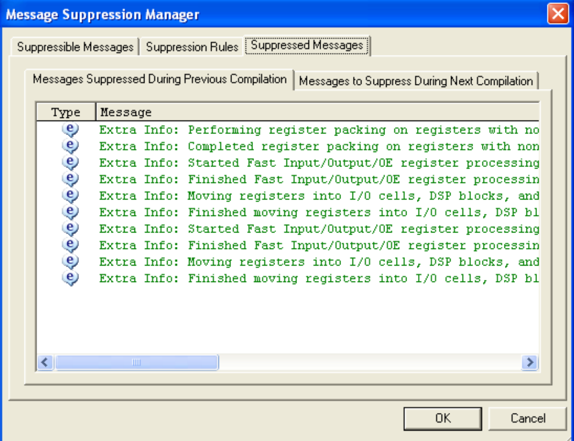
Suppressed Messages

Messages listed in the **Suppressed Messages** tab are divided into two sub-tabs:

- **Messages Suppressed During Previous Compilation**
- **Messages to Suppress During Next Compilation**

The messages listed in the **Messages Suppressed During Previous Compilation** sub-tab are all the suppressed messages from the previous compilation (Figure 4–12).

Figure 4–12. Messages Suppressed During Previous Compilation



These messages are also listed in the **Suppressed** tab in the Messages window. Messages listed in the **Messages to Suppress During Next Compilation** are messages that are suppressed during the next compilation that match suppression rules created after the last compilation finished.

In addition to appearing in the **Suppressed** tab in the Messages window, suppressed messages are included in a **Suppressed Messages** entry in the Quartus II compilation report, viewable in the GUI. Suppressed messages are not included in the <*revision*>**.**<*module*>**.rpt** text files; they are written to a separate text report file called <*revision name*>**.**<*module*>**.smsg**.

Quartus II Settings File

All assignments made in the Quartus II software are stored as Tcl commands in the **.qsf** file. The **.qsf** file is a text-based file containing Tcl commands and comments. The **.qsf** file is not a Tcl script and does not support the full Tcl scripting language.

As you make assignments in the Quartus II software, the assignments are either stored temporarily in memory or written out to the **.qsf** file. This is determined by the **Update assignments to disk during design processing only** option, which is located on the Tools menu under **Options** on the **Processing** page. If the option is turned on, all assignments are stored in memory and are written to the **.qsf** file when a compilation has started or when you save or close the project. By saving assignments to memory, the performance of the software is improved because it avoids unnecessary reading and writing to the **.qsf** file on the disk. This performance improvement is seen more dramatically when the project files are stored on a remote data disk.

You can add lines of comments into the **.qsf** file, as shown in Example 4–1:

Example 4–1.

```
# Assignments for input pin clk
# Clk is being driven by FPGA 1
set_location_assignment PIN_6 -to clk
set_instance_assignment -name IO_STANDARD "2.5 V" -to clk
```

Sourcing other **.qsf** files is supported using the following Tcl command:

```
source <filename>.qsf
```

QSF Format Preservation

The Quartus II software maintains the order of assignments in the **.qsf** file. When you make new assignments, they are appended to the end of the **.qsf** file. If you modify an assignment, the corresponding line in the **.qsf** file is modified and the order of assignments in the **.qsf** file is maintained except when you add and remove project source files, or when you add, remove, and exclude members from an assignment group. In these cases, all assignments are moved to the end of the **.qsf** file. For example, if you add a new design file to the project, the list of all your design files is removed from its current location in the file and moved to the end of the **.qsf** file.

The header at the beginning of the **.qsf** file is written only if the **.qsf** file is newly created.

The Quartus II software preserves all spaces and tabs for all unmodified assignments and comments. When you make a new assignment or modify an existing assignment, the assignment is written using the default formatting.

Quartus II Default Settings File

The **.qdf** file contains all the project and assignment default settings from the current version of the Quartus II software. The **assignment_defaults.qdf** file, located in the **bin** directory of the Quartus II installation path, is used to ensure consistent results when defaults are changed between versions of the Quartus II software.

The Quartus II software reads assignments from various files and stores the assignments in memory. The Quartus II software reads settings files in the following order and assignments in subsequent files take precedence over earlier ones:

1. **assignment_defaults.qdf** from <*Quartus II Installation directory*>**/bin or bin64**
2. **assignment_defaults.qdf** from project directory
3. <*revision name*>**_assignment_defaults.qdf** from project directory
4. <*revision name*>**.qsf** from project directory

As each new file is read, if an existing assignment from a previous file matches (following rules of case sensitivity, multi-value fields as well as other rules), the old value is removed and replaced by the new value. For example, if the first file has a non multi-valued assignment A=1, and the second file has A=2, the assignment A=1, stored in memory, is replaced by A=2.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For more information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp
```

The *Quartus II Scripting Reference Manual* includes the same information in Portable Document Format (**.pdf**) format.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about all settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Reference Manual*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

 © March 2009 Altera Corporation

Managing Revisions

You can use the following commands to create and manage revisions. For more information about managing revisions, including creating and deleting revisions, setting the current revision, and getting a list of revisions, refer to "Creating and Deleting Revisions" on page 4–3.

Creating Revisions

The `-based_on` and `-set_current` options are optional. You can also use `-copy_results` option to copy results from "based_on" revision. The following Tcl command creates a new revision called **speed_ch**, based on a revision called **chiptrip**, and sets the new revision as the current revision:

```
create_revision speed_ch -based_on chiptrip -set_current
```

Setting the Current Revision

The `-force` option enables you to open the revision that you specify under revision name and overwrite the compilation database if the database version is incompatible. Use the following Tcl command to specify the current revision:

```
set_current_revision-force <revision name>
```

Getting a List of Revisions

Use the following Tcl command to get a list of revisions in the opened project:

```
get_project_revisions <project_name>
```

Deleting Revisions

Use the following Tcl command to delete a revision:

```
delete_revision <revision name>
```

Archiving Projects

You can archive projects with a Tcl command or with a command run at the system command prompt.

The following Tcl command creates a project archive with the default settings and overwrites the specified archived file if it already exists:

```
project_archive archive.qar -overwrite
```

You can change default settings with the `project_archive` command by using options such as `-all_revisions`, `-include_libraries`, `-include_outputs`, `-use_file_set` <*file_set*> and `-version_compatible_database`. Type the following command at a command prompt to create a project archive called **top**:

```
quartus_sh --archive top ↵
```

You can use the `-overwrite` option to overwrite the existing archive file.

Restoring Archived Projects

You can restore archived projects with a Tcl command or with a command run at a command prompt. For more information about restoring archived projects, refer to "Restore an Archived Project" on page 4–8.

The following Tcl command restores the project archive named **archive.qar** in the **restored** subdirectory and overwrites existing files:

```
project_restore archive.qar -destination restored -overwrite
```

Type the following command at a command prompt to restore a project archive:

```
quartus_sh --restore archive.qar ↵
```

Importing and Exporting Version Compatible Databases

You can import and export version compatible databases with either a Tcl command or a command run at a command prompt. For more information about importing and exporting version compatible databases, refer to "Version Compatible Databases" on page 4–8.

The `flow` and `database_manager` packages contain commands to manage version compatible databases.

Use the following Tcl commands from the `database_manager` package to import or export version compatible databases.

```
export_database <directory>
import_database <directory>
```

Example 4–2 shows the Tcl commands from the `flow` package to import or export version compatible databases. If you use the flow package, you must specify the database directory variable name.

Example 4–2.

```
set_global_assignment -name VER_COMPATIBLE_DB_DIR <directory>
execute_flow -flow export_database
execute_flow -flow import_database
```

Example 4–3 shows Tcl commands to automatically generate version compatible databases after every compilation.

Example 4–3.

```
set_global_assignment -name AUTO_EXPORT_VER_COMPATIBLE_DB ON
set_global_assignment-name VER_COMPATIBLE_DB_DIR <directory>
```

The **quartus_cdb** and the **quartus_sh** executables provide commands to manage version compatible databases (Example 4–4).

Example 4–4.

```
quartus_cdb <project> -c <revision>--export_database=<directory> ↵
quartus_cdb <project> -c <revision> --import_database=<directory>↵
quartus_sh -flow export_database <project> -c \ <revision> ↵
quartus_sh -flow import_database <project> -c \ <revision> ↵
```

Specifying Libraries Using Scripts

In Tcl, use commands in the `::quartus::project` package to specify user libraries. To specify user libraries, use the `set_global_assignment` command. To specify global libraries, use the `set_user_option` command. Example 4–5 shows the typical usage of the `set_global_assignment` and `set_user_option` commands.

Example 4–5. Commands to Specify User Libraries using the SEARCH_PATH Assignment

```
set_global_assignment -name SEARCH_PATH "../other_dir/library1"
set_global_assignment -name SEARCH_PATH "../other_dir/library2"
set_global_assignment -name SEARCH_PATH "../other_dir/library3"
set_user_option -name SEARCH_PATH "../other_dir/library1"
set_user_option -name SEARCH_PATH "../other_dir/library2"
set_user_option -name SEARCH_PATH "../other_dir/library3"
```

To report any user libraries specified for a project and any global libraries specified for the current installation of the Quartus II software, use the `get_global_assignment` and `get_user_option` Tcl commands. Example 4–6 shows the Tcl script outputs the user paths and global libraries for an open Quartus II project.

Example 4–6. Commands to Report Specified User Libraries

```
get_global_assignment -name SEARCH_PATH
get_user_option -name SEARCH_PATH
```

Conclusion

Designers often try different settings and versions of their designs throughout the development process. Quartus II project revisions facilitate the creation and management of different assignments and settings. Project archives are useful to save your results, or pass designs between different members of a team. In addition, understanding how to smoothly migrate your projects from one computing platform to another, controlling messages, and reducing compilation time are important as well. The Quartus II software facilitates efficient management of your design to accommodate today’s sophisticated FPGA designs.

Referenced Documents

This chapter references the following documents:

- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Reference Manual*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 4–3 shows the revision history for this chapter.

Table 4–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Updated for the Quartus II software version 9.0.	—
November 2008 v8.1.0	Changed to 8½” × 11” page size. No change to content.	Updated for the Quartus II software version 8.1.
May 2008 v8.0.0	Updated references.	Updated for the Quartus II software version 8.0.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

Section II. I/O and PCB Tools

This section provides an overview of the I/O planning process, Altera's FPGA pin terminology, as well as the various methods for importing, exporting, creating, and validating pin-related assignments using the Quartus® II software. This section also describes the design flow that includes making and analyzing pin assignments using the **Start I/O Assignment Analysis** command in the Quartus II software, during and after the development of your HDL design. It also describes interfaces with third-party PCB design tools

This section includes the following chapters:

- Chapter 5, I/O Management
- Chapter 6, Simultaneous Switching Noise (SSN) Analysis and Optimizations
- Chapter 8, Mentor Graphics PCB Design Tools Support
- Chapter 7, Signal Integrity Analysis with Third-Party Tools
- Chapter 9, Cadence PCB Design Tools Support

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

5. I/O Management

QII52013-9.0.0

Introduction

The process of managing I/Os for current leading FPGA devices involves more than just fitting design pins into a package. The increasing complexity of I/O standards and pin placement guidelines are just some of the factors that influence pin-related assignments. The I/O capabilities of the FPGA device and board layout guidelines influence pin location and other types of assignments for each of your design pins. Therefore, it is necessary to begin I/O planning and PCB development even before starting the FPGA design.

Altera provides many resources for I/O Planning. This chapter provides information about entering pin and pin assignment information in the Quartus® II software. You can consult the device-specific pin connection guidelines available on the Altera® website for your board layout. You can also benefit from the Pin Advisors available in the Quartus II software version 8.1 and later. To get updated information about all the resources on I/O planning and board design, refer to the Board Design and I/O Resource Center on the Altera website.

This chapter provides an overview of the I/O planning process, Altera FPGA pin terminology, and the various methods for importing, exporting, creating, and validating pin-related assignments.

For guidelines about PCB designs for Altera high-speed FPGAs, refer to *AN 315: Guidelines for Designing High-Speed FPGA PCBs*.

This chapter contains the following topics:

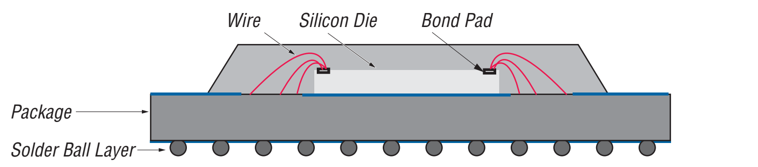
- "Understanding Altera FPGA Pin Terminology" on page 5–2
- "I/O Planning Overview" on page 5–5
- "Device Selection" on page 5–7
- "Early I/O Planning Using the Pin Planner" on page 5–8
- "Importing and Exporting Pin Assignments" on page 5–16
- "Creating Pin-Related Assignments" on page 5–19
- "Validating Pin Assignments" on page 5–56
- "Accepting Fitter Placements—Back-Annotating Assignments" on page 5–74
- "I/O Timing Analysis" on page 5–75
- "Incorporating PCB Design Tools" on page 5–83

Understanding Altera FPGA Pin Terminology

Altera FPGA devices are available in a variety of packages to meet all of your complex design requirements. To describe Altera FPGA pin terminology, this chapter uses a wire bond ball grid array (BGA) package in its examples. On the top surface of the silicon die, there is a ring of bond pads that connect to the I/O pins of the silicon. In a wire bond BGA package, the device is placed in the package and copper wires connect the bond pads to the solder balls of the package. Figure 5–1 shows a cross section of a wire bond BGA package.

For a list of all BGA packages available for each Altera FPGA device, refer to the *Altera Device Package Information Data Sheet*.

Figure 5–1. Wire Bond BGA

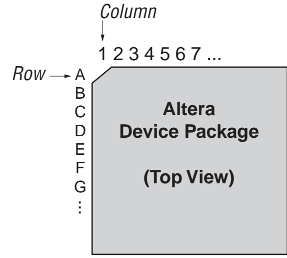


Package Pins

The pins of a BGA package are small solder balls arranged in a grid pattern on the bottom of the package. In the Quartus II software, the package pins are represented as pin numbers. The pin numbers are determined by their locations using a coordinate system with letters and numbers identifying the row and column of the pins, respectively.

The upper-most row of pins is labeled “A” and continues alphabetically as you move downward (Figure 5–2). The left-most column of pins is labeled “1” and continues with increments of 1 as you move to the right. For example, pin number “B4” represents row “B” and column “4.”

Figure 5–2. Row and Column Labeling



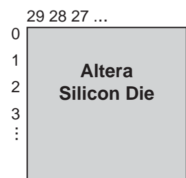
The letters I, O, Q, S, X, and Z are never used in pin numbers. If there are more rows than letters of the alphabet, the alphabet is repeated, prefixed with the letter “A.”

For more information about the pin numbers for your Altera device, refer to the device pin-out information available on the Altera website at www.altera.com.

Pads

Package pins are connected to pads located on the perimeter of the top metal layer of the silicon die (Figure 5–1). Each pad is identified by a pad ID, which is numbered starting at 0, incrementing by 1 in a counterclockwise direction (Figure 5–3).

Figure 5–3. Pad Number Ordering



To prevent signal integrity issues, the Quartus II software uses pin placement rules to validate your pin placements and pin-related assignments. It is important that you understand which pad locations your pins were assigned to, because some pin placement rules describe pad placement restrictions. For example, in certain devices, there is a restriction on the number of I/O pins supported by a VREF pad to ensure signal integrity. There are also restrictions on the number of pads between single-ended input or output pins and a differential pin. The Quartus II software performs pin placement analysis, and if pins are not placed according to pin placement rules, the design compilation fails and the Quartus II software reports an error.

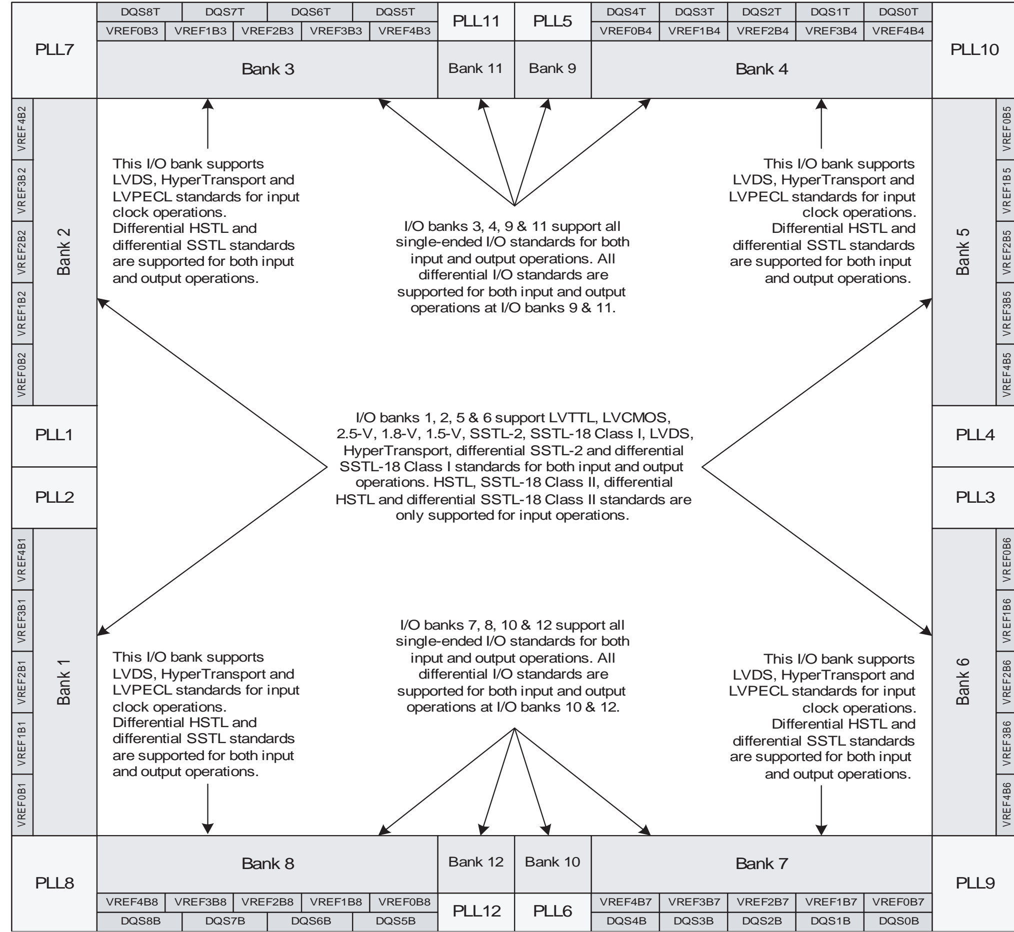
For more information about pin placement guidelines, refer to the *Selectable I/O Standards* chapter in volume 1 of the appropriate device handbook.

I/O Banks

I/O pins are organized into I/O banks designed to facilitate various supported I/O standards. Each I/O bank is numbered and has its own voltage source pins, called `VCCIO`, to offer the highest I/O performance. Depending on the device and I/O standards for the pins within the I/O bank, the specified voltage of the `VCCIO` pin is between 1.5 V and 3.3 V. Each I/O bank can support multiple pins with different I/O standards that share the same V_{CCIO}.

It is important to refer to the appropriate device handbook to determine the capabilities of each I/O bank. For example, the pins in the I/O banks on the left and right side of a Stratix® II device support high-speed I/O standards such as LVDS, whereas the pins on the top and bottom I/O banks support all single-ended I/O standards, including data strobe signaling (DQS) (Figure 5–4). Pins belonging to the same I/O bank must use the same `VCCIO` signal.

Figure 5–4. Stratix II I/O Banks *(Note 1), (2), (3), (4)*



Notes to Figure 5–4:

(1) This figure shows a top view of the silicon die that corresponds to a reverse view for flip chip packages. It is a graphical representation only.

(2) Depending on the size of the device, different device members have a different number of VREF groups. Refer to the pin list and the Quartus II software for exact locations.

(3) Banks 9 through 12 are enhanced phase-locked loop (PLL) external clock output banks.

(4) Horizontal I/O banks feature serializer/deserializer (SERDES) and dynamic phase alignment (DPA) circuitry for high-speed differential I/O standards. For more information about differential I/O standards, refer to the *High-Speed Differential I/O Interfaces with DPA in Stratix II and Stratix II GX Devices* chapter in volume 2 of the *Stratix II Device Handbook*.

VREF Groups

A VREF group is a group of pins that includes one dedicated `VREF` pin as required by voltage-referenced I/O standards. A VREF group is made up of a small number of pins, as compared to the I/O bank, to maintain the signal integrity of the `VREF` pin. One or more VREF group(s) exist in an I/O bank. The pins in a VREF group share the same V_{CCIO} and VREF voltages.

For more information about I/O banks, VREF groups, and supported I/O standards, refer to the *Architecture* and *Selectable I/O Standards* chapters in the appropriate device handbook.

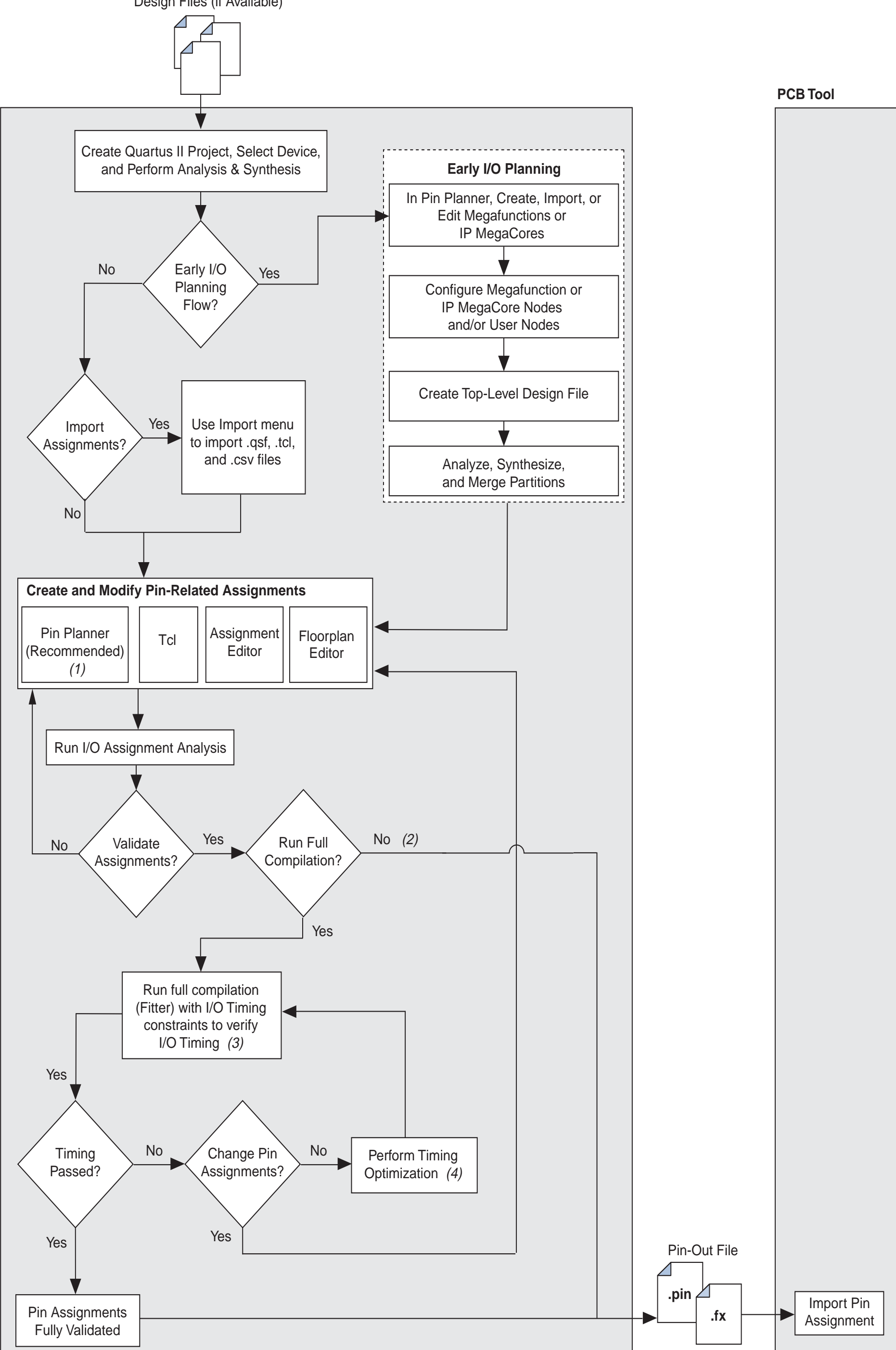
I/O Planning Overview

I/O planning of your FPGA design in Quartus II software includes:

- Selecting a device that meets your logic and I/O requirements, based on the device's supported I/O standards, I/O bank structure, supply voltage requirements such as VREF and V_{CCIO} requirements in I/O banks, available pins for user I/O, power supply requirements, and more.
- Getting your design files ready. The design files contain the top-level ports or top-level interface information. If you do not have the design files, you can use the Early I/O Planning flow to generate a top-level HDL wrapper file.
- Importing any existing assignments from a Tcl script, **.csv**, or **.qsf** file.
- Creating, modifying, and completing all pin-related assignments that include pin location assignments, I/O standards, output loading assignments for output and bidirectional pins, slew rates, current strengths, and more.
- Validating your pin-related assignments while creating them by using the Live I/O Check feature, then running I/O assignment analysis, and finally running the Fitter with timing constraints.
- Generating a validated ***.pin** file for third-party PCB tools.

The method you use to create pin assignments depends on your requirements. If you have not yet designed the PCB, create and validate your I/O assignments in the Quartus II software, then export them to the PCB tool (Figure 5–5). This is the recommended design flow for creating I/O assignments for an FPGA design.

Figure 5–5. Quartus II Software I/O Planning Flow



Notes to Figure 5–5:

(1) Use the Live I/O Check feature in the Pin Planner to validate pin assignments as you create them.

(2) To create the FPGA Xchange file (**.fx**), on the Processing menu, point to **Start** and click **EDA Netlist Writer**. The **.pin** file is created at the *<project_dir>* level. The **.fx** file is created at the *<project_dir/board/.../>* level.

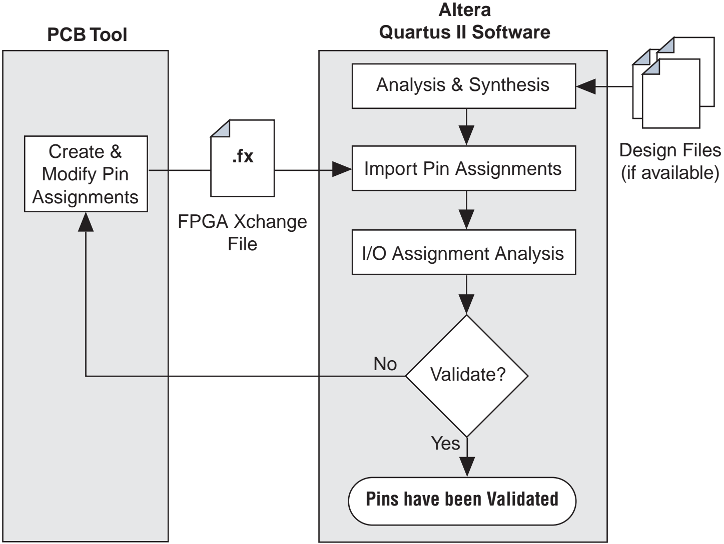
(3) Your design files and constraints must be complete before you begin full compilation. To learn how to create I/O timing constraints, refer to the *TimeQuest Timing Analyzer* and *Classic Timing Analyzer* chapters in the *Timing Analysis* section in volume 3 of the *Quartus II Handbook*.

(4) Refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

If your PCB is partially designed, create your FPGA assignments in your PCB tool and import them into the Quartus II software for validation (Figure 5–6).

Currently, only the Mentor Graphics® I/O Designer PCB tool and the Cadence Allegro PCB tool are supported in this reverse I/O planning flow.

Figure 5–6. I/O Planning Flow Using an FPGA Xchange File from a PCB Tool

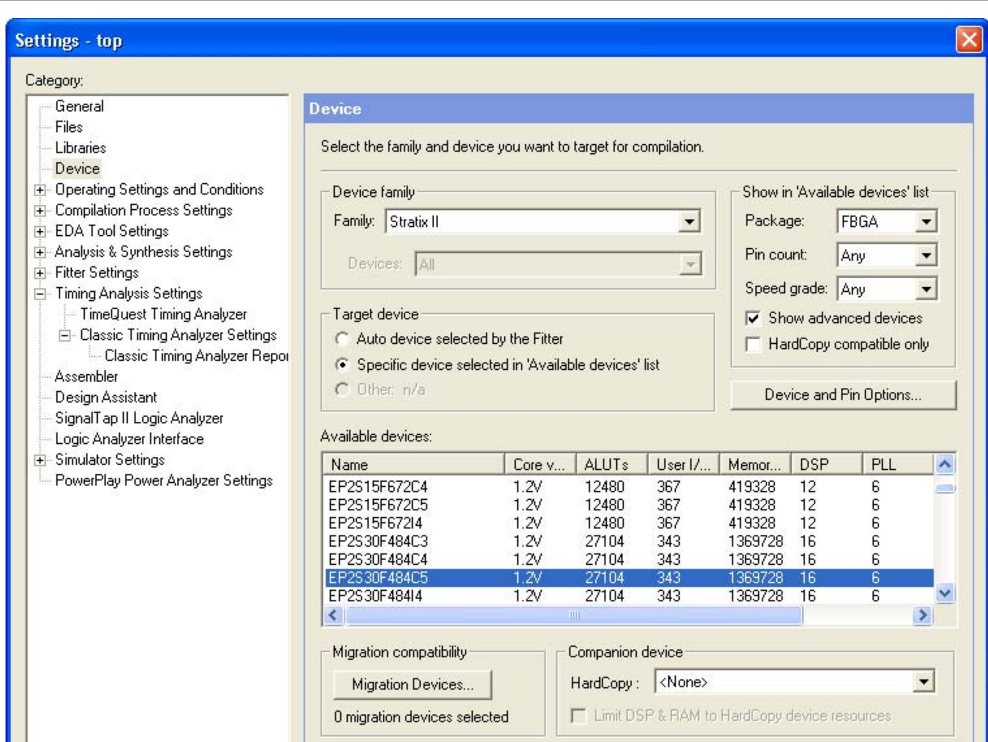


You must refer to the appropriate device handbook, available on the Altera website at www.altera.com to learn about the I/O architecture, supported I/O standards, and their requirements in your device. In the Quartus II software design flow, the most important step in I/O planning is to create, modify, complete, and validate pin-related assignments. The Quartus II software includes the Pin Planner and I/O Assignment Analysis to assist you in I/O planning.

Device Selection

Before you begin pin planning or I/O assignment analysis in the Quartus II software, refer to the device handbooks at www.altera.com to understand the I/O structure, supported I/O standards, available pins for user I/O, clocking schemes and options, and I/O bank structure for different devices. Then, choose an appropriate device from a supported device family for your design. To select a device, on the Assignments menu, click **Device** to open the **Settings** dialog box (Figure 5–7). In the **Family** list, select the device. You can set the other controls under **Show in 'Available devices list'** to filter the **Available devices** list and to select any migration devices. Under the **Available devices** list, select a device.

Figure 5–7. Device Page of the Settings Dialog Box



Early I/O Planning Using the Pin Planner

It might be difficult to plan your I/Os early in the design cycle because the design files, including the top-level design, might not be available yet. However, the interfaces between your FPGA and other devices are typically determined and documented in the design specifications. By adding those interfaces required to connect your FPGA with these other devices in the Pin Planner, you can plan your FPGA I/Os efficiently without design files. The Early I/O Planning flow allows you to create a top-level wrapper file in Verilog HDL or VHDL. The flow includes importing and/or creating any Altera IP MegaCore® functions or Altera megafunctions in the Pin Planner, as well as creating or adding additional top-level pins information, configuring the top-level pins of the design, and creating the top-level HDL file.

Creating a top-level file using the Early I/O planning flow is an optional step in your I/O planning. If you have design files that have complete information about the top-level interfaces and ports, you can skip the Early I/O planning flow step and proceed to making pin-related assignments. The top-level file you create using the Early I/O planning flow serves as the top-level file for your Quartus II software project. If you do not use the Early Pin Planning flow to create a top-level HDL file, you must analyze and elaborate your design files in the Quartus II software before importing or modifying your pin assignments.

The following sections describe the typical steps involved in the early I/O planning flow to generate a top-level wrapper file in Verilog HDL or VHDL:

- "Create or Import a Megafunction or IP MegaCore Variation from the Pin Planner"
- "Configure Nodes" on page 5–10
- "I/O Analysis for Designs with Pins Only" on page 5–14

Create or Import a Megafunction or IP MegaCore Variation from the Pin Planner

The Pin Planner can interface with the MegaWizard™ Plug-In Manager, allowing you to create or import custom megafunctions and intellectual property (IP) cores. You can add many types of interfaces, including megafunctions such as ALTPLL and ALTDDIO, and IP MegaCore functions such as PCI Compiler, QDR II, and RapidIO. Adding the interface information while planning your I/Os allows you to assign each required pin without manually creating each pin individually in the Pin Planner. Furthermore, you can add and configure your own user ports that are top-level ports in your design.

You can create or customize some megafunction variations from within the Pin Planner. When you do this, some of the pin assignments of the top-level pins of the IP MegaCore function or megafunction get transferred to the Pin Planner. To create a megafunction or IP MegaCore variation from the Pin Planner, perform the following steps:

1. In the Pin Planner, right-click anywhere in the Groups list or All Pins list.
2. On the shortcut (right-click) menu, click **Create/Import Megafunction**. The **Create/Import Megafunction** dialog box appears. You can also open this dialog box from the Edit menu or directly from the Toolbar.
3. To create a new megafunction, select **Create a new custom megafunction** and click **OK**. The **MegaWizard Plug-In Manager** appears.
4. In the list of supported megafunctions and IP MegaCore functions, select the megafunction or IP MegaCore function you want to create and complete the MegaWizard Plug-In Manager pages.
5. After you complete the wizard, a new group, based on the file name you provided, is created and all the I/O names, directions, and I/O standards are listed as members of the group in the Groups list. Any IP megafunction that has a pin planning file (**.ppf**) can be imported to the Pin Planner.

For more information about a particular megafunction, refer to the appropriate megafunction user guide, available on the User Guides literature page of the Altera website.

Besides creating an IP MegaCore function or megafunction variation directly from the Pin Planner, you can import an IP MegaCore function's or megafunction's **.ppf** file into the Pin Planner. This would import a **.ppf** file when you have generated a MegaCore function or megafunction variation and its **.ppf** file.

Or

You have generated the variation in the Pin Planner and want to import its **.ppf** in the Pin Planner for other instances of the same variation. When you import an existing **.ppf** file into the Pin Planner, you have to provide a unique instance name so that in the final top-level HDL, all instances of the same variation remain unique.

To import a megafunction variation to the Pin Planner, perform the following steps:

1. In the Pin Planner, right-click anywhere in the Groups list or All Pins list.
2. On the right-click menu, click **Create/Import Megafunction**. The **Create/Import Megafunction** dialog box appears. You can also open this dialog box from the Edit menu.
3. Select **Import an existing custom megafunction** and click the browse button. Select the Pin Planner File (**.ppf**) that was generated along with your megafunction variation or your IP MegaCore files.
4. In the **Instance name** box, type in an instance name and click **OK**.

To avoid pin name conflicts when there is more than one instance of a megafunction or IP MegaCore function, the instance name is appended to the beginning of each pin name.

When the wizard is complete, a new group based on the file name you provided is created and all the I/Os that are used externally are listed as members of the group.

Configure Nodes

Before you create a top-level design file, you must first configure the user ports, megafunction nodes, and IP MegaCore function nodes created in the Pin Planner for integration with each other and the rest of the design. This step is similar to making connections in a schematic.

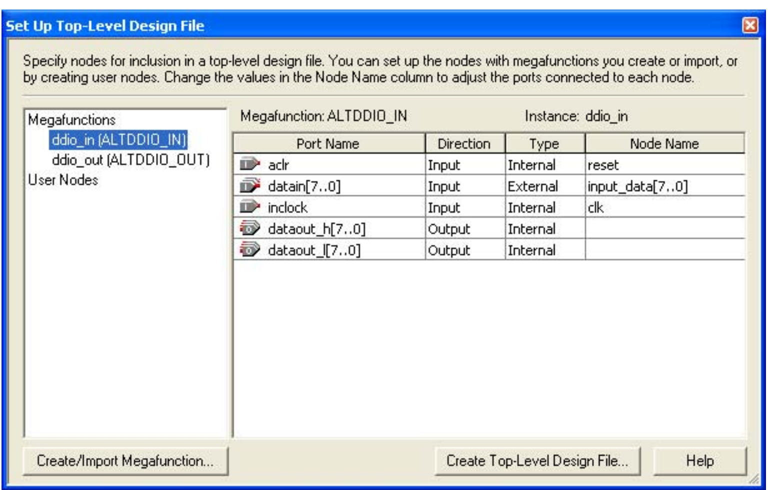
Configure Megafunction Nodes

After creating or importing custom megafunctions or IP MegaCore functions in the Pin Planner, you must configure how they will be connected to each other. You do this by specifying matching node names for selected ports of the megafunctions or IP MegaCore functions.

In this section, ports and port names refer to the generic port names of megafunctions and IP MegaCore functions in the MegaWizard Plug-In Manager. Node names refer to the unique names assigned to ports when the megafunction or IP MegaCore function is created based on the instance name given when the MegaWizard Plug-In Manager is started. By default, node names are the original port names prefixed with *<instance name>*_.

To configure your custom megafunctions and IP MegaCore functions for creating a top-level design file, on the Edit menu of the Package View, click **Set Up Top-Level Design File**. The **Set Up Top-Level Design File** dialog box appears (Figure 5–8).

Figure 5–8. Set Up Top-Level Design File Dialog Box



Click the name of a megafunction or IP MegaCore function in the list on the left. The list on the right contains all the ports for the selected megafunction.

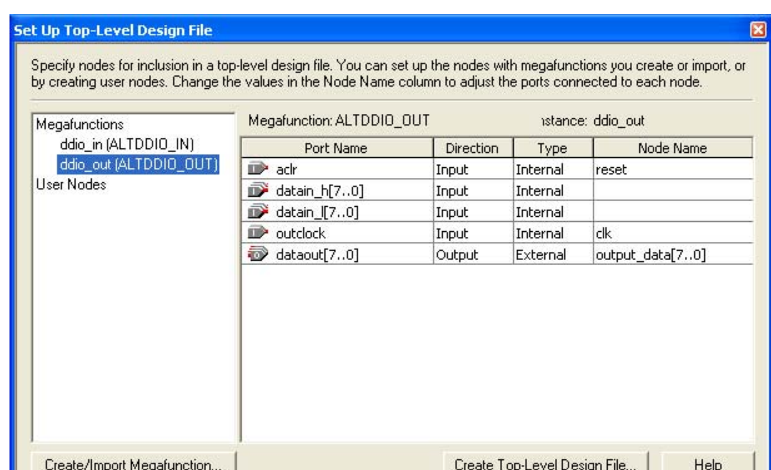
The columns in the **Set Up Top-Level Design File** dialog box provide information about megafunctions created in or imported to the Pin Planner and allow you to make adjustments to connect megafunctions together. The **Direction** column indicates the direction of the port or port group as defined by the megafunction. The direction of a port cannot be changed.

The **Type** column indicates whether a port is available externally to the device. By default, all ports on all megafunctions created through the Pin Planner are of the **External** type, meaning they appear in the Groups list and All Pins list and can be assigned to I/O pins. You can change the port type by double-clicking the cell in the **Type column** for a port and selecting **Internal** or **External** from the list. Any ports on any megafunction connected to a port that has its type changed have their type changed to match automatically. This prevents internal and external megafunction ports from being connected to each other accidentally. Internal ports do not appear in the Groups list or All Pins list. If all the ports of a megafunction are internal, the megafunction does not appear in the Groups list.

The **Node Name** column is used to assign node names or device pins to ports. To connect a port to an existing location, double-click the cell in the **Node Name** column and select an existing node or device pin. To rename the selected port, enter a new node name in the **Node Name** column. This procedure only changes the name as it appears as a group member in the Groups list. To connect ports to each other, enter a node name that matches the node name of the ports of other megafunctions or an existing node.

Figure 5–9 shows an example of the port names of the DDIO_OUT megafunction.

Figure 5–9. Port Names of the DDIO_OUT Megafunction



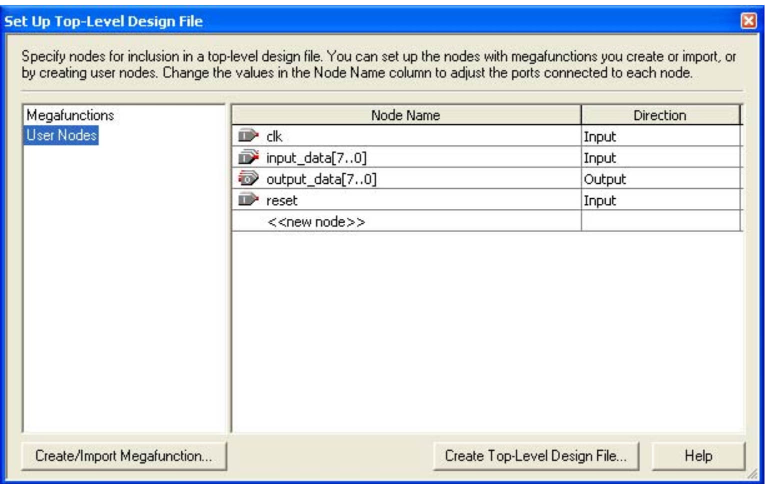
To edit the megafunction or IP MegaCore function you created in the Pin Planner, select it in the Groups list. On the Edit menu, click **Edit Megafunction** to reopen the MegaWizard Plug-In Manager and make changes as necessary. If you make changes to a megafunction, you must import it again and reconfigure its node connections in the **Set Up Top-Level Design File** dialog box.

If you edit the megafunction outside of the Pin Planner, you must re-import its **.ppf** file to the Pin Planner.

Configure User Nodes

If you use the Early I/O Pin Planning flow and start with a top-level file, either complete or partial, and analyze and elaborate that file in Quartus II software, all the top-level pins in that file appear under **User Nodes**, as shown in Figure 5–10. If you do not have any top-level files, you will not see any top-level node names under **User Nodes**.

Figure 5–10. User Nodes

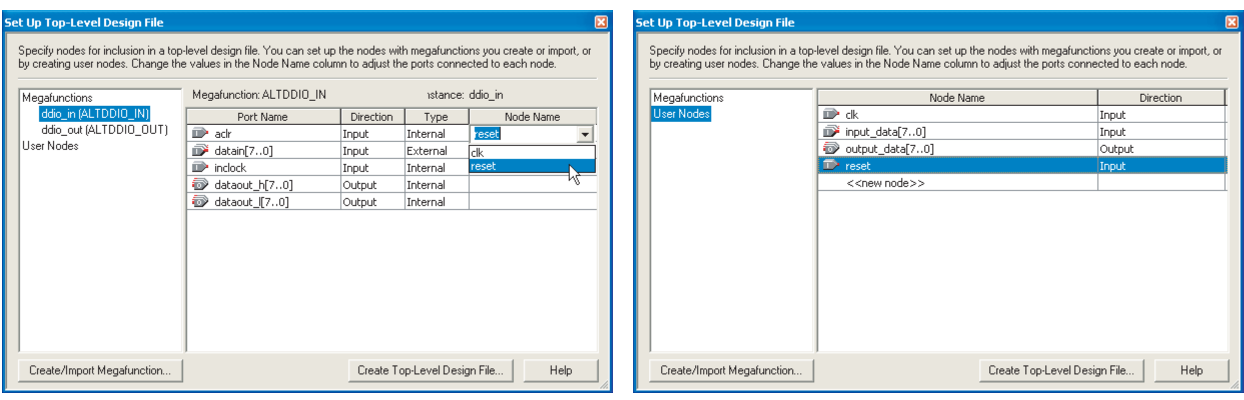


You can add new or additional nodes in the Set-Up Top Level Design File window. To create a new node, double-click in the **<<new node>>** row and enter the new node's name. When you generate the top-level file in HDL, the new or additional nodes appear as ports in your HDL file.

Each user node is associated with a node name and a direction. The direction is input, output, bidirectional, or unknown. If you do not select a direction, the node's direction is unknown. When you enter new node names in this window, the All Pins list and Groups list in the Pin Planner are also updated.

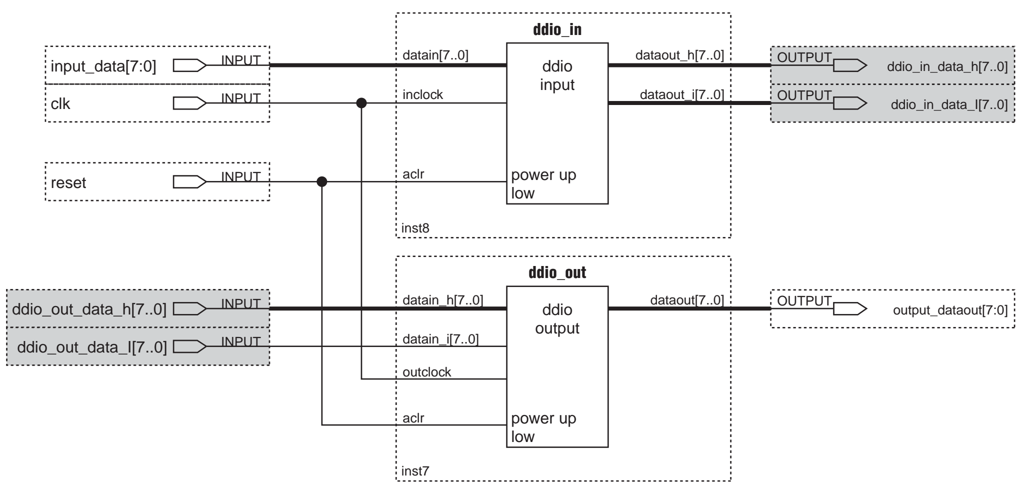
The user nodes are also displayed when you configure node names for megafunctions. For example, to connect a user node named **reset** to a megafunction's reset input port, in the **Node Name** column, select **reset** to make this connection, as shown in Figure 5–11.

Figure 5–11. Connecting a User Node to a Megafunction Port



When the port types and node names for both the input and output megafunctions and user nodes are configured as in Figure 5–8 on page 5–11, Figure 5–9 on page 5–12, and Figure 5–11, they create a circuit similar to the one shown in Figure 5–12. In this way, you can connect megafunctions to each other and to other nodes in the design, improving the thoroughness of an I/O assignment analysis. This is especially useful for clock networks that are typically attached to multiple megafunctions or IP MegaCore functions.

Figure 5–12. Connections between Input and Output Megafunctions and User Nodes *(Note 1)*



Note to Figure 5–12:

(1) Gray pins indicate internal nodes of the design.

I/O Analysis for Designs with Pins Only

You can create a top-level design file after you add or modify user ports, megafunction nodes, or IP MegaCore function nodes in your project with the Pin Planner. Whether the internal logic is complete or not, the top-level design file enables you to validate your I/O assignments and provides a base on which to build the rest of your design.

After you configure the user nodes, megafunctions, and IP MegaCore functions created in the Pin Planner, you must create a top-level design file in an HDL format. Use this file as the basis for the rest of your project and use it to validate the I/O assignments.

To generate a top-level design file, right-click in the Package View and click **Create Top-Level Design File**. You can also generate a top-level file on the File menu by pointing to **Create/Update** and clicking **Create Top-Level Design File From Pin Planner**. The **Create Top-Level Design File** dialog box appears. Enter a name and select an HDL format (Verilog HDL or VHDL). If the file already exists, you can choose to create a backup of the original file.

Example 5–1 shows a sample of an top-level HDL wrapper file as depicted in Figure 5–12.

Example 5–1. HDL Wrapper File Generated with the Early I/O Planning Flow

```
module top
(
   reset,
   input_data,
   clk,
   output_data,
   ddio_in_dataout_h, // Internal
   ddio_in_dataout_l, // Internal
   ddio_out_datain_h, // Internal
   ddio_out_datain_l  // Internal
);

input reset;
input[7:0]input_data;
input clk;
output[7:0]output_data;

output[7:0]ddio_in_dataout_h /* synthesis altera_attribute="-name VIRTUAL_PIN ON" */;
output[7:0]ddio_in_dataout_l /* synthesis altera_attribute="-name VIRTUAL_PIN ON" */;
input[7:0]ddio_out_datain_h /* synthesis altera_attribute="-name VIRTUAL_PIN ON" */;
input[7:0]ddio_out_datain_l /* synthesis altera_attribute="-name VIRTUAL_PIN ON" */;

ddio_in ddio_in_inst
(
   .aclr(reset),
   .datain(input_data),
   .inclock(clk),
   .dataout_h(ddio_in_dataout_h),
   .dataout_l(ddio_in_dataout_l)
);

ddio_out ddio_out_inst
(
   .aclr(reset),
   .datain_h(ddio_out_datain_h),
   .datain_l(ddio_out_datain_l),
   .outclock(clk),
   .dataout(output_data)
);

endmodule
```

The top-level HDL file contains all the top-level pins. When any internal type nodes are not connected to any logic inside the design, they appear in the port list of the top-level file, but are declared as virtual pins. The lines of code in Example 5–2 show examples of virtual pins in the top-level HDL file.

Example 5–2. Examples of Virtual Pins in the Top-Level HDL File

```
output[7:0]input_dataout_h /* synthesis altera_attribute="-name VIRTUAL_PIN ON" */;
output[7:0]input_dataout_l /* synthesis altera_attribute="-name VIRTUAL_PIN ON" */;
input[7:0]output_datain_h /* synthesis altera_attribute="-name VIRTUAL_PIN ON" */;
input[7:0]output_datain_l /* synthesis altera_attribute="-name VIRTUAL_PIN ON" */;
```

The synthesis attribute notifies the Quartus II software that these are virtual pins and that they should be connected to internal logic as the design is completed or finalized. These virtual pins would not be shown in the All Pins list or Groups in the Pin Planner as they are not the actual external ports of the design. The Pin Planner makes virtual pin assignments to internal nodes, so internal nodes are not assigned to device pins during compilation.

The top-level design file must be updated whenever changes are made to the design's top-level ports, including any node changes made in the **Set Up Top-Level Design File** dialog box.

After you generate the top-level file in HDL, you can continue with your design flow to modify or create pin assignments using the Pin Planner.

Importing and Exporting Pin Assignments

After analysis and synthesis of your Quartus II project, you can transfer pin-related assignments between the Quartus II software and other tools by importing and exporting these assignments in the following file formats: Comma Separated Value (**.csv**) file, Quartus II Settings File (**.qsf**), Tcl (**.tcl**), FPGA Xchange (**.fx**) file, and Pin-Out (**.pin**) file (export only).

Spreadsheets and .csv Files

You can transfer pin-related assignments as a **.csv** file. This file consists of a row of column headings followed by rows of comma-separated data. The row of column headings in the exported file is in the same order and format as the columns displayed in the **Pin** category in the Assignment Editor or in the All Pins list in the Pin Planner. Do not modify the row of column headings if you plan to import the **.csv** file later.

To import a **.csv** file into your project, on the Assignment menu, click **Import Assignments** and browse to the file.

You can export pin-related assignments from the Quartus II Pin Planner or the Assignment Editor. To export your pin-related assignments to a **.csv** file, on the Assignment menu, click **Pin Planner** or **Assignment Editor**. For the Pin Planner, make sure the All Pins list is visible. If the list is not visible, on the View menu, click **All Pins List**. For the Assignment Editor, from the **Category** list, select the **Pin** category. Then, to create the **.csv** file, on the File menu, click **Export**.

The All Pins list in the Quartus II Pin Planner, the **Pin** category in the Quartus II Assignment Editor, and the device **.pin** files all display detailed properties about each pin of the device, in addition to the pin name and pin number. The device **.pin** files are available on the Altera website at www.altera.com.

Quartus II Settings Files

When you make pin assignments with the Pin Planner or the Assignment Editor in the Quartus II software, all your pin-related assignments are written to the Quartus II Settings File (**.qsf**). You can also export pin-related assignments to a **.qsf** file. The pin-related assignments are stored as Tcl commands in the **.qsf** file.

To import a **.qsf** file, on the Assignments menu, click **Import Assignments** and browse to the **.qsf** file, or source the **.qsf** file in the Tcl console. To export a **.qsf** file, on the Assignments menu, click **Export Assignments**, type in a file name, and click **OK**. When you export assignments to a **.qsf** file, the **.qsf** file contains all the pin-related assignments along with other Quartus II project-related assignments.

For more information about **.qsf** files, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

Tcl Script

You can export pin-related assignments from the Quartus II Pin Planner or the Assignment Editor. To export pin-related assignments as a Tcl script, on the Assignments menu, click **Pin Planner** or **Assignment Editor**. For the Pin Planner, make sure the All Pins list is visible. If the list is not visible, on the View menu, click **All Pins List**. For the Assignment Editor, select the **Pin** category from the **Category** list. Then, to create the **.tcl** file, on the File menu, click **Export**. In the **Export** dialog box, type a file name, select **Tcl Script File** (***.tcl**), and click **OK**. All pin-related assignments displayed in the All Pins list of the Pin Planner and the spreadsheet of the Assignment Editor are saved as Tcl commands in the Tcl script.

For more information about the All Pins list in the Pin Planner, refer to "Using the Pin Planner" on page 5–20.

To import the pin-related assignments from a Tcl script, source the Tcl script in the Tcl console or run the Tcl script with the `quartus_sh` executable. For example, type the following command at a system command prompt:

```
quartus_sh -t my_pins.tcl ↵
```

For more information about Quartus II scripting support, including examples, refer to the *Tcl Scripting* and *Command-Line Scripting* chapters in volume 2 of the *Quartus II Handbook*.

FPGA Xchange File

An **.fx** file contains device and pin-related information that allows you to transfer information between the Quartus II software and the Mentor Graphics I/O Designer software. To transfer pin information between the I/O Designer and the Quartus II software, use an **.fx** file. The **.fx** file format is imported with the ***.pin** file produced in the Quartus II software and I/O Designer. However, the Quartus II software requires only the **.fx** file to import back from the I/O Designer.

To learn more about the I/O Designer and the DxDesigner interface and support, refer to *Mentor Graphics PCB Tools Support* chapter in volume 2 of the *Quartus II Handbook*.

To import an **.fx** file into the Quartus II software, perform the following steps:

1. On the Assignments menu, click **Import Assignments**.
2. In the **File name** box, click the browse button and click **FPGA Xchange Files (*.fx**) in the **Files of type** list.
3. Select the **.fx** file and click **Open**.

4. Click **OK**.

To generate an **.fx** file in the Quartus II software, perform the following steps:

1. Perform an I/O Assignment Analysis or a full compilation in the Quartus II software.
2. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
3. Select **Board-Level**. The **Board-Level** page appears.
4. Under **Board-level symbol**, in the **Format** list, select **FPGA Xchange**.
5. Set the Output directory to the location where you want to save the file. The default output file path is <*project directory*>**/symbols/fpgaxchange**.
6. Click **OK**.
7. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer**.

The output directory you selected is created when you generate the **.fx** file using the Quartus II software.

.pin File

A **.pin** file is an ASCII text file containing pin location results and other pin information. To generate a **.pin** file for your project, you must successfully perform an I/O Assignment Analysis or full compilation in the Quartus II software. The **.pin** file of your project is written at the <*project directory*> level.

Use the **.pin** file to transfer your project's pin information into third-party PCB tools for board development. Figure 5–13 shows an example **.pin** file, and Table 5–1 describes the columns in a **.pin** file.

Figure 5–13. Example of a .pin File

Pin Name/Usage	Location	Dir.	I/O Standard	Voltage	I/O Bank	User Assignment
VCCA_PLL1	9	power		1.5V		
clk	10	input	LVTTL		1	N

Table 5–1. .pin File Header Description

Column Name	Description
Pin Name/Usage	The name of a design pin, ground, or power
Location	The pin number of the location on the device package
Dir	The direction of the pin
I/O Standard	The name of the I/O standard to which the pin is configured
Voltage	The voltage level that is required to be connected to this pin
I/O Bank	The I/O bank number that the pin belongs to
User Assignment	Y or N indicating if the location assignment for the design pin was user assigned (Y) or assigned by the Fitter (N)

For more information about Pin Name/Usage, refer to the Device Pin-Out tables for the targeted device, available on the Altera website at www.altera.com.

Creating Pin-Related Assignments

A pin-related assignment is any assignment applied to a top-level pin. For example, a pin location assignment is a pin-related assignment that assigns a top-level port or node to a pin number (location) on the targeted device. Other examples of pin-related assignments include assigning an I/O standard, assigning current drive strength, or assigning slew rates to a pin.

The recommended approach for making pin-related assignments is to have all the top-level pins in the Pin Planner and create pin-related assignments for these pins.

If you do not have complete information for all the top-level pins, you can reserve certain device pins to temporarily represent your top-level design I/O pins until the I/O pins are defined in your design files. Reserved pins are intended for future use but do not currently perform a function in your design. Reserved pins require a unique pin name and pin location. Using reserved pins as place holders for future design pins increases the accuracy of the I/O assignment analysis.

The Quartus II software offers many tools and features for creating reserved pins and other pin-related assignments (Table 5–2). Each tool and feature is described in more detail in the following sections.

Table 5–2. Overview of Quartus II Tools and Features to Create Pin-Related Assignments (Part 1 of 2)

Feature	Overview
Pin Planner	■ Make pin location assignments to one or more node names by dragging and dropping unassigned pins into the Package View ■ Edit pin location assignments for one or more node names by dragging and dropping groups of pins within the Package View ■ Visually analyze pin resources in the Package View ■ Display I/O banks and VREF groups ■ View the function of package pins using the pin legend ■ Make correct pin location decisions by referring to the Pad View window ■ Create, import, and edit megafunctions and IP MegaCore functions for early I/O planning ■ Generate a top-level wrapper file without design files based on early I/O assignments ■ Configure board trace models of selected pins for use in “board-aware” signal integrity reports generated with the **Enable Advanced I/O Timing** option
Assignment Editor	■ Create and edit all types of pin-related assignments ■ Create and edit multiple assignments simultaneously with the Edit bar ■ Create pin assignments efficiently by viewing the different font styles used to display assigned and unassigned node names, as well as occupied and available pin locations ■ Provides user-selectable information about each pin, including the pad number, t_{CO} requirement, and t_H requirement
Tcl	■ Create any pin-related assignments for multiple pins ■ Store and reapply all pin-related assignments with Tcl scripts ■ Make assignments from the command line

Table 5–2. Overview of Quartus II Tools and Features to Create Pin-Related Assignments (Part 2 of 2)

Feature	Overview
Chip Planner or Timing Closure Floorplan	■ Create and change pin locations by dragging and dropping pins into the floorplan ■ Make correct pin location decisions by referring to the pad ID number and spacing ■ Display I/O banks, VREF groups, and differential pin pairing information
Synthesis Attributes	■ Embed pin-related assignments using attributes in the design files to pass assignments to the Quartus II software

Using the Pin Planner

The Pin Planner is the main interface for creating and editing pin-related assignments. Use the Pin Planner Package View to make pin location and other assignments using a device package view instead of pin numbers. With the Pin Planner, you can identify I/O banks, VREF groups, and differential pin pairings to help you through the I/O planning process.

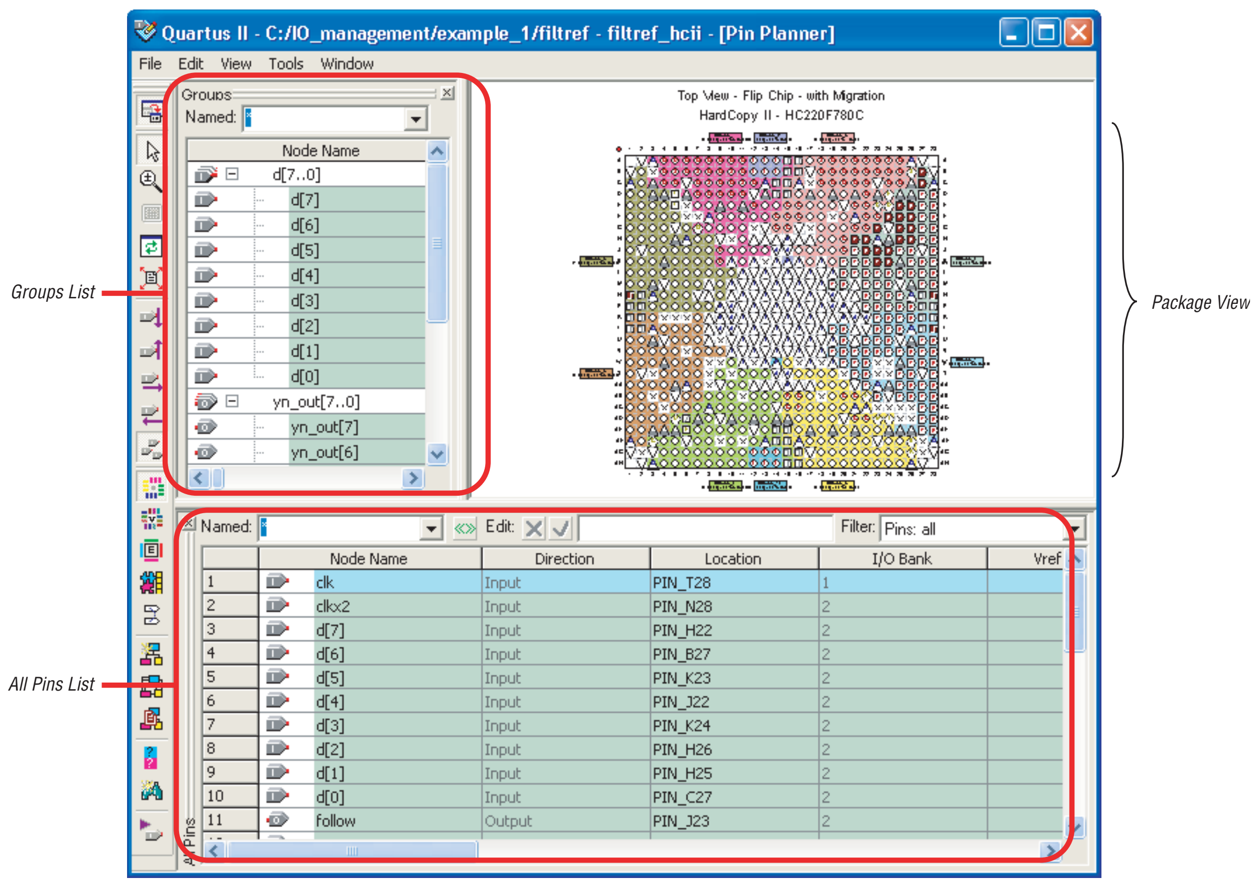
When planning your I/Os, it can be cumbersome to try to correlate pin numbers with their relative location on the package and their pin properties. The Pin Planner provides an intuitive graphical representation of the targeted device, also known as the Package View, that makes it easy to plan your I/Os, create reserved pins, and make pin location assignments. When deciding on a pin location, use the Pin Planner to gather information about available resources, as well as the functionality of each individual pin, I/O bank, and VREF group. You can assign locations to design pins by dragging and dropping each pin into the Package View.

Maintaining good signal integrity (SI) requires that you follow pad distance and pin placement rules. Complementing the Pin Planner is the Pad View window, which displays the pads in order around the silicon die.

The Pin Planner includes the following sections (refer to Figure 5–14 through Figure 5–19):

■ "Groups List" on page 5–21
■ "All Pins List" on page 5–24
■ "Pad View Window" on page 5–26
■ "Package View" on page 5–28
■ "Pin Migration View" on page 5–30
■ "Using the Pin Finder to Find Compatible Pin Locations" on page 5–33
■ "Creating Reserved Pin Assignments" on page 5–34
■ "Creating Pin Location Assignments" on page 5–35
■ "Changing Pin Locations" on page 5–42
■ "Show I/O Banks" on page 5–44
■ "Show VREF Groups" on page 5–45
■ "Show Edges" on page 5–47
■ "Show DQ/DQS Pins" on page 5–48
■ "Displaying and Accepting Fitter Placements" on page 5–49

Figure 5–14. Pin Planner



The Pin Planner feature supports cross-probing that allows you to select a pin in one view while simultaneously highlighting the pin in all of the different views. For example, if you select a pin in the Package View of the Pin Planner, the corresponding pad in the Pad View window is highlighted. If the pin has an assigned node name, the node name in the All Pins list and the Groups list is highlighted.

The Pin Planner and the Assignment Editor get their I/O Bank colors from Timing Closure Floorplan colors, which you can customize by performing the following steps:

1. On Tools menu, click **Options**. The **Options** dialog box appears.
2. In the **Category** list, under **Timing Closure Floorplan**, select **Colors**. The **Colors** page appears.
3. Make your color selections and click **OK**.

Groups List

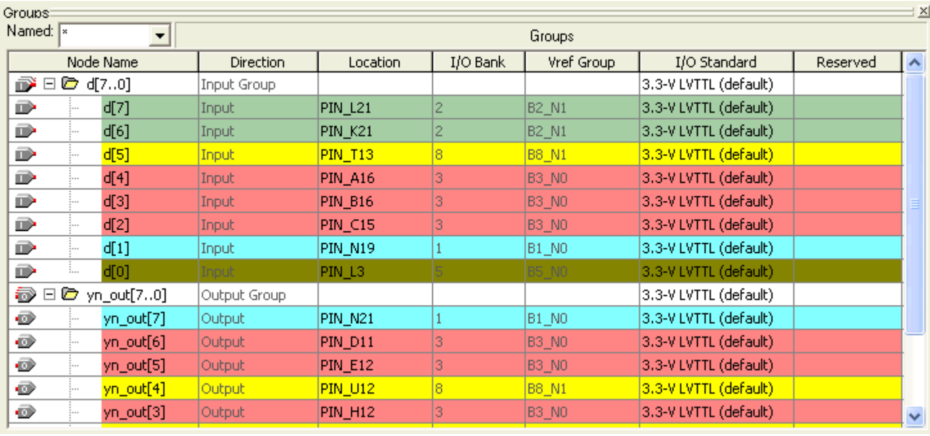
The Groups list displays all of the buses from the top-level pins of your design and all the assignment groups in your project (Figure 5–15). Filter the group names displayed by typing in a wild card filter into the **Named** list. The Groups list allows you to create your own custom groups of pins and make location assignments to groups by dragging them into the Package View of the Pin Planner.

In the Groups list, all members of an assignment group are displayed, regardless of whether the member is a pin or an internal node.

The background color of pin locations in the Groups list easily identifies which pins belong to which I/O banks. The colors match the I/O bank colors in the Package View when **Show I/O Banks** is enabled. You can turn off the colors in both the Groups list and the All Pins list. On the Tools menu, click **Options**. In the **Category** list, select **Pin Planner**, and turn off **Show I/O bank color in lists**.

You can create and organize custom groups and group members in the **Assignment Groups** dialog box or directly in the Groups list in the Pin Planner. To open the **Assignment Groups** dialog box, on the Assignments menu, click **Assignment (Time) Groups**.

Figure 5–15. Groups List



Groups
Named: *
Groups

Node Name	Direction	Location	I/O Bank	Vref Group	I/O Standard	Reserved
d[7..0]	Input Group				3.3-V LVTTL (default)	
d[7]	Input	PIN_L21	2	B2_N1	3.3-V LVTTL (default)	
d[6]	Input	PIN_K21	2	B2_N1	3.3-V LVTTL (default)	
d[5]	Input	PIN_T13	8	B8_N1	3.3-V LVTTL (default)	
d[4]	Input	PIN_A16	3	B3_N0	3.3-V LVTTL (default)	
d[3]	Input	PIN_B16	3	B3_N0	3.3-V LVTTL (default)	
d[2]	Input	PIN_C15	3	B3_N0	3.3-V LVTTL (default)	
d[1]	Input	PIN_N19	1	B1_N0	3.3-V LVTTL (default)	
d[0]	Input	PIN_L3	5	B5_N0	3.3-V LVTTL (default)	
yn_out[7..0]	Output Group				3.3-V LVTTL (default)	
yn_out[7]	Output	PIN_N21	1	B1_N0	3.3-V LVTTL (default)	
yn_out[6]	Output	PIN_D11	3	B3_N0	3.3-V LVTTL (default)	
yn_out[5]	Output	PIN_E12	3	B3_N0	3.3-V LVTTL (default)	
yn_out[4]	Output	PIN_U12	8	B8_N1	3.3-V LVTTL (default)	
yn_out[3]	Output	PIN_H12	3	B3_N0	3.3-V LVTTL (default)	

To add a new group to the Groups list without opening the **Assignment Groups** dialog box, perform the following steps:

1. In the Groups list, in the **Node Name** column, double-click **<<new node>>**.
2. Type the group name.
3. Press **Enter**. The **Add Members** dialog box appears.
4. Type node names, wild cards, and assignment groups in the **Members** box, or browse to and select the node names from the **Node Finder** dialog box.
5. Click **OK**.

For more information about using Assignment Groups, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

You can also create a new group by selecting one or more node names within the Groups list or All Pins list. Right-click one of the selected node names, and on the right-click menu, click **Add to Group**.

As you plan your I/O placement, you might decide to add and remove members from a group.

© March 2009 Altera Corporation

To add a member to a custom group in the Groups list without opening the **Assignment Groups** dialog box, perform the following steps:

1. Right-click a group name in the Groups list and click **Add Members**.
2. Type in the name of the member or click the browse button to select one or more nodes from the **Node Finder** dialog box.

To remove a member from a group in the Groups list, perform the following steps:

1. Expand the group from which you want to remove a member.
2. Select one or more members that you want to remove.
3. Right-click the selected members, point to **Edit** and click **Delete**.

The Groups list provides many columns, some for information purposes and others to make assignments. You can edit the following columns only:

- **Node Name**
- **Location**
- **I/O Standard**
- **Reserved**
- **Enable**

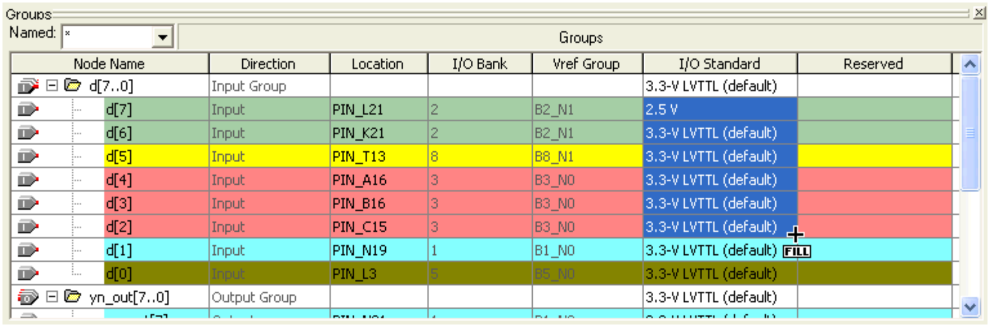
Make changes to any of the values in these columns to adjust pin-related assignments. Other columns provide useful information during I/O planning, including the I/O Bank number, the VREF group, and the direction. To show or hide a column, right-click the column and click **Customize Columns**. You can also reorder and sort the columns from this menu.

If an assignment group contains pins with different directions, the direction of the assignment group is a bidir group.

You can edit the columns in the Groups list in the same manner as a spreadsheet. You can copy and paste the **Location**, **I/O Standard**, and **Reserved** assignments to other rows in the list within the same column. You can also use **Auto Fill** to copy these assignments to other rows quickly.

To automatically fill a block of rows, set the desired assignment in one row and select the assignment's cell. Place the cursor over the lower right-hand corner of the cell until it changes to a cross with the word **FILL** (Figure 5–16). Click and drag up or down the column to select which cells to fill. When all the desired cells are selected, release the mouse button. The selected assignment is copied to all of the selected cells.

Figure 5–16. Auto Fill the Groups List

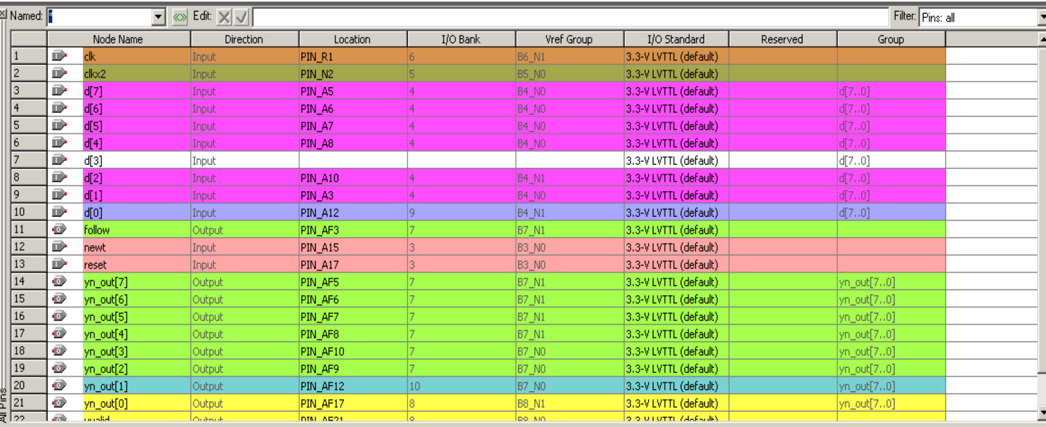


All Pins List

The All Pins list displays all of the pins in your design, including user-created pins (Figure 5–17). The All Pins list does not display buses; instead, it displays each individual pin of the bus. The background color of pin locations in the All Pins list easily identifies which pins belong to which I/O banks. The colors match the I/O bank colors in the Package View when **Show I/O Banks** is turned on. You can turn off the colors in both the All Pins list and the Groups list. On the Tools menu, click **Options**. In the **Category** list, select **Pin Planner**, and turn off **Show I/O bank color in lists**.

You must perform Analysis and Elaboration successfully to display pins in your design in the All Pins list. Individual user-reserved pins and nodes with pin-related assignments are always shown in the All Pins list.

Figure 5–17. All Pins List



You can filter the list of pins in the All Pins list based on their node names by typing in a portion of the pin name in combination with wild card characters in the **Named** list. You can also filter the list of pins in the All Pins list based on the pins' attributes by selecting from the **Filter** list.

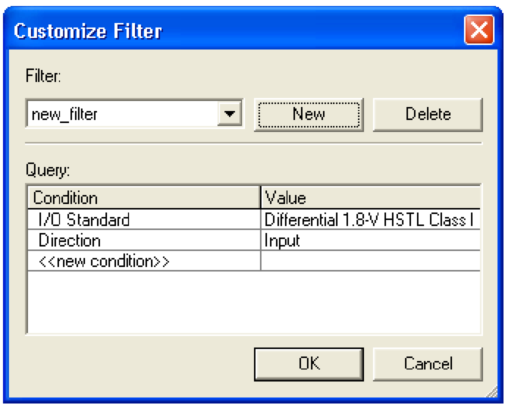
To create your own custom filter in the **Filter** list, specify a set of conditions from the following list:

- Assigned or unassigned

- Current strength
- Direction
- Edge location
- I/O Bank location
- I/O Standard
- VREF Group

To create a new filter in the All Pins list, in the **Filter** list in the All Pins list, select **<<new filter>>**. The **Customize Filter** dialog box appears (Figure 5–18).

Figure 5–18. Customize Filter Dialog Box



To create a custom filter for the All Pins list, perform the following steps:

1. In the **Customize Filter** dialog box, click **New**. The **New Filter** dialog box appears.
2. Enter the name of your custom filter in the **Filter name** text box.
3. You can base your new custom filter on existing filters by selecting from the **Based on Filter** list. If you do not want to base your custom filter on any other filter, select **Pins: all** from the **Based on Filter** list.
4. Click **OK**.
5. Add as many conditions as you require to the **Query** list. To add a condition, double-click **<<new condition>>** and select a condition from the **Condition** list. Select a value for the condition by double-clicking the cell next to your selected condition under the **Value** column.

To remove a condition from your filter, right-click the condition in the **Query** list and select **Delete**.

After specifying your conditions, the pins meeting the specified conditions are the only pins shown in the All Pins list. If the set of conditions contains a condition with more than one value, the pins displayed must meet at least one of the values for that multiple-value condition.

To edit an existing custom filter, select **<<new filter>>** from the **Filter** list in the All Pins list. In the **Customize Filter** dialog box, select the custom filter you want to edit from the **Filter** list and add and remove conditions to the **Query** list.

Pins generated from a compilation or from a bus group are not editable. All other user-created pins are editable.

The All Pins list provides many columns, some for information purposes and others to make assignments. To show or hide a column, right-click the column heading and select **Customize Columns**. In addition, you can reorder and sort the columns from this menu.

You can edit the columns in the Groups list in the same manner as a spreadsheet. You can copy and paste assignments to other rows in the list within the same column. You can also use Auto Fill to quickly copy these assignments to other rows. To automatically fill a block of rows, set the desired assignment in one row and select the assignment's cell. Place the cursor over the lower right-hand corner of the cell until it changes to a cross with the word **FILL**, as shown with the Groups list in Figure 5–16 on page 5–24. Click and drag up or down the column to select which cells to fill. When all the desired cells are selected, release the mouse button. The selected assignment is copied to all the selected cells.

Pad View Window

To maintain good signal integrity in designs, use the Pad View window to guide your pin placement decisions. Each device family is accompanied with pin placement rules, including pad spacing between various pin types.

For more information about pin placement rules, refer to the appropriate device handbook.

Edit or make pin assignments in the Pad View window by dragging and dropping a design pin into an available pad location.

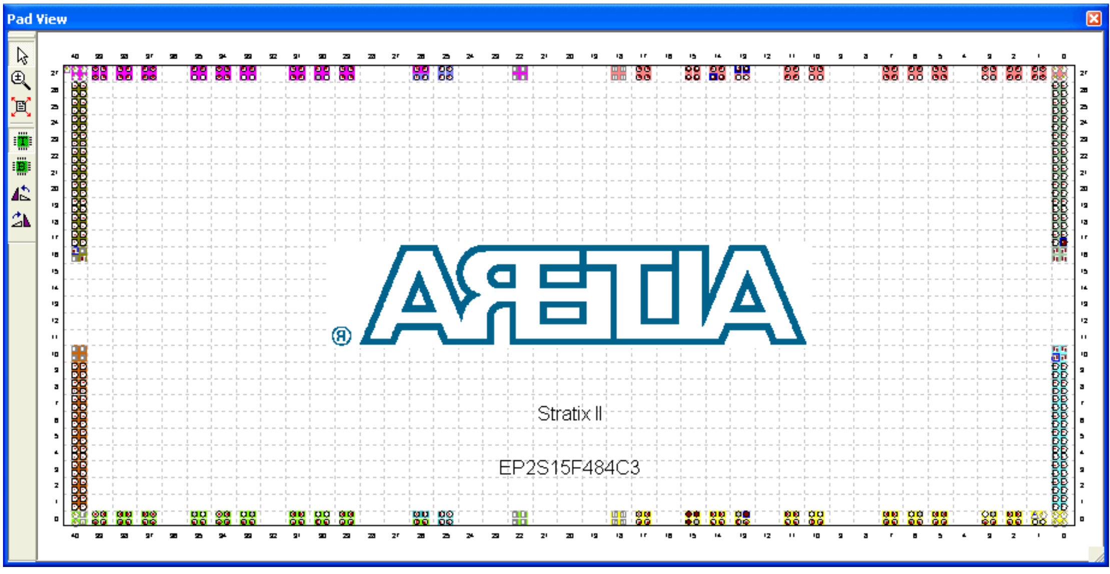
When you drag and drop a design pin into an available pad location, the corresponding pin number of the pad is assigned to the design pin. To assign a pad number to the design pin, perform the following steps:

1. On the Tools menu, click **Options**. The **Options** dialog box appears.
2. Click **Pin Planner** and turn on **Create pad assignment in the Pad View window**.

The column and row numbering around the Pad View window helps identify the pad row or pad column where each pad is located. This is useful when the pin placement guidelines for your targeted device refer to pad rows and columns.

Since the Pad View window is a view of the I/O ring of the silicon within the package, flip chip packages appear inverted. Notice the reversed ALTERA logo in Figure 5–19. To understand the correlation between the package pins and the pads on the silicon die, the Pad View window and Package View are closely integrated. When a pad is selected, the corresponding pin in the Package View is highlighted. Similarly, when a pin is selected in the Package View, the corresponding pad is highlighted in the Pad View window.

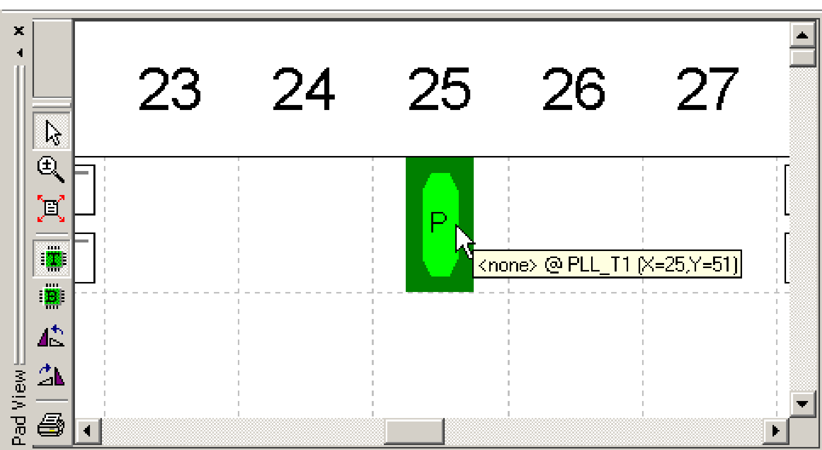
Figure 5–19. Pad View Window of a Stratix II Flip Chip Device



In the Quartus II software version 9.0 and later, you can see the PLLs and DLLs of your device in the Pad View and make location assignments to them. The flexibility of making or deleting location assignments directly to the PLLs and DLLs from the Pin Planner provides you finer control over your design.

PLLs and DLLs appear as an oval in a rectangle, as shown in Figure 5–20. This is the same way they are shown in the Chip Planner in the Quartus II software.

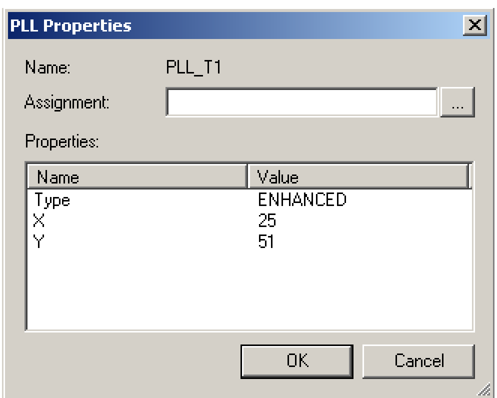
Figure 5–20. PLLs and DLLs in Pad View of Pin Planner



In the Pad view, global PLLs are shown with the letter "P," HSSI PLLs are shown with the letter "H," and DLLs are shown with the letter "D" in the oval symbol. The Tooltip of the PLLs or DLLs shows the name, location assignment, and XY locations.

To view the properties window of the PLLs and DLLs shown in Figure 5–20, double-click the P, H, or D symbol in the Pad View. The PLL Properties window (Figure 5–21) shows the PLL name, location assignment, and its properties, which includes the PLL type and its X and Y location values. You can create or delete the location assignment from this window.

Figure 5–21. PLL Properties

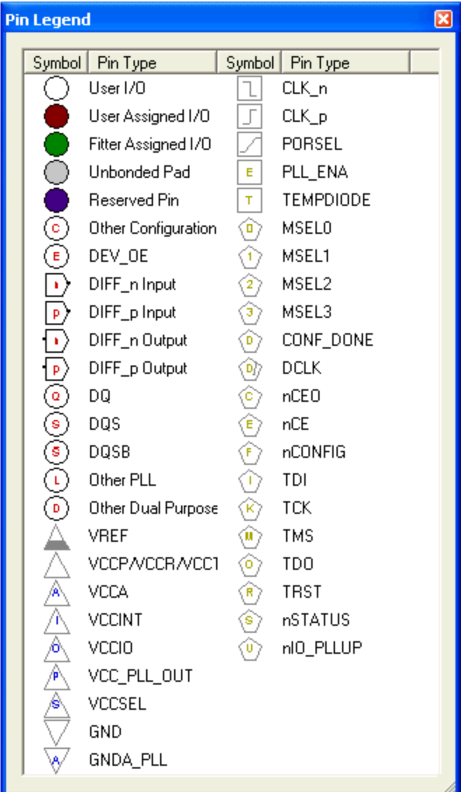


Package View

The Package View in the Pin Planner uses annotated pin symbols in different shapes and colors as visual representations of pins of the actual package (Figure 5–14 on page 5–21). The Package View eliminates the need to cross-reference each pin number with its physical location on the package described in the device package datasheet. When making pin location assignments in the Package View, switch between the different views to help you decide on a pin location. The different views in the Package View include I/O banks, VREF groups, Edges, DQ/DQS pins, and differential pin pairs. For more information about the different views in the Package View, refer to the section in this chapter about the specific view you want to use. The sections are listed on page 5–20.

The Pin Legend window provides a quick reference to the meanings of the pin symbol shapes, notations, and colors in the Package View. To view the Pin Legend window, on the View menu, click **Pin Legend Window** (Figure 5–22). You can also open the Pin Legend window from the Pin Planner toolbar or from the right-click menu in the Package View.

Figure 5–22. Pin Legend Window



Planning your FPGA I/O assignments with your board design is necessary. If your FPGA device is oriented differently than in the Package View and Pad View windows of the Pin Planner, rotate the Package View.

To rotate the Package View, on the View menu, point to **Show** and click **Rotate Left 90°** or **Rotate Right 90°** until your FPGA is shown in the desired orientation in the Package View. The red dot in the Package View indicates the location of the first pin. For example, the red circle identifies where Pin A1 is located on a BGA package and where Pin 1 is located on a TQFP package.

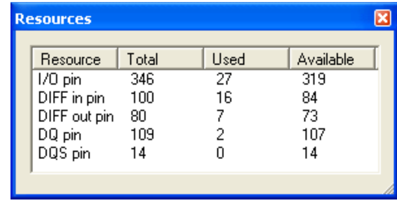
You can also print the Package View with the pin names and pin types visible (Figure 5–23). To show the pin name (if available) or pin type for each pin in the Package View, on the View menu, click **Show Node Names** and **Show Pin Types**.

Figure 5–23. Package View with Show Node Names and Show Pin Types

To view pin resource usage, on the View menu, click **Resources Window**. The Resources window appears (Figure 5–24).

For more detailed information about resources, view the **Resource** section of the Compilation Report.

Figure 5–24. Resources Window



Resources

Resource	Total	Used	Available
I/O pin	346	27	319
DIFF in pin	100	16	84
DIFF out pin	80	7	73
DQ pin	109	2	107
DQS pin	14	0	14

If a HardCopy® II companion device is selected, the Pin Planner shows the Package View for the Stratix II device. To ensure correct pin migration between Stratix II and HardCopy II devices, run the **I/O Assignment Analysis** command or the Fitter.

Pin Migration View

If a migration device is selected, the Pin Planner shows only pins that are available for migration. Selecting a migration device allows you to either vertically migrate to a different density, while using the same package, or to migrate between packages with different densities and ball counts.

The Pin Migration View in the Pin Planner shows the pins that change function in a migration device if you select one or more migration devices for your project. You can see changes for a pin by checking the **Show migration differences** box in the migration view. On the View menu, click **Pin Migration View** to open the Pin Migration View window. A pin is also highlighted in other views of the Pin Planner when you select any pin in the Pin Migration View.

The migration view provides detailed information about the pins that are affected in the migrated device. To select migration devices, perform the following steps:

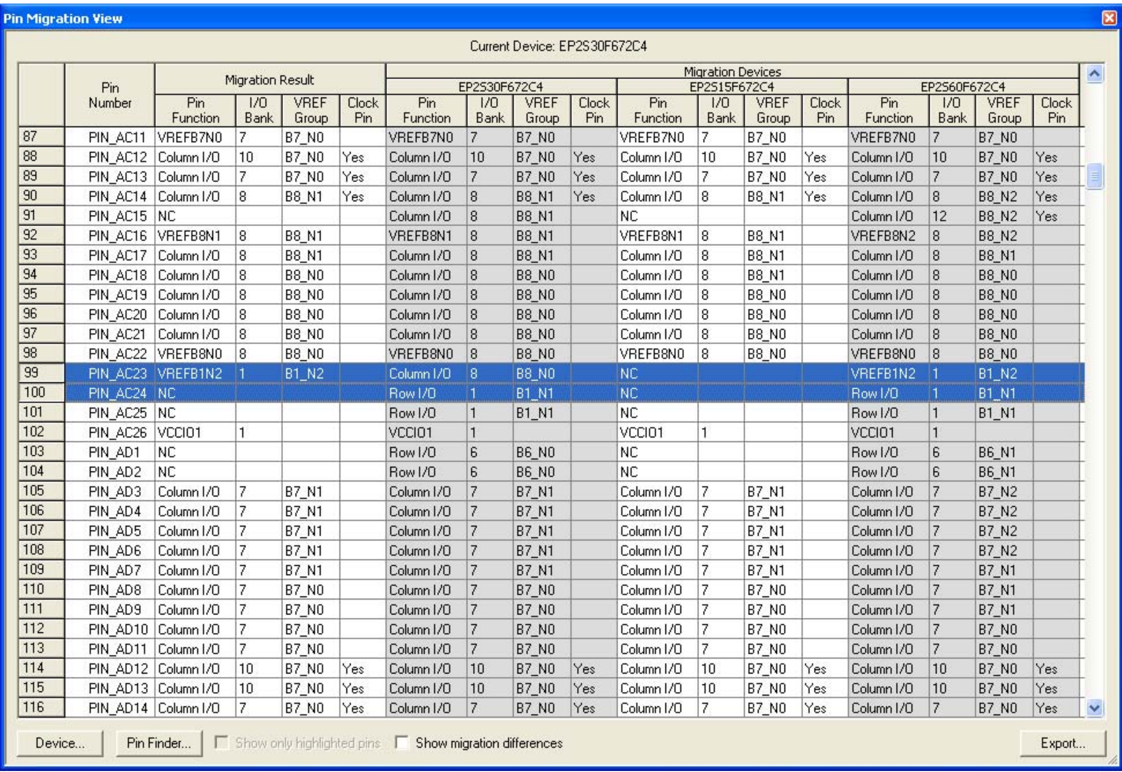
1. On the Assignments menu, click **Device**. The **Settings** dialog box appears.
2. Click **Migration Devices**. The **Migration Devices** dialog box appears.
3. Make your migration device selections and click **OK**.

The Pin Migration View helps you identify the difference in pins that can exist between migration devices. For example, in Figure 5–25, the highlighted pin `AC24` existed in the original EP2S30 device selected, but does not exist in one of the migration devices. Therefore, the migration result is a No Connect (NC). If you select your migration devices after you have successfully compiled a design and these migration devices have certain differences, an error occurs if you try to recompile your design.

For example, if you have a pin assignment on a pin in the original design, that pin might not be present in a migration device. If you do not select migration devices early in the design cycle, you might get a successful fit initially. However, if you select a migration device or devices later in your design cycle for which the pin assignment cannot be honored because the pin does not exist in the migration device, an error occurs when you try to recompile. Therefore, Altera recommends you choose the supported migration devices early in the design planning process. When you select migration devices early in the design process, only the pins that exist in all migration devices are available in the Pin Planner and the Assignment Editor.

Additional differences can exist between migration devices, as shown in Figure 5–25.

Figure 5–25. Pin Migration View


Pin Migration View

Current Device: EP2S30F672C4

	Pin Number	Migration Result				Migration Devices EP2S30F672C4				EP2S15F672C4				EP2S60F672C4			
		Pin Function	I/O Bank	VREF Group	Clock Pin	Pin Function	I/O Bank	VREF Group	Clock Pin	Pin Function	I/O Bank	VREF Group	Clock Pin	Pin Function	I/O Bank	VREF Group	Clock Pin
87	PIN_AC11	VREFB7N0	7	B7_N0		VREFB7N0	7	B7_N0		VREFB7N0	7	B7_N0		VREFB7N0	7	B7_N0	
88	PIN_AC12	Column I/O	10	B7_N0	Yes	Column I/O	10	B7_N0	Yes	Column I/O	10	B7_N0	Yes	Column I/O	10	B7_N0	Yes
89	PIN_AC13	Column I/O	7	B7_N0	Yes	Column I/O	7	B7_N0	Yes	Column I/O	7	B7_N0	Yes	Column I/O	7	B7_N0	Yes
90	PIN_AC14	Column I/O	8	B8_N1	Yes	Column I/O	8	B8_N1	Yes	Column I/O	8	B8_N1	Yes	Column I/O	8	B8_N2	Yes
91	PIN_AC15	NC				Column I/O	8	B8_N1		NC				Column I/O	12	B8_N2	Yes
92	PIN_AC16	VREFB8N1	8	B8_N1		VREFB8N1	8	B8_N1		VREFB8N1	8	B8_N1		VREFB8N2	8	B8_N2	
93	PIN_AC17	Column I/O	8	B8_N1		Column I/O	8	B8_N1		Column I/O	8	B8_N1		Column I/O	8	B8_N1	
94	PIN_AC18	Column I/O	8	B8_N0		Column I/O	8	B8_N0		Column I/O	8	B8_N1		Column I/O	8	B8_N0	
95	PIN_AC19	Column I/O	8	B8_N0		Column I/O	8	B8_N0		Column I/O	8	B8_N0		Column I/O	8	B8_N0	
96	PIN_AC20	Column I/O	8	B8_N0		Column I/O	8	B8_N0		Column I/O	8	B8_N0		Column I/O	8	B8_N0	
97	PIN_AC21	Column I/O	8	B8_N0		Column I/O	8	B8_N0		Column I/O	8	B8_N0		Column I/O	8	B8_N0	
98	PIN_AC22	VREFB8N0	8	B8_N0		VREFB8N0	8	B8_N0		VREFB8N0	8	B8_N0		VREFB8N0	8	B8_N0	
99	PIN_AC23	VREFB1N2	1	B1_N2		Column I/O	8	B8_N0		NC				VREFB1N2	1	B1_N2	
100	PIN_AC24	NC				Row I/O	1	B1_N1		NC				Row I/O	1	B1_N1	
101	PIN_AC25	NC				Row I/O	1	B1_N1		NC				Row I/O	1	B1_N1	
102	PIN_AC26	VCCIO1	1			VCCIO1	1			VCCIO1	1			VCCIO1	1		
103	PIN_AD1	NC				Row I/O	6	B6_N0		NC				Row I/O	6	B6_N1	
104	PIN_AD2	NC				Row I/O	6	B6_N0		NC				Row I/O	6	B6_N1	
105	PIN_AD3	Column I/O	7	B7_N1		Column I/O	7	B7_N1		Column I/O	7	B7_N1		Column I/O	7	B7_N2	
106	PIN_AD4	Column I/O	7	B7_N1		Column I/O	7	B7_N1		Column I/O	7	B7_N1		Column I/O	7	B7_N2	
107	PIN_AD5	Column I/O	7	B7_N1		Column I/O	7	B7_N1		Column I/O	7	B7_N1		Column I/O	7	B7_N2	
108	PIN_AD6	Column I/O	7	B7_N1		Column I/O	7	B7_N1		Column I/O	7	B7_N1		Column I/O	7	B7_N2	
109	PIN_AD7	Column I/O	7	B7_N1		Column I/O	7	B7_N1		Column I/O	7	B7_N1		Column I/O	7	B7_N1	
110	PIN_AD8	Column I/O	7	B7_N0		Column I/O	7	B7_N0		Column I/O	7	B7_N0		Column I/O	7	B7_N1	
111	PIN_AD9	Column I/O	7	B7_N0		Column I/O	7	B7_N0		Column I/O	7	B7_N0		Column I/O	7	B7_N1	
112	PIN_AD10	Column I/O	7	B7_N0		Column I/O	7	B7_N0		Column I/O	7	B7_N0		Column I/O	7	B7_N0	
113	PIN_AD11	Column I/O	7	B7_N0		Column I/O	7	B7_N0		Column I/O	7	B7_N0		Column I/O	7	B7_N0	
114	PIN_AD12	Column I/O	10	B7_N0	Yes	Column I/O	10	B7_N0	Yes	Column I/O	10	B7_N0	Yes	Column I/O	10	B7_N0	Yes
115	PIN_AD13	Column I/O	10	B7_N0	Yes	Column I/O	10	B7_N0	Yes	Column I/O	10	B7_N0	Yes	Column I/O	10	B7_N0	Yes
116	PIN_AD14	Column I/O	7	B7_N0	Yes	Column I/O	7	B7_N0	Yes	Column I/O	7	B7_N0	Yes	Column I/O	7	B7_N0	Yes

Device... Pin Finder... Show only highlighted pins Show migration differences Export...

Notice that for PIN_AC23, the Migration Result for Pin Function is not an NC but a voltage reference VREFB1N2. This is because it is an NC in one of the migration devices, but a VREFB1N2 in the other migration device. In this type of result, VREF standards have a higher priority than an NC and the result is VREFB1N2. You might not be making use of that pin for a port connection in your design, but you must tie the VREF standard for I/O standards that require it on the actual board for the migration device.

To see the migration differences in the migration result, in the Pin Migration View window, turn on **Show migration differences**. This option is useful if you are vertically migrating your device to a package with less or more I/O counts than your original device. You can also export the migration view spreadsheet as a ***.csv** file. To do this, in the Pin Migration View window, select a file name and click **Export**.

You can see additional I/O features in the Pin Migration View window in the Clocks, DQ/DQS, Differential, and Configuration columns with the Quartus II software 8.0 and later. To hide or view additional columns, right-click in the Pin Migration View window, point to **Show Columns**, and select the appropriate column.

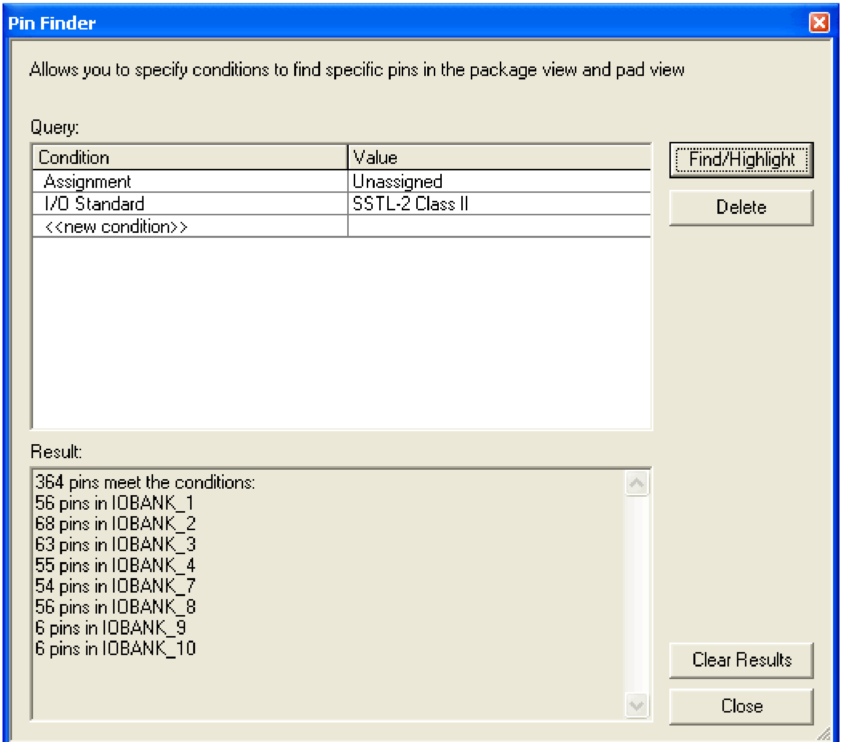
For more information about migration, refer to *AN90: SameFrame Pin-Out Design for FineLine BGA Packages*. For more information about designing for HardCopy II devices, refer to the *Quartus II Support for HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook*.

Using the Pin Finder to Find Compatible Pin Locations

As FPGA pin-counts and I/O capabilities continue to increase, it becomes more difficult to understand the capabilities of each I/O pin and to correctly assign your design I/Os. To help you address this problem, the Pin Planner highlights all pins that match the list of conditions that you enter. To enter your conditions, perform the following steps with the Pin Planner open:

1. On the View menu, click **Pin Finder**. The Pin Finder window appears (Figure 5–26).

Figure 5–26. Pin Finder Window



2. In the Pin Finder window, create a list of conditions in the **Query** list.

   To add a condition to the **Query** list, double-click **<<new condition>>** and select a condition from the list. Double-click the cell next to the new condition and select a desired value. For example, if you want to highlight all available pins that support the SSTL-2 Class II I/O standard, create an assignment condition and an I/O standard condition as shown in Figure 5–26.

   If the same condition type occurs more than once in the list, the Pin Finder searches for results that match any of its specified values. If you add more than one condition type, the Pin Finder searches for results that match all of the specified conditions.

3. In the Pin Finder window, click **Find/Highlight**. All of the pins that meet the specified conditions are highlighted in the Package View and Pad View windows.

   At the same time, the **Results** list in the Pin Finder window displays a summary of the number of pins in each I/O Bank that meet the specified conditions.

SSN Visualization View

In the Quartus II software version 9.0 and later, you can view the simultaneous switching noise (SSN) visualization in the Pin Planner by right-clicking in the Package View and selecting **Show SSN Analyzer results**. The SSN Analyzer is a new tool introduced in the Quartus II software version 9.0 to estimate SSN on the input and output pins of your FPGA device. Refer to the Quartus II Help to find out the devices supported for SSN analysis. The integration of the SSN Analyzer and Pin Planner in the Quartus II software allows you to perform SSN analysis while planning your I/O pins.

For more information about the SSN Analyzer, refer to the *Simultaneous Switching Noise (SSN) Analysis and Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Creating Reserved Pin Assignments

You can make reserved pin assignments to act as place holders for future design pins in the Package View or in the All Pins list. To create a reserved pin in the Package View, right-click an available pin, point to **Reserve** and click one of the available configurations.

When you reserve a pin from the Package View, the name of the reserved pin is set to `user_reserve_`<*number*> by default and the pin symbol is filled with a dark purple color. The number increments by 1 for each additional reserved pin.

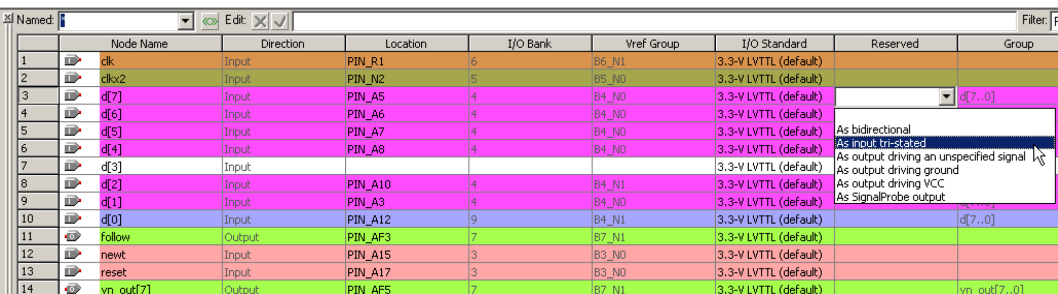
Alternately, you can reserve a pin in the All Pins list by performing the following steps:

1. Type the pin name into an empty cell in the **Node Name** column. The pin name must not already exist in your design.
2. Select a pin configuration from the **Reserved** list (Figure 5–27).

   The following configurations are available:

   - **As bidirectional**
   - **As input tri-stated**
   - **As output driving an unspecified signal**
   - **As output driving ground**
   - **As output driving VCC**
   - **As SignalProbe output**

Figure 5–27. Reserving a Pin in the All Pins List in the Pin Planner



Release reserved pins by selecting the blank entry from the **Reserved** column.

The **Direction** column is a read-only column and changes direction depending on the reserved selection.

Creating Pin Location Assignments

You can create pin location assignments for one or more pins with the following methods:

- Assigning a location for unassigned pins
- Assigning a location for differential pins
- Assigning an unassigned pin to a pin location

You can disable or prevent any of these assignments using the **Enable** column in either the Groups list or the All Pins list. The **Enable** column is a special column that allows you to disable only the location assignment for a selected pin. Change the value of the cell in the **Enable** column for a selected pin from **Yes** to **No** by double-clicking the cell and selecting **No** from the list. A disabled pin only prevents location assignments when signals are assigned using drag-and-drop, as described below. You can still make assignments directly in the **Location** columns in both the Groups list and All Pins list. To enable the location assignment again, change the **Enable** column back to **Yes**.

Assigning Locations for Unassigned Pins

To assign locations for all of your design pins, perform the following steps:

1. On the Edit menu, select an assignment direction.

   You can assign several pins simultaneously by choosing an assignment direction (Table 5–3). When assigning an entire bus, assignments are made in order from the most significant bit (MSB) to the least significant bit (LSB).

Table 5–3. Multiple Pins

Assignment	Pin Group
Assign Down	From the selected group of unassigned design pins, assign the pins downwards starting from the selected pin.
Assign Up	From the selected group of unassigned design pins, assign the pins upwards starting from the selected pin.
Assign Right	From the selected group of unassigned design pins, assign the pins from left to right starting from the selected pin.
Assign Left	From the selected group of unassigned design pins, assign the pins from right to left starting from the selected pin.
Assign One by One	Select a pin location for each of the design pins selected from the **Pins: unassigned** list in the **Filter** list.

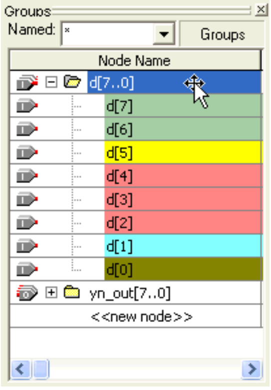
If there is an unassignable location in the path of the selected assignment direction, pins are assigned as far in the assignment direction as possible. Assign the rest of the pins in a separate location.

2. In the **Filter** list, select **Pins: unassigned**.

3. In the All Pins list, select one or more unassigned node names, or in the Groups list, select one or more buses.

   You can click on multiple node names using the control and shift keys. When you click on a pin or bus in the All Pins list or Groups list, the node name is highlighted and a crossing arrow displays above the cursor. Drag the selected cells into the Package View (Figure 5–28).

Figure 5–28. Drag Node Name in the Groups List



4. Drag and drop the selected pins or buses from the All Pins list or Groups list to a location in the Package View.

Before you drag-and-drop your pins, you can optionally use the Pin Finder to locate pin locations that support your selected pins. When creating a query in the Pin Finder, add an Assignment condition and set it to **Unassigned**.

If you do not use the Pin Finder, you can drop pins directly into any of the following locations in the Pin Planner Package View:

- Available user I/O pin
- I/O Bank
- VREF Group
- Edge

On the View menu, you can display either I/O banks, VREF groups, or edges by going to the Show submenu and toggling between **Show I/O Banks**, **Show VREF Groups**, and **Show Edges**. You can also toggle between these views from the Pin Planner toolbar or from the right-click menu in the Package View.

Available single-ended user I/O pins are represented by empty circles in the Package View. The letter inside the circle provides information about the user I/O pin. Negative and positive differential pins are shaped like hexagons and contain the letters "n" and "p", respectively. For a complete listing of I/O pin shapes, notations, and colors, from the View menu, toolbar, or Package View right-click menu, open the Pin Legend window.

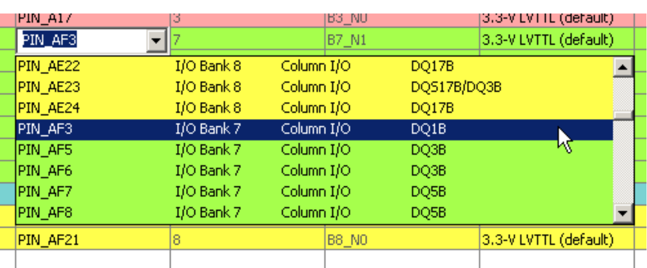
In the Package View, I/O banks are displayed as rectangles labeled `IOBANK_`<*number*> (Figure 5–35 on page 5–44). In each I/O bank, there are one or more VREF groups. VREF groups are displayed as rectangles labeled `VREFGROUP_B`<*I/O Bank number*>`_N`<*index*> (Figure 5–37 on page 5–46).

Edge locations are displayed as rectangles labeled `EDGE_`<*direction*>. To make an edge assignment, drag and drop pins into one of the four edges, `EDGE_TOP`, `EDGE_BOTTOM`, `EDGE_LEFT`, or `EDGE_RIGHT`.

You can drag-and-drop pins from the **Node Finder** dialog box or from the Block Diagram/Schematic File into the Package View.

Click on the **New Node** button ( ) in the All Pins list to jump directly to the new node row without scrolling all the way down. When you click on the **Location Assignment** cell in the All Pins list, a drop-down combo box with all the assignable pins is opened (Figure 5–29).

Figure 5–29. Combo Box in the Pin Planner



The combo box shown in Figure 5–29 displays each row in a color matching its I/O bank color. In the combo box, each pin location row displays the location assignment column, its I/O bank column, and its special function column. While making assignments for a node, if you double click the **Edit** field in the All Pins list, a pull-down box appears with all the possible values for the assignment.Choose a value and click the green check button ( ) to make the assignment or click the red cross button ( ) to cancel the assignments change.

Creating Exclusive Group Assignments

In the Quartus II software version 9.0 and later, you can create exclusive groups comprised of pins by using the following assignment:

```
set_instance_assignment -name "EXCLUSIVE_IO_GROUP" -to pin
```

When you create exclusive I/O group(s) in your FPGA design and use the Quartus II software to map the signals onto device pins, the Fitter does not place the I/O pins belonging to one exclusive group in an I/O bank if the pins belong to another exclusive I/O group. To understand this, consider an example in which you have a set of signals assigned exclusively to a group called `group_a`, and another set of signals assigned to `group_b`. In both exclusive groups you might have pins with different I/O standards. When you create these groups, the Quartus II software maps the pins of both groups in such a way that they are placed in different I/O banks.

Assigning a Location for Differential Pins

You can assign the top-level pins of your design as differential pins in Altera devices using the Pin Planner. Before you assign the top-level pins of your design as differential pins in the Pin Planner, it is important to understand the process for creating differential pins in your design. If your design has top-level pins that are single ended, you can use the Pin Planner to assign the required differential standard to that pin. The Quartus II software automatically creates the negative pin of the differential standard. For example, if you have a top-level pin defined as `lvds_in` in your design, the Quartus II software creates a `lvds_in(n)` pin when you assign a differential standard to `lvds_in`, as shown in Figure 5–30.

Figure 5–30. Creating a Differential Pin Pair in the Pin Planner

Named: * Edit: lvds_in Filter: Pins: all

	Node Name	Differential Pair	I/O Standard	Direction
5	input_data[4]		3.3-V LVTTL (default)	Input
6	input_data[3]		3.3-V LVTTL (default)	Input
7	input_data[2]		3.3-V LVTTL (default)	Input
8	input_data[1]		3.3-V LVTTL (default)	Input
9	input_data[0]		3.3-V LVTTL (default)	Input
10	lvds_in	lvds_in(n)	LVDS	Input
11	output_data[7]		3.3-V LVTTL (default)	Output
12	output_data[6]		3.3-V LVTTL (default)	Output
13	output_data[5]		3.3-V LVTTL (default)	Output
14	output_data[4]		3.3-V LVTTL (default)	Output
15	output_data[3]		3.3-V LVTTL (default)	Output
16	output_data[2]		3.3-V LVTTL (default)	Output
17	output_data[1]		3.3-V LVTTL (default)	Output
18	output_data[0]		3.3-V LVTTL (default)	Output
19	reset		3.3-V LVTTL (default)	Input

All Pins

For Stratix III and Cyclone® III FPGA families, you can use low-level I/O differential primitives to define both positive and negative pins of a differential pair in your design's HDL code.

For more information about supported I/O primitives and details about their assignments, refer to the *Designing with Low-Level Primitives Users Guide* or the Quartus II Help.

When you use low-level I/O primitives to define various pin-related assignments for I/Os, the assignments are honored after you perform full compilation. These assignments are not shown in the Pin Planner after analysis and synthesis. After a full compilation, you can populate these assignments in the Pin Planner by back-annotating pin assignments. To back annotate your pin assignments, on the Assignments menu, click **Back-Annotate Assignments**.

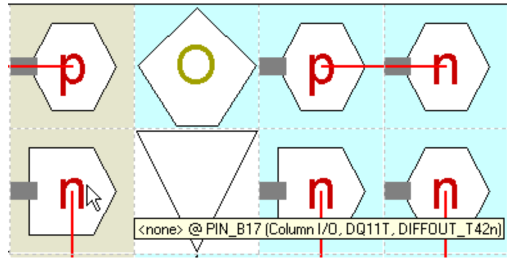
To identify and assign differential pins using the Pin Planner, perform the following steps:

1. On the View menu, click **Show Differential Pin Pair Connections**.

   A red line connects the positive and negative pins of the differential pin pairing. The positive and negative pins are labeled in the Package View with the letters "p" and "n", respectively (Figure 5–31).

2. Use the tool tips to identify LVDS-compatible pin locations by holding the mouse pointer over a differential pin in the Package View (Figure 5–31).

Figure 5–31. Tool Tip of a Differential Pin



The tool tip shows the design pin name and pin number, as well as its general and special functions.

For more information about the general and special functions of pins displayed by the tool tip, refer to the device pin-out tables for your device family at www.altera.com.

3. From the All Pins list or Groups list, drag-and-drop the selected pin of the differential pair to a differential pin location in the Package View.

Pay attention when you drag the positive or negative pin of the differential pin pair to the Package View. Connect the positive pin to the “p” pin on the device and negative pin to the “n” pin on the device. An error appears in the Quartus II software if you do not connect the differential pins properly.

Optionally, before you drag-and-drop your pins, you can use the Pin Finder to locate pin locations that support your selected pins. When creating a query in the Pin Finder, add an assignment condition set to **Unassigned** and an I/O standard condition set to your differential I/O standard.

The Quartus II software recognizes the negative pin as part of the differential pin pair assignment. The location assignment for the negative pin pair is written to the **.qsf** file. However, the assignment I/O standard is not entered in the **.qsf** file for the negative pin of the differential pair.

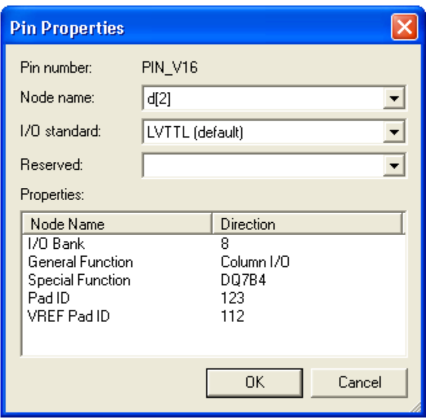
If you have a single-ended clock that feeds a PLL, assign the pin only to the positive clock pin of a differential pair in the targeted device. Single-ended pins that feed a PLL and are assigned to the negative clock pin in the targeted device cause the design to not fit.

Assigning an Unassigned Pin to a Pin Location

Perform the following steps to select a pin location and assign a design pin to that location:

1. In the Package View, select an available pin location.
2. On the View menu, click **Pin Properties.** The **Pin Properties** dialog box appears (Figure 5–32).

Figure 5–32. Pin Properties Dialog Box



You can use the **Pin Properties** dialog box to create pin location and I/O standard assignments. The **Pin Properties** dialog box also displays the properties of the pin location, including the pad ID (Table 5–4). The pad ID is important information when following pin spacing guidelines. Adjacent pin numbers do not always represent adjacent pads on the die. Use the Pad View window to help correlate pad location and the distance between your user I/O pins and VREF pins.

3. Select a pin from the **Node Name** list.
4. To assign or change the I/O standard, select an I/O standard from the **I/O standard** list.
5. Click **OK**.

For more information about pin placement, refer to the appropriate device handbook.

Table 5–4 provides a description of each field in the **Pin Properties** dialog box.

Table 5–4. Pin Properties

Pin Property	Description
Pin Number	Pin number used in the package *(1)*
Node Name	Node name assigned to the pin location
I/O Standard	I/O standard assigned to the pin name and location
Reserved	If reserved, determines how to reserve this pin
I/O Bank	I/O bank number of the pin
General Function	General function of the pin (row and column I/O, dedicated clock pin V_{CC}, GND)
Special Function	Special function of the pin (LVDS, PLL)
Pad ID	Pad number connected to pin
VREF Pad ID	The pad ID for the VREF pin used for voltage referenced I/O standards

Note to Table 5–4:

(1) For more information about how pin numbers are derived, refer to the device pin-out on the Altera website, www.altera.com.

© March 2009 Altera Corporation

You can also open the **Pin Properties** dialog box by double-clicking on a pin in the Package View of the Pin Planner, or by right-clicking the pin in the Package View of the Pin Planner, and clicking **Pin Properties**.

Changing the Slew Rate and Current Drive Strength in Pin Planner

Current strength is an important property of an I/O pin. It affects the integrity of the signal going out of the device. You can set the current strengths as part of the I/O planning flow in the Pin Planner. You can also change the current strength after the compilation of your design. In the Pin Planner's All Pins list, you can view or edit the current strengths for each of the output and bidirectional pins in the **Current Strength** column. If the **Current Strength** column is not visible in All Pins list, right-click in the All Pins list and use the **Customize Columns** dialog box to add the **Current Strength** column. The settings you make in Pin Planner are honored during Live I/O Check, I/O Assignment analysis, and full compilation.

To make changes to the current drive strength after compilation has finished, you can perform engineering change orders (ECOs). To perform ECOs, use the Resource Property Editor (RPE) tool in the Quartus II software. Performing ECO compilation does not recompile the whole design, but compiles only the changes.

For more information about ECOs, refer to the Quartus II Help or the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

The slew rate is another important property of an I/O pin that affects the outgoing signal integrity of the device. Slew rate options are not supported for all device families. To learn more about the supported families for slew rates, refer to the specific device handbooks. You can set the slew rate as part of the I/O planning flow in the Pin Planner. You can also change the slew rate setting after compiling your design.

In the Pin Planner's **All Pins** list, you can view or edit the slew rate for each of the output and bidirectional pins in the **Slew Rate** column. If the **Slew Rate** column is not visible in the All Pins list, right-click in the All Pins list and use the **Customize Columns** dialog box to add the **Slew Rate** column. The settings you make in Pin Planner are honored during Live I/O Check, I/O Assignment Analysis, and full compilation. To change the slew rate after compilation has finished, you can perform an ECO by using the RPE tool in the Quartus II software. Performing ECO compilation does not perform place-and-route on the whole design, but only on the changes.

For more information about ECOs, refer to the Quartus II Help or the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

You can also set the current drive strength and slew rate settings by using the following Tcl assignments in the **.qsf** file:

```
set_instance_assignment -name CURRENT_STRENGTH_NEW 8MA -to e[0]
set_instance_assignment -name SLEW_RATE 2 -to e[0]
```

For more information about the effect of I/O settings on Signal Integrity on the board for Stratix III devices, refer to *AN 476: Impact of I/O Settings on Signal Integrity in Stratix III Devices*.

Error Checking Capability

The Pin Planner has basic pin placement checking capability, preventing pin placements that violate the fitting rules. The following checks are performed by the Pin Planner as you make pin-related assignments:

- An I/O bank or VREF group is an unassignable location if there are no available pins in the I/O bank or VREF group.
- The negative pin of a differential pair is unassignable if the positive pin of the differential pair has been assigned with a node name with a differential I/O standard.
- Dedicated input pins (for example, dedicated clock pins) are an unassignable location if you attempt to assign an output or bidirectional node name.
- Pin locations that do not support the I/O standard assigned to the selected node name become unassignable.
- All nodes in the same VREF group must have the same VREF voltage. Apply this only to HSTL- and SSTL-type I/O standards.

To perform a more comprehensive check on your pin placements, perform I/O assignment analysis.

For more information about assignment analysis, refer to "Using I/O Assignment Analysis to Validate Pin Assignments" on page 5–59.

To display live information about the warnings and errors in your pin-related assignments, enable the live I/O check feature in the Quartus II software. For more information about the live I/O check feature, refer to the section "Using the Live I/O Check Feature to Validate Pin Assignments" on page 5–57.

After creating a pin location, the **Location**, **I/O Bank**, and **VREF Group** columns are populated in both the All Pins list and the Groups list. In the Package View, the occupied pins are filled with a dark brown color.

Changing Pin Locations

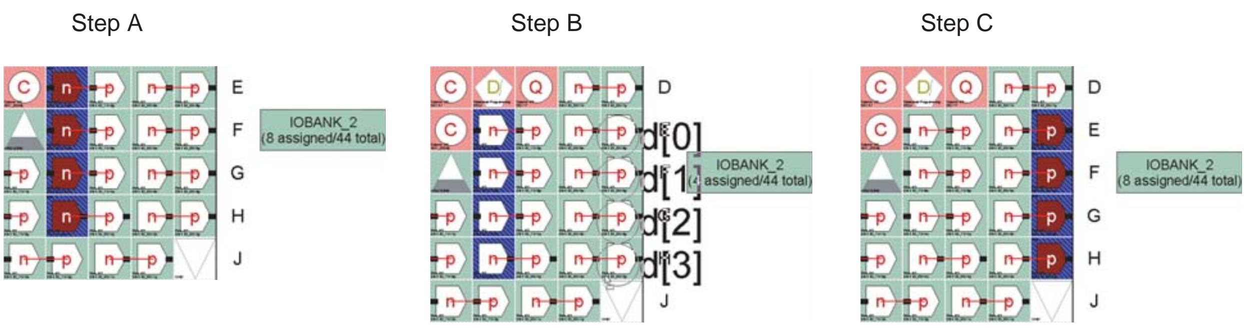
The Pin Planner allows you to change the location of multiple pins simultaneously. To change pin locations, select one or more pins in the Package View or Pad View window, and drag the pins to a new location.

You can change pin locations more quickly and easily if you understand which user I/O pins are available and where they are physically on the device. For example, in the Package View, you can move a column of pins closer to the edge of the device for easier PCB routing (Figure 5–33). In this example, you are moving multiple I/O pins to the area closest to the edge of the I/O bank. To change pin locations, perform the following steps:

1. In the Package View, select multiple pins by holding down the left mouse button and dragging over the pins you want to move (Figure 5–33, step A).
2. Drag the group of pins to the placement area (Figure 5–33, step B).
3. Drop the pins into the area closest to the edge of the I/O bank (Figure 5–33, step C).

© March 2009 Altera Corporation

Figure 5–33. Changing the Locations for a Group of Pins

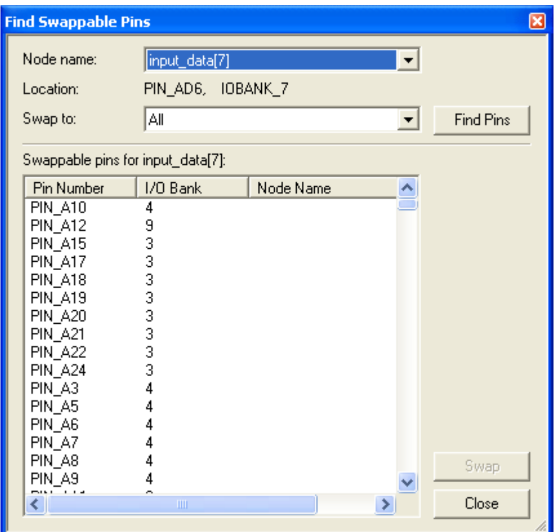


Swapping Pin Locations

In Quartus II software version 8.0 and later, you can use the Pin Planner to swap pin locations in your design. The Pin Swapping feature can help you move one or more pins from one location to another for last minute PCB layout or design changes after you have made most of your pin assignments.

To use the Pin Swapping feature, right click in the Pin Planner's Package View and click **Find Swappable Pins**. The **Find Swappable Pins** dialog box appears (Figure 5–34). As shown in Figure 5–34, the **Node Name** field drop down list shows all the nodes in your design, a **Location** field, which gives information of any existing location assignment of the node selected in the **Node Name** field, a **Swap To** filter that you can use to narrow down the searchable pins in a particular I/O bank or VREF group, and a **Swap** button.

Figure 5–34. Find Swappable Pins Dialogue Box in the Pin Planner



In the **Find Swappable Pins** dialog box, **Node name** enables you to choose one of the top-level pins or nodes of your design and shows you all the possible swappable locations in your selected device when you click **Find Pins**. The list of swappable pins returned to you depends on what filter you use in the **Swap to** filter to find the swappable pins. For example, if you select **All**, a list of all the swappable locations for

that node will resemble the pins shown in Figure 5–34. The returned list contains all the assigned and unassigned pin locations. If you want to swap or move a signal name to a certain pin location in the list, select the pin location and click **Swap**. The **Swap** button is only enabled and highlighted when you select a pin location in the list. When you click the **Swap** button, the node in the "Node Name" is assigned to that pin location. If the selected pin location is assigned to another node, the original node in the **Node name** list and your selected node get swapped.

The Pin Swap feature requires the **Live Check** option to be turned on; therefore, you might get warnings or errors if any pin placement rules are violated during pin swapping. The warning and errors are printed in the Messages window. You must run I/O Assignment Analysis after performing any pin swapping.

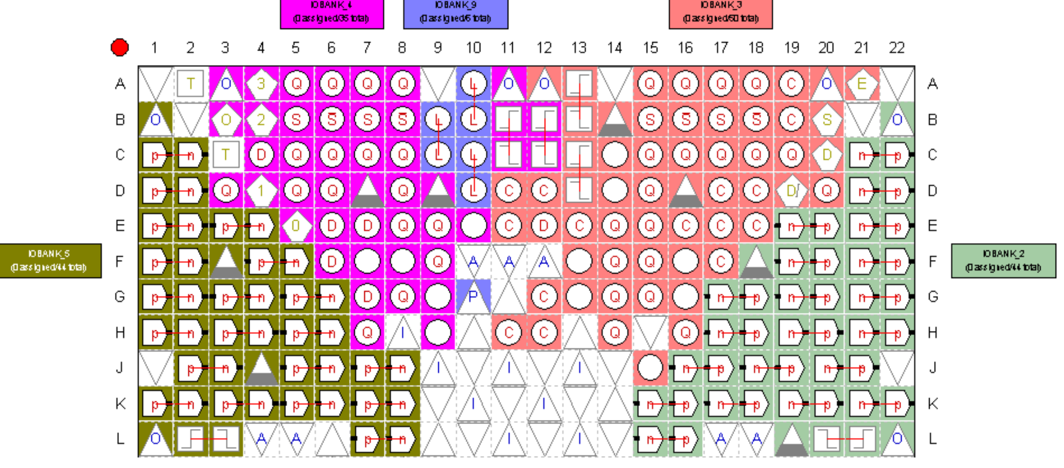
When you select the pin location, the pin location is highlighted in the Package View of Pin Planner. This helps you locate the physical location where you intend to swap.

Show I/O Banks

When you turn on Show I/O Banks in the View menu, in the Show submenu, or on the right-click menu in the Package View, the Package View groups I/O pins that share the same `VCCIO` pin using different colors (Figure 5–35). When planning your I/O pins, it is important to guide your pin placement decisions by placing pins with compatible I/O standards into the same I/O bank. For example, you cannot place an `LVTTL` pin with an I/O standard of LVTTL in the same bank as another pin with an I/O standard of 1.5 V HSTL Class I.

For more information about compatible I/O standards, refer to the appropriate device handbook.

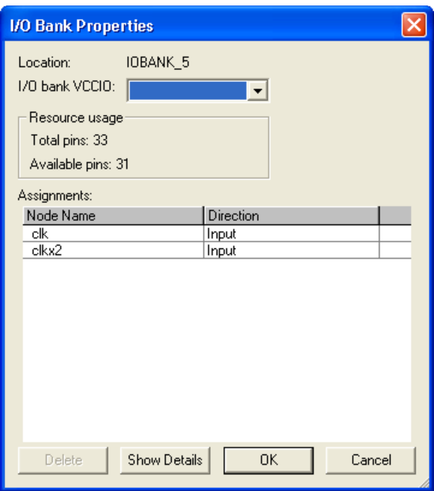
Figure 5–35. Package View with I/O Banks



When you turn on **Show I/O Banks**, the Package View allows you to view the properties of each I/O bank. Select an I/O bank in the Package View. On the View menu, click **I/O Bank Properties.** The **I/O Bank Properties** dialog box appears (Figure 5–36). The **I/O Bank Properties** dialog box lists all node names assigned to that I/O bank.

To view all node names that are assigned within the I/O bank, click **Show Details** in the **I/O Bank Properties** dialog box. You can also assign the V_{CCIO} for the I/O bank by selecting a voltage from the **I/O bank VCCIO** list.

Figure 5–36. I/O Bank Properties



Under **Resource Usage**, the total number of pins in the I/O banks is displayed, including assignable and unassignable pins, as well as the total number of available assignable pins. Adjust the intensity of colors of the I/O banks in the Package View by performing the following steps:

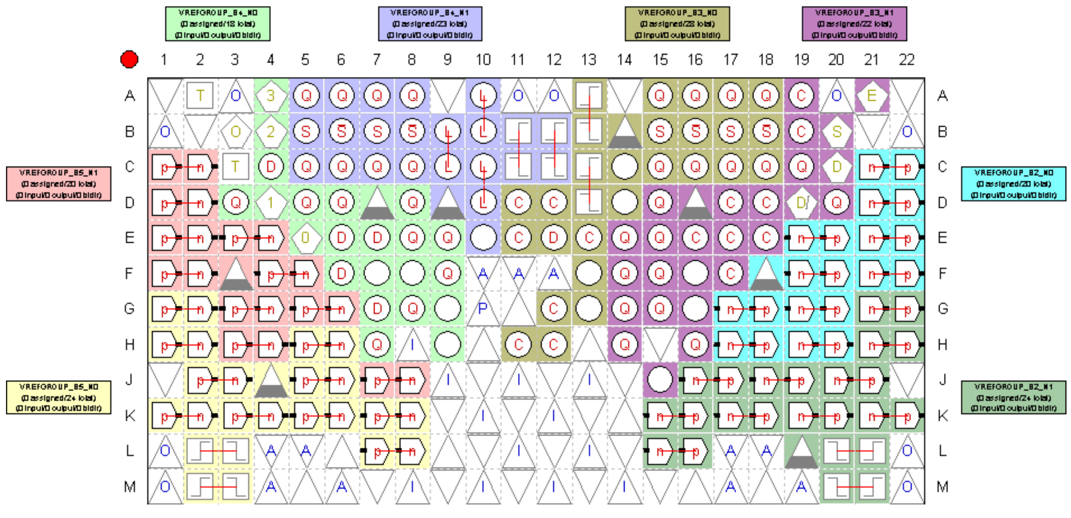
1. On the Tools menu, click **Options**. The **Options** dialog box appears.
2. In the **Category** list, select **Pin Planner**. The **Pin Planner** page appears.
3. Under **I/O banks color setting**, adjust the **I/O bank color intensity** using the slide bar.
4. Click **OK**.

Show VREF Groups

You can use different colors to indicate different groups of I/O pins sharing the same `VCCIO` and `VREF` pins in the package view (Figure 5–37). When planning your I/O pins, it is important to place pins with compatible voltage-referenced I/O standards in the same I/O bank. To guide your pin placement decisions by placing compatible I/O standards requiring `VREF` pins into the same VREF group, on the View menu, point to **Show** and click **Show VREF Groups**. For example, pins with I/O standards SSTL-18 Class II and 1.8V-HSTL Class II are compatible and can be placed into the same VREF group. It is also important to be aware of the number and direction of pins within a VREF group for simultaneous switching noise (SSN) analysis.

For more information about compatible I/O standards, refer to the appropriate device handbook.

Figure 5–37. Package View with VREF Groups

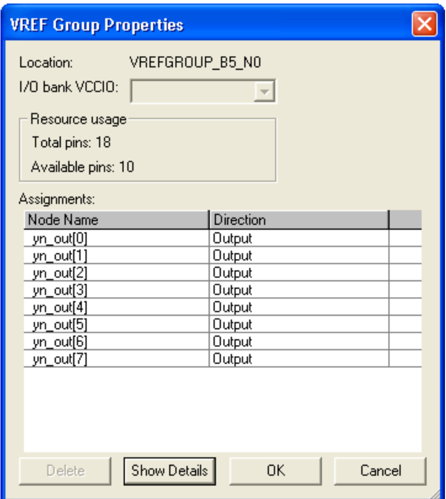


When you turn on **Show VREF Groups**, the Package View allows you to show the properties of each VREF group. Select a VREF group in the Package View, and on the View menu, click **VREF Group Properties**. The **VREF Group Properties** dialog box appears (Figure 5–38).

In the VREF **Group Properties** dialog box, all node names assigned to the VREF group are listed. Click **Show Details** to view node names that are assigned to pin numbers within the VREF group.

Any design pins that are assigned to the VREF group and not to a pin number are listed in the **Assignments** list. The **Resource usage** section describes the total number of pins in the VREF group and the total number of available assignable pins. It also keeps a running tally of the input, output, and bidirectional pins.

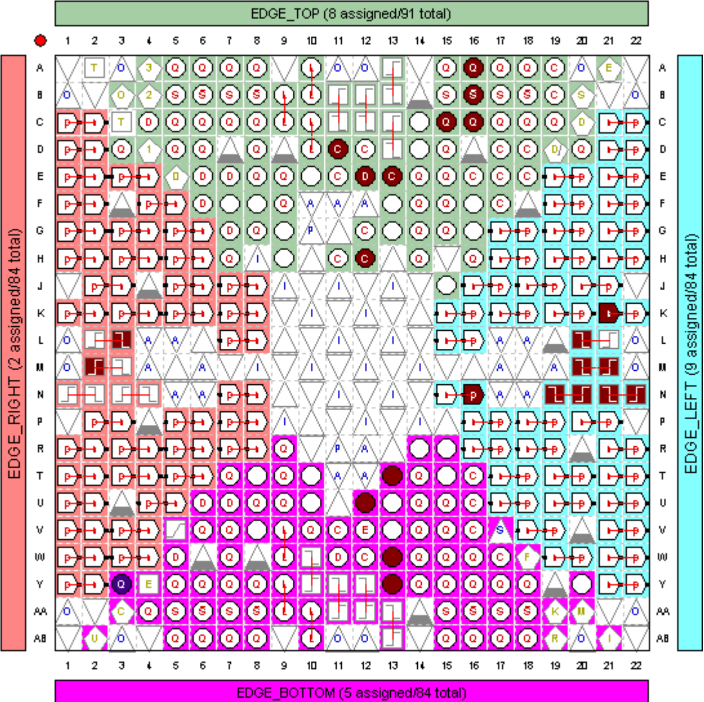
Figure 5–38. VREF Group Properties



Show Edges

You can use different colors to indicate the four edges of the package in the Package View (Figure 5–39). To do this, on the View menu, point to **Show** and click **Show Edges**, or from the right-click menu, click **Show Edges**. If the exact location of a pin is not a priority when planning your I/O pins, use an Edge assignment.

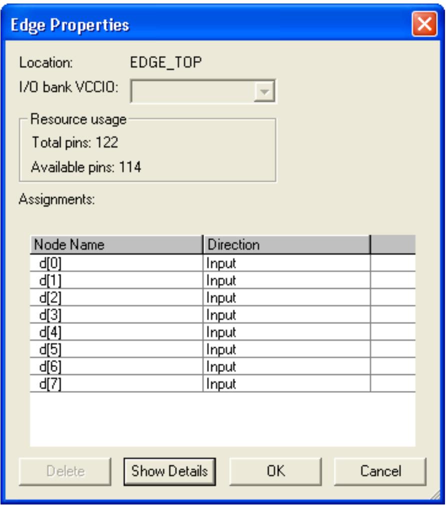
Figure 5–39. Package View with Edges



When you turn on **Show Edges**, the Package View allows you to show the properties of each Edge. Select an Edge in the Package View. On the View menu, click **Edge Properties.** The **Edge Properties** dialog box appears.

In the **Edge Properties** dialog box, all node names assigned to the Edge are listed (Figure 5–40). To view all node names assigned to a pin number within an Edge, in the **Edge Properties** dialog box, click **Show Details**.

Figure 5–40. Edge Properties



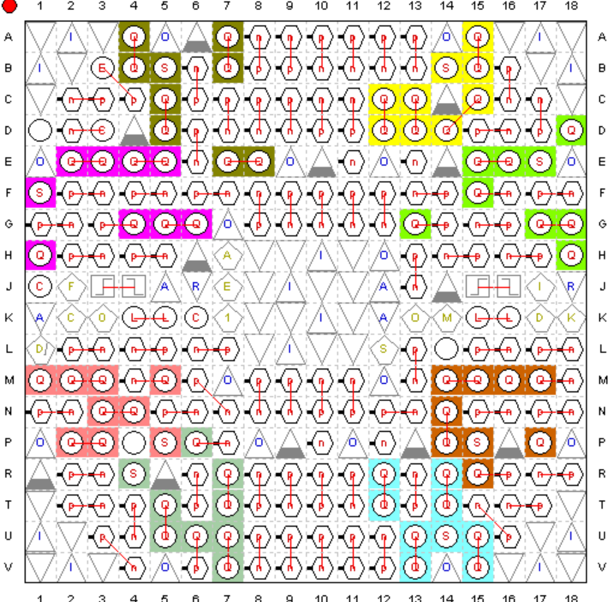
Show DQ/DQS Pins

You can use different colors to highlight groups of DQ and DQS pins in the Package View (Figure 5–41). To do this, on the View menu, point to **Show** and click **Show DQ/DQS Pins**, or from the right-click menu, click **Show DQ/DQS Pins**. Highlighting these DQ/DQS groups easily identifies which DQ pins are associated with a specific DQS strobe pin. Select between the following DQ/DQS modes:

- ×4 Mode
- ×8/×9 Mode
- ×16/×18 Mode
- ×32/×36 Mode

© March 2009 Altera Corporation

Figure 5–41. DQ/DQS Pins *(Note 1)*



Note to Figure 5–41:

(1) This DQ/DQS view shows an ×8 mode.

For example, when implementing DDR II in a Stratix II device, there are dedicated pins designed specifically to be used as DQ and DQS pins.

For information about using the ALTDQ and ALTDQS megafunctions to configure DQ and DQS pins, refer to the *Data (DQ) and Data Strobe (DQS) Megafunction User Guide (ALTDQ and ALTDQS)*.

Displaying and Accepting Fitter Placements

In addition to the **Show I/O Banks**, **Show VREF Groups**, and **Show Edge** views, you can also show pins placed by the Fitter. To display these pins, on the View menu, or in the Pin Planner toolbar, or on the right-click menu in the Package View, point to **Show** and click **Show Fitter Placements**.

The Fitter provides optimal placement to unassigned pins based on design constraints when you perform a compilation or an I/O Assignment Analysis. When you choose **Show Fitter Placements**, the Fitter-placed pins are shown as green-filled pins in the Package View. You can create a copy of the Fitter placements in your project **.qsf** file using the **Back-Annotate Assignments** command. Refer to "Accepting Fitter Placements—Back-Annotating Assignments" on page 5–74.

Altera recommends you use the Pin Planner to create and edit pin-related assignments. However, you might find some of the other tools provided for use with the Quartus II software to be useful for working with pin-related assignments. The following sections describe these tools.

Assignment Editor

The Assignment Editor provides a spreadsheet-like interface that allows you to create and change all pin-related assignments.

Two methods are available for making pin assignments with the Assignment Editor. The first involves selecting from all assignable pin numbers of the device and assigning a pin name from your design to this location.

The second involves selecting from all pin names in your design and assigning a device pin number to the design pin name. In either method, take advantage of row background coloring (pin numbers within the same I/O bank have a common background color), auto fill node names, and pin numbers to assist in making your assignments.

Setting Pin Locations from the Device Pin Number List

It is important to understand the properties of a pin location before assigning that location to a pin in your design. For example, you must know which I/O bank or VREF group the pin belongs to when following pin placement guidelines.

For more information about pin placement guidelines, refer to the appropriate device handbook.

Before creating pin-related assignments, perform analysis and elaboration or analysis and synthesis on your design to create a database of your design pin names. Then perform the following steps:

1. To open the Assignment Editor, on the Assignments menu, click **Assignment Editor.**
2. In the **Category list**, select **Pin**.

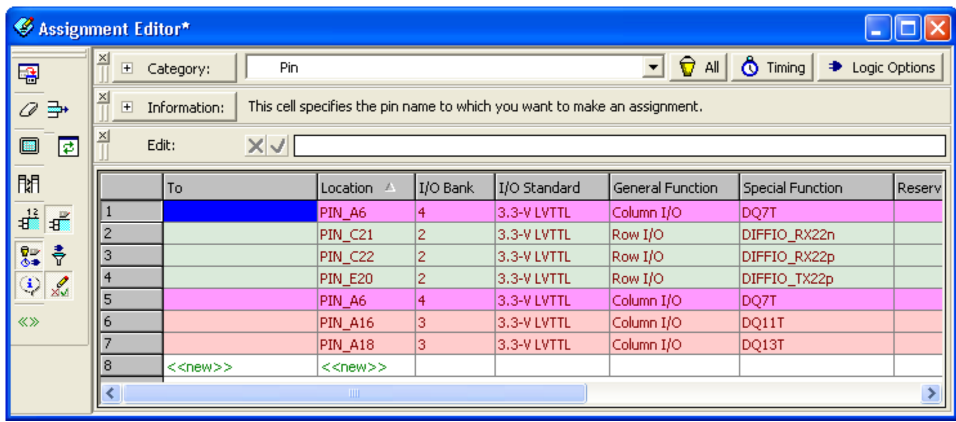
   Creating pin assignments can be difficult when you must check which I/O bank the pin belongs to or which VREF pad the pin uses. By selecting the **Pin** category, more pin-related information is visible in the spreadsheet to help you create pin location assignments.

   The Assignment Editor does not show assignments to individual nodes made with wildcards or assignment groups.

3. On the View menu, click **Show All Assignable Pin Numbers**.

You can also view all assignable pins in the All Pins list in the Pin Planner. Right-click anywhere in the Groups list or All Pins list and click **Show Assignable Pins**. When the All Pins list filter is set to **Pins: unassigned** or **Pins: all**, a list of all assignable pin numbers for the targeted device is shown in the **Location** column (Figure 5–42).

Figure 5–42. Assignment Editor with Show All Assignable Pin Numbers



	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reserv
1		PIN_A6	4	3.3-V LVTTL	Column I/O	DQ7T	
2		PIN_C21	2	3.3-V LVTTL	Row I/O	DIFFIO_RX22n	
3		PIN_C22	2	3.3-V LVTTL	Row I/O	DIFFIO_RX22p	
4		PIN_E20	2	3.3-V LVTTL	Row I/O	DIFFIO_TX22p	
5		PIN_A6	4	3.3-V LVTTL	Column I/O	DQ7T	
6		PIN_A16	3	3.3-V LVTTL	Column I/O	DQ11T	
7		PIN_A18	3	3.3-V LVTTL	Column I/O	DQ13T	
8	<<new>>	<<new>>					

4. Find a pin number in the spreadsheet. In the same row, double-click the cell in the **To** column. Type the pin name or select a pin from the pull-down list. If analysis and elaboration has been performed, your design pins are listed in the pull-down list.

As you type in a pin name, the Assignment Editor automatically completes the field by looking up the pin names stored in the database created from the initial analysis and elaboration. Pin names already assigned to a pin location are shown in italics.

Setting Pin Locations from the Design Signal Name List

It is important to understand the properties of a pin location before assigning that location to a pin in your design. For example, you must know which I/O bank or VREF group the pin belongs to when following pin placement guidelines.

For more information about pin placement guidelines, refer to the appropriate device handbook.

To set the pin locations from the design pin name list, perform the following steps:

1. To open the Assignment Editor, on the Assignments menu, click **Assignment Editor**.
2. In the **Category** list, select **Pin**.

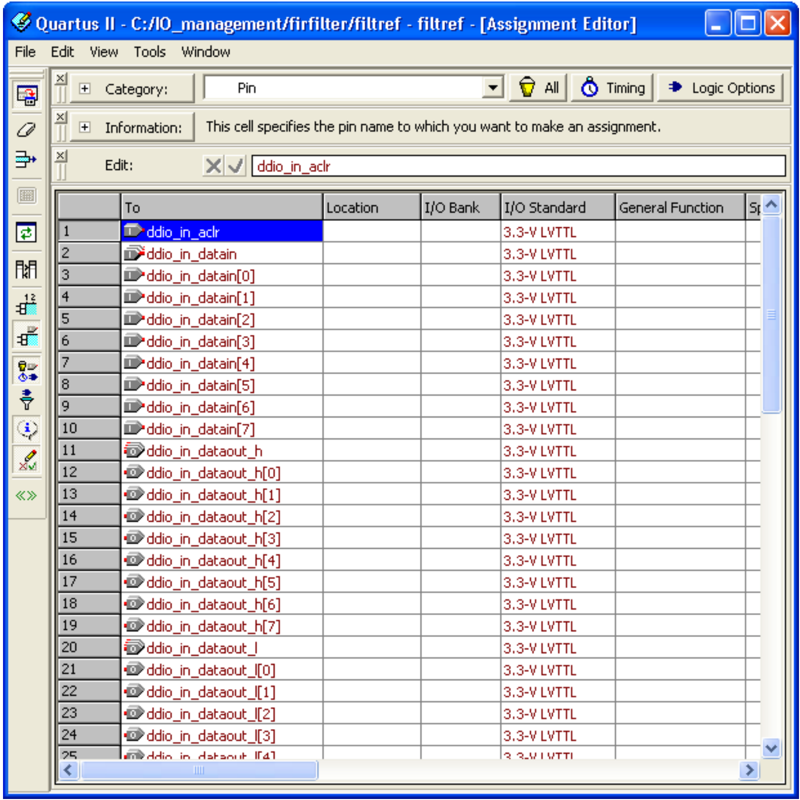
   Creating pin assignments can be difficult when you have to check which I/O bank the pin belongs to, or which VREF pad the pin uses. By selecting the **Pin** category, more pin-related information is visible in the spreadsheet to help you create pin location assignments.

   The Assignment Editor does not show assignments to nodes made with wildcards or assignment groups.

3. On the View menu, click **Show All Known Pin Names**.

   A list of all pin names in your design is shown in the **To** column (Figure 5–43).

Figure 5–43. Assignment Editor with Show All Known Pin Names



To list a selection of pin names from your design into the spreadsheet of the Assignment Editor, type the pin names with or without wild cards into the **Node Filter** bar. This is effective when you want to assign common pin-related assignments to a selection of pins in your design.

For more information about using the **Node Filter** bar, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

4. Find a pin name in the spreadsheet and double-click the cell in the same row of the **Location** column. Select a pin number from the pull-down list that contains all assignable pin numbers in the selected device. You can also start typing the pin number and let the Assignment Editor automatically complete it for you. Instead of typing `PIN_AA3`, type `AA3` and let the Assignment Editor auto complete the pin number to `PIN_AA3`.

   Pin locations that already have a pin name assignment appear in the Assignment Editor in italics.

For more information about using the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

Tcl Scripts

Tcl scripting allows you to write scripts to create pin-related assignments. To run a Tcl script with your project, type the following command at a system prompt:

```
quartus_sh -t my_tcl_script.tcl ↵
```

You can also type individual Tcl commands into the Tcl console window. To use the Tcl console, on the View menu, point to **Utility Windows** and click **Tcl Console**. In the Tcl Console window, type your Tcl commands. Example 5–3 shows a list of Tcl commands that creates pin-related assignments to the input pin `address[10]`.

Example 5–3. Tcl Commands to Create Pin-Related Assignments

```
set_location_assignment PIN M20 -to address[10] -comment"Address pin to Second FPGA"
set_instance_assignment -name IO_STANDARD "2.5 V" -to address[10]
set_instance_assignment -name CURRENT_STRENGTH_NEW "MAXIMUM CURRENT" -to address[10]
```

When you make an assignment in the Assignment Editor or the Pin Planner, display the equivalent Tcl command in the Messages window by performing the following steps:

1. On the Tools menu, click **Options**. The **Options** dialog box appears.
2. In the **Category** list, select **Assignment Editor**. The **Assignment Editor** page opens.
3. Turn on **Echo Tcl Commands**.
4. Click **OK**.

For more information about using Tcl scripts to create pin-related assignments, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook* and to the *Quartus II Scripting Reference Manual*.

Chip Planner or Timing Closure Floorplan

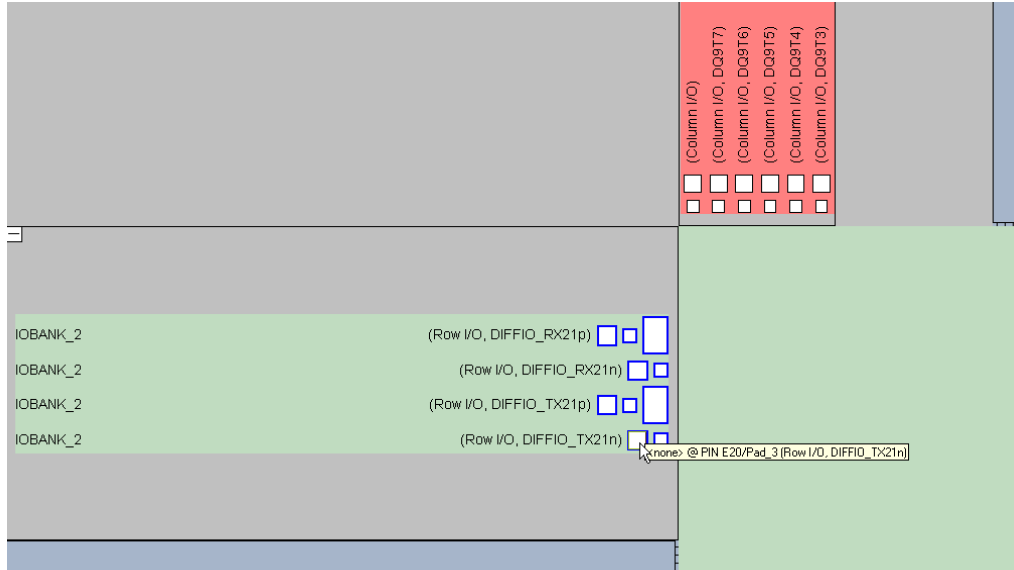
The floorplan of the device shows the pins in the same order as the pads of the device. Understanding the relative distance between a pad and related logic can help you meet your timing requirements.

You can view the floorplan of the device in the Chip Planner or Timing Closure Floorplan. For more information about supported device families in the Chip Planner or Timing Closure Floorplan, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Use either tool to find the distances between user I/O pads and V_{CC}, GND, and VREF pads to avoid signal integrity issues (Figure 5–44).

For more information about pin placement guidelines, refer to the *Selectable I/O Standards* chapter of the appropriate device handbook.

Figure 5–44. Timing Closure Floorplan of EP1C6F256I7



You can create a pin location assignment by selecting a pin and selecting a desired location. To do this, perform the following steps:

1. To open the Timing Closure Floorplan, on the Assignment menu, click **Timing Closure Floorplan**. To open the Chip Planner, on the Tools menu, click **Chip Planner (Floorplan & Chip Editor)**.
2. On the View Chip Planner, point to **Utility Windows** and click **Node Finder**. The **Node Finder** dialog box appears.
3. In the **Filter** list, select **Pins: all** and click **List** to see all the nodes in the design.
4. Select a node from the **Nodes Found** list and drag the selection into a pin location in the floorplan.

For more information about using the Timing Closure Floorplan, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Using HDL

You can use synthesis attributes or I/O primitives to embed pin-related assignments in your HDL code directly. When you analyze and synthesize your HDL code, the information in the HDL code is converted into appropriate assignments. There are two ways to specify pin related assignments using HDL:

- Using synthesis attributes for signal names that are top-level pins
- Using low-level I/O primitives such as ALT_BUF_IN to specify input, output, and differential buffers, and setting their parameters or attributes

The following sections explain how to use synthesis attributes and I/O primitives for your top-level pins.

Synthesis Attributes

Synthesis attributes allow you to embed assignments in your HDL code. The Quartus II software reads these synthesis attributes and translates them into assignments. The Quartus II integrated synthesis supports `chip_pin`, `useioff`, and `altera_attribute` synthesis attributes.

For more information about integrated synthesis, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

For synthesis attributes support by third-party synthesis tools, contact your vendor.

chip_pin and useioff

You can use the `chip_pin` and `useioff` synthesis attributes to embed pin location and fast output and input register assignments, respectively. For all other assignments, including pin-related assignments, use the `altera_attribute` synthesis attribute as discussed in the "altera_attribute" section.

Example 5–4 and Example 5–5 embed a location and fast input assignment into both a Verilog HDL and VHDL design file using the `chip_pin` and `useioff` synthesis attributes.

Example 5–4. Verilog HDL Example

```
input my_pin1 /* synthesis chip_pin = "C1" useioff = 1 */;
```

Example 5–5. VHDL Example

```
entity my_entity is
       port(
                 my_pin1: in std_logic
       );
end my_entity;

architecture rtl of my_entity is
attribute useioff : boolean;
attribute useioff of my_pin1 : signal is true;
attribute chip_pin : string;
attribute chip_pin of my_pin1 : signal is "C1";
begin -- The architecture body
end rtl;
```

altera_attribute

To create other pin-related assignments, use the `altera_attribute` attribute. The `altera_attribute` attribute is understood only by Quartus II integrated synthesis and supports all types of instance assignments. Example 5–6 and Example 5–7 use `altera_attribute` to embed the fast input register and I/O standard assignments into both a Verilog HDL and a VHDL design file.

Example 5–6. Verilog HDL Example

```
input my_pin1 /* synthesis altera_attribute = "-name FAST_INPUT_REGISTER ON; -name
IO_STANDARD \"2.5 V\" " */ ;
```

Example 5–7. VHDL Example

```
entity my_entity is
       port(
              my_pin1: in std_logic
       );
end my_entity;
architecture rtl of my_entity is
begin

attribute altera_attribute : string;
attribute altera_attribute of my_pin1: signal is "-name FAST_INPUT_REGISTER ON;
-- The architecture body
end rtl;
```

In Quartus II software version 8.0 and later, the pin-related assignments made using synthesis attributes are shown in Pin Planner's All Pins list and Package views. When you modify or delete these pin assignments in the Pin Planner, you get an informational message suggesting that the pin-related assignments have been changed. If you recompile your project, your pin-related assignment in the Pin Planner, which is contained in your **.qsf** file, has precedence over the assignments you made using synthesis attributes in your HDL file.

For detailed information about synthesis attributes and their usage syntax, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook* or the Quartus II Help.

Using Low-Level I/O Primitives

You can make pin related assignments for your top-level nodes using low-level primitives, which allow you to create pin location assignments, and set I/O standards, current drive strengths, slew rates, and on-chip termination (OCT).

For more information about using low-level I/O primitives in your design, refer to *Designing with Low-Level Primitives User Guide* or the Quartus II Help.

The pin-related assignments made using primitives do not appear in the Pin Planner.

During and after pin-related assignments creation, you must validate your pin-related assignments using the **Enable Live I/O Check** option and running I/O Assignment Analysis.

Validating Pin Assignments

The Quartus II software includes built-in I/O rules to guide you in pin placement. The Quartus II software checks your pin-related assignments against these rules during pin planning. You must validate all pin-related assignments in your design. You can enable the live I/O check feature and must use I/O Assignment Analysis to validate pin-related assignments against the predefined I/O rules encoded in the Quartus II software. To fully validate these assignments against all the I/O timing checks, you must perform full compilation.

Using the Live I/O Check Feature to Validate Pin Assignments

In Quartus II software version 7.2 and later, the live I/O check feature provides live I/O rules checking capability. When the live I/O check feature is enabled, pin-related assignment error and warning messages appear immediately in the Quartus II Messages window as you create pin-related assignments in the Pin Planner. This feature enhances your productivity by showing you warnings and errors as you create pin-related assignments, before you proceed to the next step in your design flow.

The most basic I/O rules are the I/O buffer rules. The I/O buffer rules checked by the live I/O Check feature include:

- V_{CCIO} voltage compatibility rules
- VREF voltage compatibility rules
- Electromigration (current density rules)
- Simultaneous Switching Output (SSO) rules
- I/O properties compatibility rules such as drive strength compatibility, I/O standard compatibility, `PCI_IO` clamp diode compatibility, and I/O direction compatibility

An additional category of I/O rules is the set of I/O system rules. These rules can be checked only after you generate a synthesized (mapped) netlist of your design. The I/O system rules are checked when you perform I/O assignment analysis as described in "Using I/O Assignment Analysis to Validate Pin Assignments" on page 5–59.

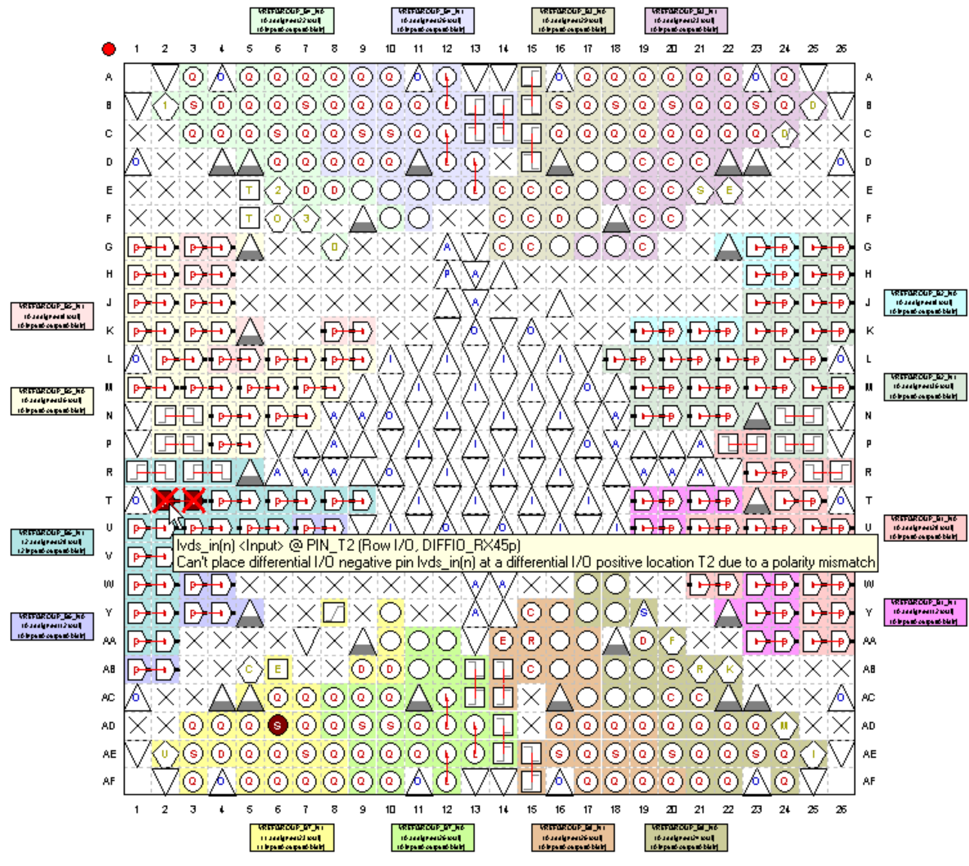
You can enable or disable the live I/O check feature at any time. By default, the live I/O check feature is turned off.

To enable or disable the live I/O check feature in the Quartus II user interface:

1. Verify the Pin Planner tool in the Quartus II software is active.
2. In the Quartus II View menu, select **Live I/O Check,** or, in the Pin Planner, click on the Live I/O Check icon.

While the live I/O check feature is enabled, the Quartus II software immediately checks whether your new pin-related assignments and revisions pass the basic I/O buffer rules. The detailed messages are printed in the Messages window of the Quartus II software and shown in Package View (Figure 5–45).

Figure 5–45. Live I/O Check Results in Package View

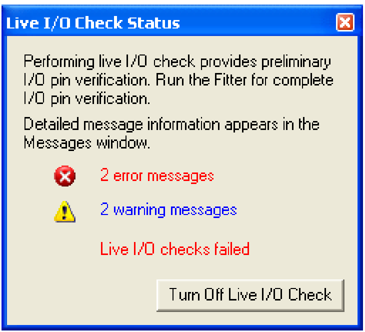


The Live I/O Check Status window displays the total numbers of errors and warnings while you create and edit pin-related assignments. To open the Live I/O Check Status window, shown in Figure 5–46, in the Quartus II View menu, click **Live I/O Check Status**.

For details about a specific message, refer to the Quartus II Help.

© March 2009 Altera Corporation

Figure 5–46. Live I/O Check Status Window in the Quartus II Software



Though the live I/O check feature checks all the basic I/O buffer rules, you must run I/O assignment analysis to validate your pin-related assignments against the complete set of I/O system rules. All rules including the basic I/O buffer rules and I/O system rules can be found in Table 5–5 on page 5–71 and Table 5–6 on page 5–72.

Using I/O Assignment Analysis to Validate Pin Assignments

This section describes a design flow that includes making and analyzing pin assignments with the **Start I/O Assignment Analysis** command in the Quartus II software during and after the development of your HDL design.

The **Start I/O Assignment Analysis** command allows you to check your I/O assignments early in the design process. With this command, you can check the legality of pin assignments before, during, or after you compile your design. If design files are available, you can use this command to perform more thorough legality checks on your design's I/O pins and surrounding logic. These checks include proper reference voltage pin usage, valid pin location assignments, and acceptable mixed I/O standards.

The **Start I/O Assignment Analysis** command can be used for designs that target Stratix series, Cyclone series, and MAX® II device families.

I/O Assignment Analysis Design Flows

The I/O assignment analysis design flows depend on whether your project contains design files. The following examples show two different circumstances in which I/O Assignment Analysis can be used:

- Use the flow shown in Figure 5–47 on page 5–61 if the board layout must be complete before starting the FPGA design. This flow does not require design files and checks the legality of your pin assignments.
- With a complete design, use the flow shown in Figure 5–49 on page 5–63. This flow thoroughly checks the legality of your pin assignments against any design files provided.

Each flow involves creating pin assignments, running analysis, and reviewing the report file.

You should run the analysis each time you add or modify a pin-related assignment. You can use the **Start I/O Assignment Analysis** command frequently because it completes quickly.

The analysis checks pin assignments and surrounding logic for illegal assignments and violations of board layout rules. For example, the analysis checks whether your pin location supports the assigned I/O standard, current strength, supported VREF voltages, and whether a PCI diode is permitted.

Along with the pin-related assignments, the **Start I/O Assignment Analysis** command also checks blocks that directly feed or are fed by resources such as a PLLs, LVDS, or gigabit transceiver blocks.

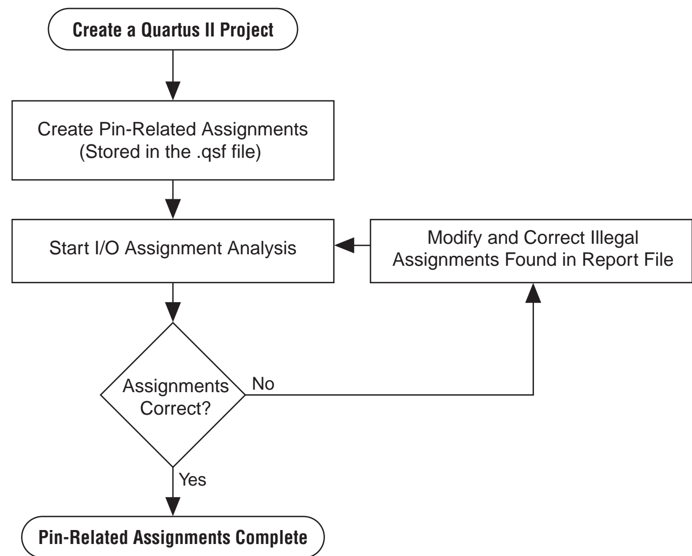
I/O Assignment Analysis without Design Files

During the early stages of developing an FPGA device, board layout engineers might request preliminary or final pin-outs. It is time consuming to manually check whether the pin-outs violate any design rules. Instead, use the **Start I/O Assignment Analysis** command to quickly perform basic checks on the legality of your pin assignments.

Without a complete design, the analysis performs limited checks and cannot guarantee that your assignments do not violate design rules.

The **I/O Assignment Analysis** command can perform limited checks on pin assignments made in a Quartus II project that has a device specified, but might not yet include any HDL design files. For example, you can create a Quartus II project with only a target device specified and create pin-related assignments based on circuit board layout considerations that are already determined. Even though the Quartus II project does not yet contain any design files, you can reserve input and output pins and make pin-related assignments for each pin using the Pin Planner or Assignment Editor. After you assign an I/O standard to each reserved pin, run the I/O Assignment Analysis to ensure that there are no I/O standard conflicts in each I/O bank. Figure 5–47 shows the work flow for assigning and analyzing pin-outs without design files.

Figure 5–47. Assigning and Analyzing Pin-Outs without Design Files



To assign and analyze pin-outs using the **Start I/O Assignment Analysis** command without design files, perform the following steps:

1. In the Quartus II software, create a project.
2. Use the Pin Planner, Assignment Editor, or a Tcl script to create pin locations and related assignments. For I/O Assignment Analysis to determine the type of pin, you must reserve your I/O pins. For information about reserving pins in the Pin Planner, refer to "Creating Reserved Pin Assignments" on page 5–34. For information about reserving pins in the Assignment Editor, refer to "Reserving Pins" on page 5–65.

If you make pin-related assignments in Mentor Graphics I/O Designer software, you can import an **.fx** file into the Quartus II software.

3. To start the analysis, on the Processing menu, point to **Start** and click **Start I/O Assignment Analysis**.

For information about using a Tcl script or command prompt to start the analysis, refer to "Scripting Support" on page 5–72.

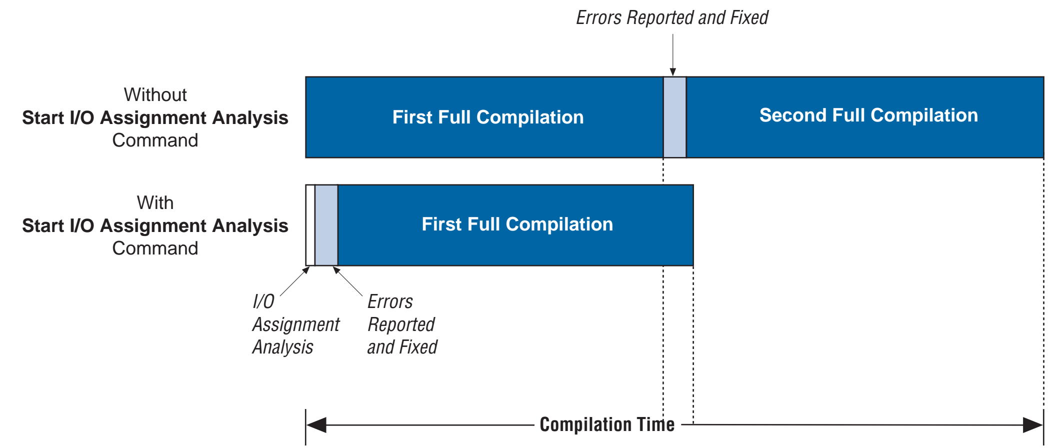
4. View the messages in the Compilation Report window, Fitter report file (<*project name*>**.fit.rpt**), or in the Messages window.
5. Correct any errors and violations reported by the I/O Assignment Analysis.

Repeat steps 1 through 5 until all of the errors are corrected.

I/O Assignment Analysis with Design Files

During a full compilation, the Quartus II software does not report illegal pin assignments until the Fitter stage. To validate pin assignments sooner, run the **Start I/O Assignment Analysis** command after performing analysis and synthesis and before performing a full compilation. Typically, the analysis runs quickly. Figure 5–48 shows the benefits of using the **Start I/O Assignment Analysis** command.

Figure 5–48. Saving Compilation Time with the Start I/O Assignment Analysis Command

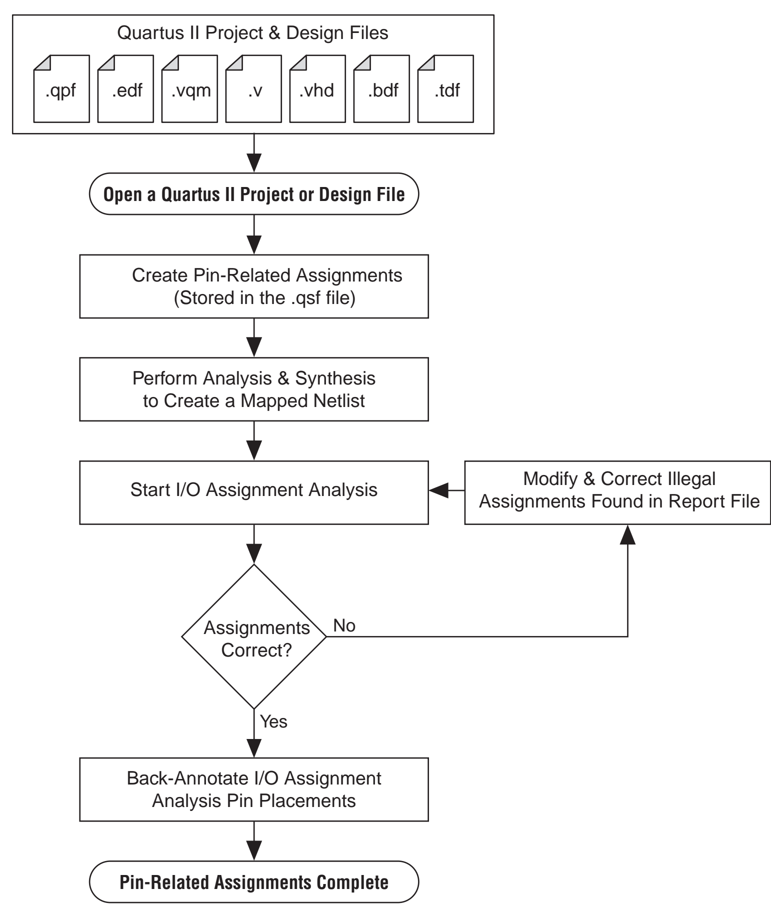


The rules that are checked by the I/O assignment analysis depend on the completeness of the design. With a complete design, the **Start I/O Assignment Analysis** command thoroughly checks the legality of all pin-related assignments. With a partial design, which can be just the top-level wrapper file, the **Start I/O Assignment Analysis** command checks the legality of those pin-related assignments for which it has enough information.

For example, you might assign a clock to a user I/O pin instead of assigning it to a dedicated clock pin, or design the clock to drive a PLL that has not yet been instantiated in the design. Because the **Start I/O Assignment Analysis** command does not account for the logic that the pin drives, it is not able to check that only a dedicated clock input pin can drive the clock port of a PLL.

To obtain better coverage, analyze as much of the design as possible, especially logic that connects to pins. For example, if your design includes PLLs or LVDS blocks, you should include these MegaWizard Plug-In Manager-generated files in your project for analysis (Figure 5–49).

Figure 5–49. Assigning and Analyzing Pin-Outs with Design Files



To assign and analyze pin-outs using the **Start I/O Assignment Analysis** command with design files, perform the following steps:

1. Create a project including your design files.
2. Create pin-related assignments with the Pin Planner or Assignment Editor.

You can also create pin-related assignments by importing them from a **.csv** or **.fx** file, executing Tcl commands, or editing the **.qsf** file directly. On the Processing menu, point to **Start** and click **Start Analysis & Synthesis** to generate an internal mapped netlist.

For information about using a Tcl script or the command prompt to start the analysis, refer to "Scripting Support" on page 5–72.

3. On the Processing menu, point to **Start** and click **Start I/O Assignment Analysis** to start the analysis.
4. View the messages in the Compilation Report or in the Messages window.
5. Use the Pin Planner or Assignment Editor to correct any errors and violations reported.
6. Use the **Start I/O Assignment Analysis** command until all the errors are corrected.

Using Output Enable Group Logic Option Assignments with I/O Assignment Analysis

Each device has a certain number of VREF pins, and each VREF pin supports a certain number of I/O pins. Check the device pin-outs to locate the VREF pins and their associated I/O pins. A VREF pin and its supported I/O pins are called a VREF bank. The VREF pins are only used for VREF I/O standards; for example, SSTL and HSTL input pins. VREF outputs do not require the VREF pin. When a voltage-referenced input is present in a VREF bank, only a certain number of outputs can be present in that VREF bank. For the Stratix II flip chip package, only 20 outputs can be present in a VREF bank when a VREF I/O standard input is present in that bank.

For interfaces that use bidirectional VREF I/O pins, the VREF restriction must be met when the pins are driving in either direction. If a set of bidirectional signals are controlled by different output enables, the **I/O Assignment Analysis** command treats these as independent output enables. Use the output enable group logic option assignment to treat the set of bidirectional signals as a single output enable. This is important in the case of external memory interfaces.

For example, in the case of a DDR2 interface in a Stratix II device, a Stratix II device can have 30 pins in a VREF group. Each byte lane for a ×8 DDR2 interfaces has 1 DQS pin and 8 DQ pins, for a total of 9 pins per byte lane. DDR2 uses SSTL18 as its I/O standard, which is a VREF I/O standard. In typical interfaces, each byte lane has its own output enable. In this example, the DDR2 interface has 4 byte lanes. Using 30 I/O pins in a VREF group, there are 3 byte lanes and an extra byte lane that supports the 3 remaining pins. If you do not use the output enable group logic option assignment, the **I/O Assignment Analysis** command analyzes each byte lane as an independent group driven by a unique output enable. With this arrangement, the worst-case scenario is when the 3 pins are inputs, and the other 27 pins are outputs. In this case, the 27 output pins violate the 20-output pin limit.

In a DDR2 interface, all DQS and DQ pins are always driven in the same direction. Therefore, the I/O Assignment Analysis reports an error that is not applicable to your design. Assigning an output enable group logic option assignment to the DQS and DQ pins forces the I/O Assignment Analyzer to check these pins as a group driven by a common output enable. When using the output enable group logic option assignment, the DQS and DQ pins are checked as all input pins or all output pins. This does not violate the rules described in Table 5–5 on page 5–71 and Table 5–6 on page 5–72.

The value for the output enable group logic option assignment should be an integer value. All sets of signals that are driving in the same direction should be given the same integer value. You can also use the output enable group logic option assignment with pins that are driven only at certain times. For example, the data mask signal in DDR2 interfaces is an output signal, but it is driven only when the DDR2 is writing (bidirectional signals are outputs). Therefore, an output enable group logic option assignment should assign to the data mask the same value as to the DQ and DQS signals.

Output enable groups can also be used on VREF input pins. If the VREF input pins are not active during the time the outputs are driving, add the VREF input pins to the output enable group. This procedure removes the VREF input pins from the VREF analysis. For example, the QVLD signal for RLDRAM II is only active during a read. During a write, the QVLD pin is not active and so it does not count as an active VREF input pin within the VREF group. The QVLD pins can be placed in the same output enable group as the RLDRAM II data pins.

Inputs for I/O Assignment Analysis

The **Start I/O Assignment Analysis** command reads the following inputs:

- Internal mapped netlist
- **.qsf** file

The internal mapped netlist is used when you have a partial or complete design. The **.qsf** file is always used to read all pin-related assignments for analysis.

Generating a Mapped Netlist

The **Start I/O Assignment Analysis** command uses a mapped netlist, if available, to identify the pin type and the surrounding logic. The mapped netlist is stored internally in the Quartus II software database.

To generate a mapped netlist, on the Processing menu, point to **Start** and click **Start Analysis & Synthesis**.

To use the `quartus_map` executable to run analysis and synthesis, type the following command at a system command prompt:

```
quartus_map <project name> ↵
```

Creating Pin-Related Assignments

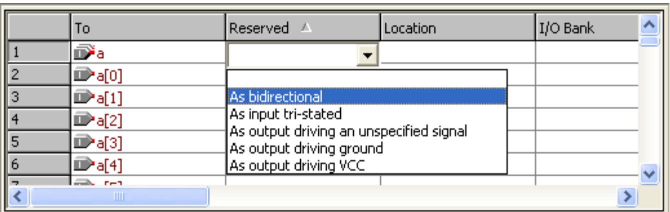
The **I/O Assignment Analysis** command reads a **.qsf** file containing all of your pin-related assignments. These pin-related assignments include pin settings such as I/O standards, drive strength, and location assignments. The following sections highlight some of the location assignments you can make.

Reserving Pins

If you do not have any design files, you can still reserve pin locations and create pin-related assignments. Reserving pins is necessary so that the **Start I/O Assignment Analysis** command has information about the pin and the pin type (input, output, or bidirectional) to correctly analyze the pins.

To reserve a pin, on the Assignments menu, click **Assignment Editor**. In the **Category** list, click **Pin** to open the Pin assignment category. Double-click the cell in the **Reserved** column that corresponds to the pin that you want to reserve. Use the drop-down arrow to select from the reserved pin options (Figure 5–50).

Figure 5–50. Reserving an Input Pin with the Assignment Editor



For more information about using the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

You can also reserve pins using the Pin Planner. For more information about the Pin Planner, refer to "Creating Reserved Pin Assignments" on page 5–34.

Location Assignments

You can create the following types of location assignments for your design and its reserved pins:

- Pin number
- I/O bank
- VREF group
- Edge

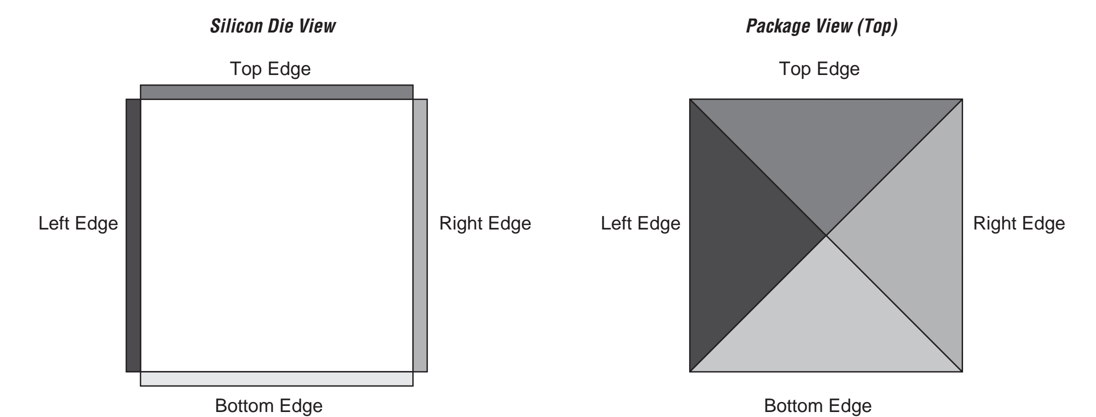
I/O bank, VREF group, and Edge location assignments are supported only for Stratix and Cyclone series device families.

You can assign a location to your pins using the Pin Planner or the Assignment Editor. To make a pin location assignment using the Assignment Editor, on the Assignments menu, click **Assignment Editor** and select the **Pin** category from the **Category** list. Type the pin name and select a location from the **Location** list.

It is common to place a group of pins (or bus) with compatible I/O standards in the same I/O bank or VREF group. For example, two buses with two compatible I/O standards, such as 2.5 V and SSTL-II, can be placed in the same I/O bank.

An easy way to place large buses that exceed the pins available in a particular I/O bank is to use edge location assignments. Edge location assignments improve the circuit board routing ability of large buses, because they are close together near an edge. Figure 5–51 shows Altera device package edges.

Figure 5–51. Die View and Package View of the Four Edges on an Altera Device



Suggested and Partial Placement

The **Start I/O Assignment Analysis** command automatically assigns suggested pin locations to unassigned pins in your design so it can perform pin legality checks. For example, if you assign an edge location to a group of LVDS pins, the **I/O Assignment Analysis** command assigns pin locations for each LVDS pin in the specified edge location and then performs legality checks.

To accept these suggested pin locations, on the Assignments menu, click **Back-Annotate Assignments**, select **Pin & device assignments**, and click **OK**. Back-annotation saves your pin and device assignments in the **.qsf** file.

Understanding the I/O Assignment Analysis Report and Messages

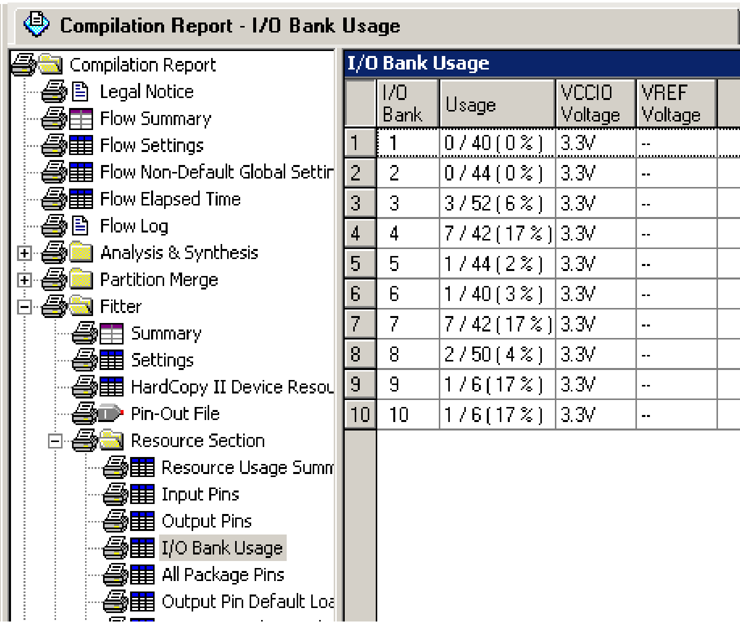
The **Start I/O Assignment Analysis** command generates detailed analysis reports and a **.pin** file. The detailed messages in the reports help you quickly understand and resolve pin assignment errors. Each message includes a related node name and a description of the problem.

To view the report file, on the Project menu, click **Compilation Report.** The Fitter section of the Compilation Report contains the following sections:

- Summary
- Settings
- Resource Section
- I/O Rules Section
- Device Options
- Advanced Fitter Data
- Pin-Out File
- Fitter Messages

The Resource Section categorizes the pins as Input Pins, Output Pins, and Bidir Pins. View the utilization of each I/O bank in your device in the I/O Bank Usage section (Figure 5–52).

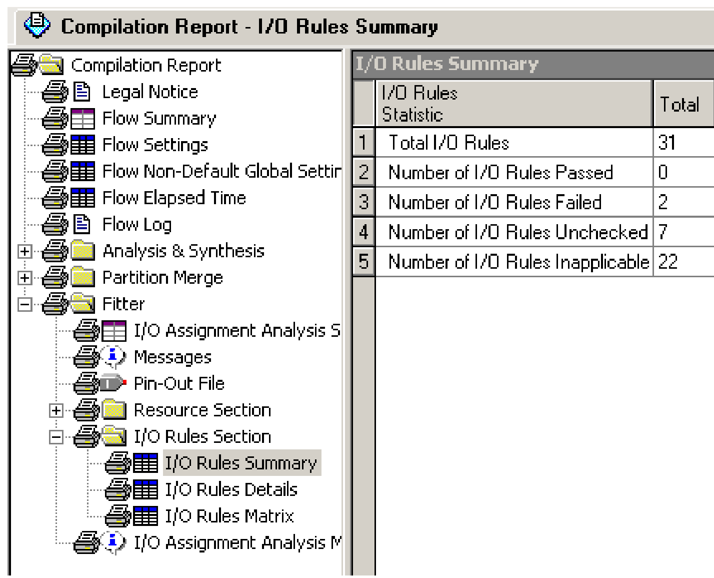
Figure 5–52. I/O Bank Usage Summary in the I/O Assignment Analysis Report



	I/O Bank	Usage	VCCIO Voltage	VREF Voltage
1	1	0 / 40 (0 %)	3.3V	--
2	2	0 / 44 (0 %)	3.3V	--
3	3	3 / 52 (6 %)	3.3V	--
4	4	7 / 42 (17 %)	3.3V	--
5	5	1 / 44 (2 %)	3.3V	--
6	6	1 / 40 (3 %)	3.3V	--
7	7	7 / 42 (17 %)	3.3V	--
8	8	2 / 50 (4 %)	3.3V	--
9	9	1 / 6 (17 %)	3.3V	--
10	10	1 / 6 (17 %)	3.3V	--

The I/O Rules Section includes detailed information about the I/O rules tested during I/O Assignment Analysis, in three sub-reports. The I/O Rules Summary report provides a quick summary of the number of I/O rules tested and how many applicable rules passed, how many failed, and how many were unchecked because of other failing rules (Figure 5–53).

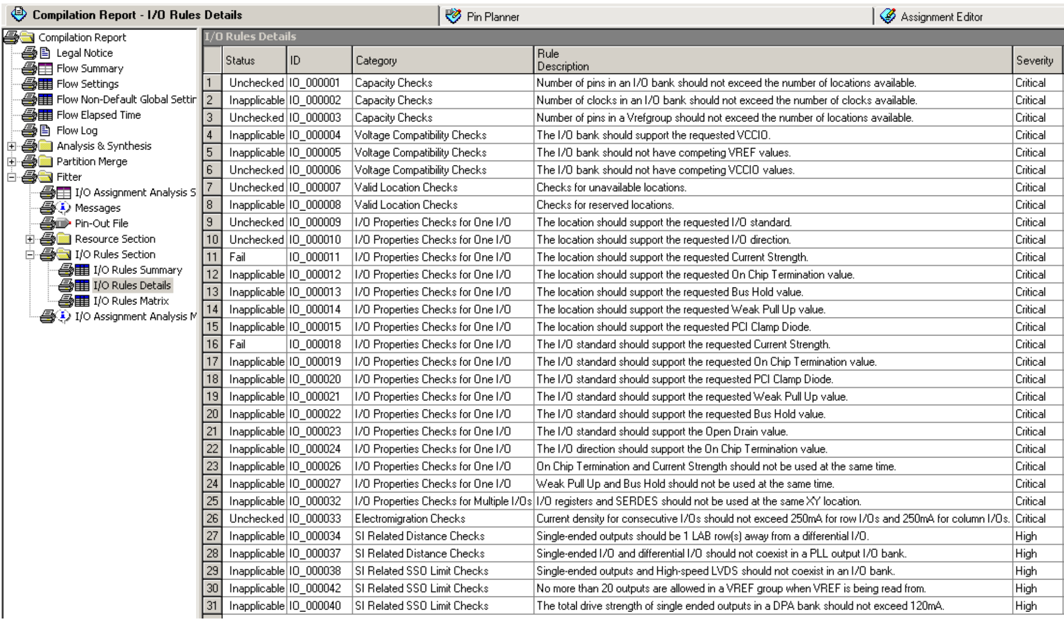
Figure 5–53. I/O Rules Summary Report



	I/O Rules Statistic	Total
1	Total I/O Rules	31
2	Number of I/O Rules Passed	0
3	Number of I/O Rules Failed	2
4	Number of I/O Rules Unchecked	7
5	Number of I/O Rules Inapplicable	22

The I/O Rules Details report provides detailed information on all I/O rules. Applicable rules indicate whether they passed, failed, or could not be checked (Figure 5–54). All rules are given a level of severity from Low to Critical to indicate their level of importance for an effective analysis.

Figure 5–54. I/O Rules Details Report

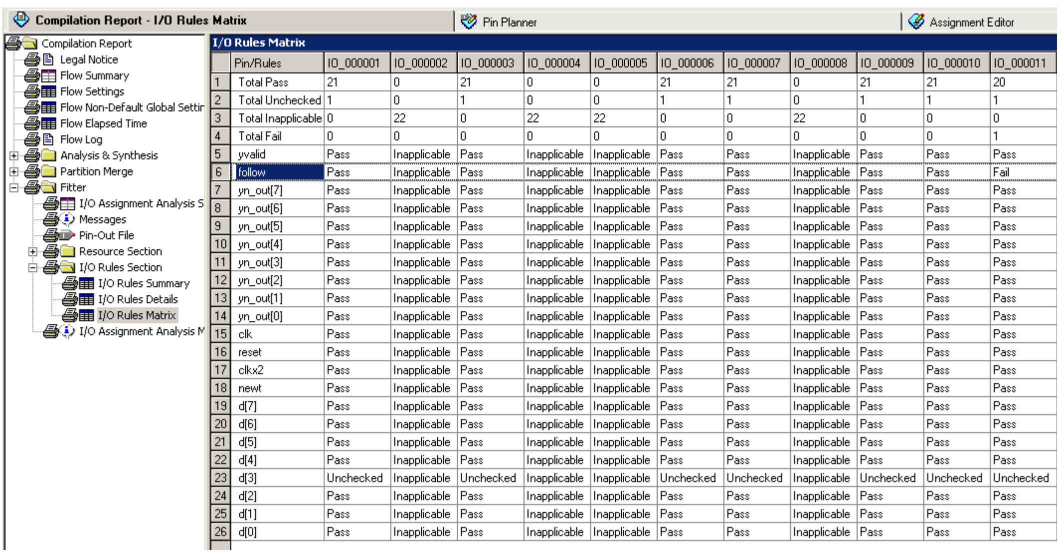


	Status	ID	Category	Rule Description	Severity
1	Unchecked	IO_000001	Capacity Checks	Number of pins in an I/O bank should not exceed the number of locations available.	Critical
2	Inapplicable	IO_000002	Capacity Checks	Number of clocks in an I/O bank should not exceed the number of clocks available.	Critical
3	Unchecked	IO_000003	Capacity Checks	Number of pins in a Vrefgroup should not exceed the number of locations available.	Critical
4	Inapplicable	IO_000004	Voltage Compatibility Checks	The I/O bank should support the requested VCCIO.	Critical
5	Inapplicable	IO_000005	Voltage Compatibility Checks	The I/O bank should not have competing VREF values.	Critical
6	Unchecked	IO_000006	Voltage Compatibility Checks	The I/O bank should not have competing VCCIO values.	Critical
7	Unchecked	IO_000007	Valid Location Checks	Checks for unavailable locations.	Critical
8	Inapplicable	IO_000008	Valid Location Checks	Checks for reserved locations.	Critical
9	Unchecked	IO_000009	I/O Properties Checks for One I/O	The location should support the requested I/O standard.	Critical
10	Unchecked	IO_000010	I/O Properties Checks for One I/O	The location should support the requested I/O direction.	Critical
11	Fail	IO_000011	I/O Properties Checks for One I/O	The location should support the requested Current Strength.	Critical
12	Inapplicable	IO_000012	I/O Properties Checks for One I/O	The location should support the requested On Chip Termination value.	Critical
13	Inapplicable	IO_000013	I/O Properties Checks for One I/O	The location should support the requested Bus Hold value.	Critical
14	Inapplicable	IO_000014	I/O Properties Checks for One I/O	The location should support the requested Weak Pull Up value.	Critical
15	Inapplicable	IO_000015	I/O Properties Checks for One I/O	The location should support the requested PCI Clamp Diode.	Critical
16	Fail	IO_000018	I/O Properties Checks for One I/O	The I/O standard should support the requested Current Strength.	Critical
17	Inapplicable	IO_000019	I/O Properties Checks for One I/O	The I/O standard should support the requested On Chip Termination value.	Critical
18	Inapplicable	IO_000020	I/O Properties Checks for One I/O	The I/O standard should support the requested PCI Clamp Diode.	Critical
19	Inapplicable	IO_000021	I/O Properties Checks for One I/O	The I/O standard should support the requested Weak Pull Up value.	Critical
20	Inapplicable	IO_000022	I/O Properties Checks for One I/O	The I/O standard should support the requested Bus Hold value.	Critical
21	Inapplicable	IO_000023	I/O Properties Checks for One I/O	The I/O standard should support the Open Drain value.	Critical
22	Inapplicable	IO_000024	I/O Properties Checks for One I/O	The I/O direction should support the On Chip Termination value.	Critical
23	Inapplicable	IO_000026	I/O Properties Checks for One I/O	On Chip Termination and Current Strength should not be used at the same time.	Critical
24	Inapplicable	IO_000027	I/O Properties Checks for One I/O	Weak Pull Up and Bus Hold should not be used at the same time.	Critical
25	Inapplicable	IO_000032	I/O Properties Checks for Multiple I/Os	I/O registers and SERDES should not be used at the same XY location.	Critical
26	Unchecked	IO_000033	Electromigration Checks	Current density for consecutive I/Os should not exceed 250mA for row I/Os and 250mA for column I/Os.	Critical
27	Inapplicable	IO_000034	SI Related Distance Checks	Single-ended outputs should be 1 LAB row(s) away from a differential I/O.	High
28	Inapplicable	IO_000037	SI Related Distance Checks	Single-ended I/O and differential I/O should not coexist in a PLL output I/O bank.	High
29	Inapplicable	IO_000038	SI Related SSO Limit Checks	Single-ended outputs and High-speed LVDS should not coexist in an I/O bank.	High
30	Inapplicable	IO_000042	SI Related SSO Limit Checks	No more than 20 outputs are allowed in a VREF group when VREF is being read from.	High
31	Inapplicable	IO_000040	SI Related SSO Limit Checks	The total drive strength of single ended outputs in a DPA bank should not exceed 120mA.	High

The I/O Rules Matrix shows how each I/O rule was tested on each pin in the design (Figure 5–55). Applicable rules that could be checked either pass or fail for each pin.

To find and make pin assignment adjustments on a pin that fails an I/O rule, right-click the pin name. Point to **Locate** and select a location where the pin exists, such as the Pin Planner. Make appropriate changes to fix the pin assignments and rerun I/O Assignment Analysis. Check the resulting I/O Rules Matrix to verify that your changes fixed the problem and allowed the failing pin assignment to pass. To rerun I/O rule analysis, on the Processing menu, point to **Start** and click **Start I/O Assignment Analysis**.

Figure 5–55. I/O Rules Matrix



	Pin/Rules	IO_000001	IO_000002	IO_000003	IO_000004	IO_000005	IO_000006	IO_000007	IO_000008	IO_000009	IO_000010	IO_000011
1	Total Pass	21	0	21	0	0	21	21	0	21	21	20
2	Total Unchecked	1	0	1	0	0	1	1	0	1	1	1
3	Total Inapplicable	0	22	0	22	22	0	0	22	0	0	0
4	Total Fail	0	0	0	0	0	0	0	0	0	0	1
5	yvalid	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
6	follow	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Fail
7	yn_out[7]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
8	yn_out[6]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
9	yn_out[5]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
10	yn_out[4]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
11	yn_out[3]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
12	yn_out[2]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
13	yn_out[1]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
14	yn_out[0]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
15	clk	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
16	reset	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
17	clkx2	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
18	newt	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
19	d[7]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
20	d[6]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
21	d[5]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
22	d[4]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
23	d[3]	Unchecked	Inapplicable	Unchecked	Inapplicable	Inapplicable	Unchecked	Unchecked	Inapplicable	Unchecked	Unchecked	Unchecked
24	d[2]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
25	d[1]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass
26	d[0]	Pass	Inapplicable	Pass	Inapplicable	Inapplicable	Pass	Pass	Inapplicable	Pass	Pass	Pass

The **Fitter Messages** page stores all messages including errors, warnings, and information messages.

You can view the detailed messages in the **Fitter Messages** page in the Compilation Report and in the **Processing** tab in the Messages window. To open the Messages window, on the View menu, point to **Utility windows** and click **Messages**.

Use the **Location** box to help resolve error messages. Select from the **Location** list and click **Locate**.

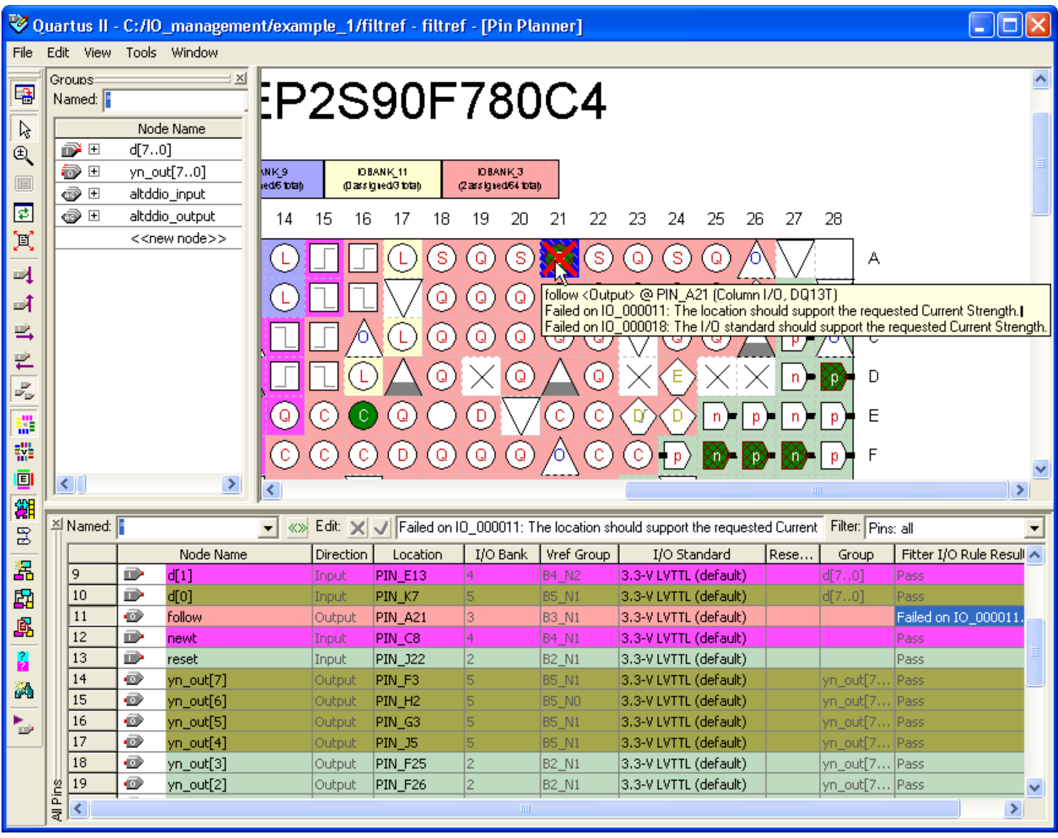
Figure 5–56 shows an example of error messages reported by I/O Assignment Analysis.

Figure 5–56. Error Message Report by I/O Assignment Analysis

```
Error: Current Strength logic option is set to 10mA for pin follow, but setting  is not supported by I/O standard 3.3-V LVTTL
Error: Pin AF3 does not support I/O standard 3.3-V LVTTL with Current Strength 10mA for location follow
Error: Can't fit design in device
Error: Quartus II I/O Assignment Analysis was unsuccessful. 3 errors, 1 warning
```

Fitter messages can also be seen in the Package View (Figure 5–57). Right click in the Package View and click **Show Fitter Placements** to see the failing pins. The failing pins are shown with a cross sign, as shown in Figure 5–57. A tool tip is displayed when the mouse cursor is pointed over the failing pin. The tooltip displays the failed I/O rules.

Figure 5–57. Tool Tip



You can correct the I/O Assignment Analysis failure shown for the pin in Figure 5–56 and Figure 5–57 easily by setting the proper current drive strength for the I/O standard assigned for that pin. Current drive strength can be set in the Assignment Editor using the “Current Drive Strength” assignment.

For more information about the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

The effectiveness of I/O Assignment Analysis is relative to the completeness of your pin-related assignments and design. To ensure your design functions correctly, include all pin-related assignments and as many design files as possible in your Quartus II project.

Table 5–5 on page 5–71 and Table 5–6 on page 5–72 list a subset of the I/O rule checks performed when you run I/O Assignment Analysis with and without design files.

For more detailed information about each I/O rule, refer to the appropriate device handbook.

Table 5–5. Examples of I/O Rule Checks *(Note 1)*

Rule	Description	Device Families	HDL Required?
I/O bank capacity	Checks the number of pins assigned to an I/O bank against the number of pins allowed in the I/O bank.	All	No
I/O bank V_{CCIO} voltage compatibility	Checks that no more than one V_{CCIO} is required for the pins assigned to the I/O bank.	All	No
I/O bank VREF voltage compatibility	Checks that no more than one VREF is required for the pins assigned to the I/O bank.	All	No
I/O standard and location conflicts	Checks whether the pin location supports the assigned I/O standard.	All	No
I/O standard and signal direction conflicts	Checks whether the pin location supports the assigned I/O standard and direction. For example, certain I/O standards on a particular pin location can only support output pins.	All	No
Differential I/O standards cannot have open drain turned on	Checks that open drain is turned off for all pins with a differential I/O standard.	All	No
I/O standard and drive strength conflicts	Checks whether the drive strength assignments are within the specifications of the I/O standard.	All	No
Drive strength and location conflicts	Checks whether the pin location supports the assigned drive strength.	All	No
`BUSHOLD` and location conflicts	Checks whether the pin location supports `BUSHOLD`. For example, dedicated clock pins do not support `BUSHOLD`.	All	No
`WEAK_PULLUP` and location conflicts	Checks whether the pin location supports `WEAK_PULLUP` (for example, dedicated clock pins do not support `WEAK_PULLUP`)	All	No
Electromigration check	Checks whether combined drive strength of consecutive pads exceeds a certain limit. For example, the total current drive for 10 consecutive pads on a Stratix II device cannot exceed 200 mA.	All	No
`PCI_IO` clamp diode, location, and I/O standard conflicts	Checks whether the pin location along with the I/O standard assigned supports `PCI_IO` clamp diode.	All	No
SERDES and I/O pin location compatibility check	Checks that all pins connected to a SERDES in your design are assigned to dedicated SERDES pin locations.	All	Yes
PLL and I/O pin location compatibility check	Checks whether pins connected to a PLL are assigned to the dedicated PLL pin locations.	All	Yes

Note to Table 5–5:

(1) The supported device families are: Arria® GX, Stratix III, Stratix II, Stratix II GX, Stratix, Stratix GX, Cyclone III, Cyclone II, Cyclone, HardCopy, and MAX II devices.

Table 5–6. SSN-Related Rules

Rule	Description	Device Families (1)	HDL Required?
I/O bank can not have single-ended I/O when DPA exists	Checks that no single-ended I/O pin exists in the same I/O bank as a DPA.	Stratix II Stratix GX	No
A PLL I/O bank does not support both a single-ended I/O and a differential signal simultaneously	Checks that there are no single-ended I/O pins present in the PLL I/O Bank when a differential signal exists.	Stratix II	No
Single-ended output is required to be a certain distance away from a differential I/O pin	Checks whether single-ended output pins are a certain distance away from a differential I/O pin.	All	No
Single-ended output has to be a certain distance away from a VREF pad	Checks whether single-ended output pins are a certain distance away from a VREF pad.	Cyclone II Cyclone	No
Single-ended input is required to be a certain distance away from a differential I/O pin	Checks whether single-ended input pins are a certain distance away from a differential I/O pin.	Cyclone II Cyclone	No
Too many outputs or bidirectional pins in a VREFGROUP when a VREF is used	Checks that there are no more than a certain number of outputs or bidirectional pins in a VREFGROUP when a VREF is used.	All	No
Too many outputs in a VREFGROUP	Checks whether too many outputs are in a VREFGROUP.	All	No

Note to Table 5–6:

(1) "All" includes the following device families: Arria GX, Stratix III, Stratix II, Stratix II GX, Stratix, Stratix GX, Cyclone III, Cyclone II, Cyclone, HardCopy, and MAX II devices.

Scripting Support

A Tcl script allows you to run procedures and determine settings described in this chapter. You can also run some of these procedures at a command prompt.

For detailed information about specific scripting command options and Tcl API packages, type the following command at a system command prompt to run the Quartus II Command-Line and Tcl API Help browser:

```
quartus_sh --qhelp ↵
```

For more information about Quartus II scripting support, including examples, refer to the *Tcl Scripting* and *Command-Line Scripting* chapters in volume 2 of the *Quartus II Handbook*.

Running the I/O Assignment Analysis

You can run I/O Assignment Analysis with a Tcl command or with a command run at a command prompt. For more information about running the I/O Assignment Analysis, refer to "Understanding the I/O Assignment Analysis Report and Messages" on page 5–67.

Enter the following in a Tcl console or script:

```
execute_flow -check_ios ↵
```

Type the following at a (non-Tcl) system command prompt:

```
quartus_fit <project name> --check_ios ↵
```

Generating a Mapped Netlist

You can generate a mapped netlist with a Tcl command or with a command-line command. For more information about generating a mapped netlist, refer to "Generating a Mapped Netlist" on page 5–65.

Enter the following in the Tcl console or in a script:

```
execute_module -tool map
```

The `execute_module` command is in the flow package.

Type the following at a system command prompt:

```
quartus_map <project name> ↵
```

Reserving Pins

Use the following Tcl command to reserve a pin:

```
set_instance_assignment -name RESERVE_PIN <value> -to <signal name>
```

For more information about reserving pins, refer to "Reserving Pins" on page 5–65.

Valid values are:

- `"AS BIDIRECTIONAL"`
- `"AS INPUT TRI-STATED"`
- `"AS OUTPUT DRIVING AN UNSPECIFIED SIGNAL"`
- `"AS OUTPUT DRIVING GROUND"`
- `"AS SIGNALPROBE OUTPUT"`

Include the quotes when specifying the value.

Location Assignments

Use the following Tcl command to assign a signal to a pin or device location.

```
set_location_assignment <location> -to <signal name> ↵
```

For more information about location assignments, refer to "Location Assignments" on page 5–66.

Valid locations are pin location names, such as `PIN_A3`. The Stratix and Cyclone series of devices also support edge and I/O bank locations. Edge locations are `EDGE_BOTTOM`, `EDGE_LEFT`, `EDGE_TOP`, and `EDGE_RIGHT`. I/O bank locations include `IOBANK_1` up to `IOBANK_`*n*, in which *n* is the number of I/O banks in a particular device.

In Stratix III devices only, I/O bank names have the form `IOBANK_`*nx* where `n` is a number and *x* is the letter *A*, *B*, or *C*. Though I/O banks can share the same number with different letters, such as 1A and 1C, they are separate banks and not related to each other.

For more information, refer to the *Stratix III Device Handbook*.

Validating Pin Assignments after Full Compilation

If you used the Live I/O check feature during pin placements, many of the I/O assignments have been verified immediately as you made the assignment. There are some placement rules that are checked only during I/O assignment analysis and full compilation of your design. The Quartus II software validates I/O assignments at three levels. The first level checking is done with the Live I/O check feature ON. A more comprehensive level of checking is performed with I/O Assignments Analysis. The final I/O timing check is done when you fully compile your design. (To better understand I/O timing analysis, refer to "I/O Timing Analysis" on page 5–75.)

To avoid costly board re-spins, you must perform full validation with full compilation with complete design files and constraints. With timing information, the Quartus II Fitter makes intelligent placements and routing to achieve optimal timing performance in your design. Use the TimeQuest SDC editor to create timing constraints for inputs, outputs, and bidirectional pins. If you are using the Quartus II Classic Timing Analyzer, specify timing constraints on the **Classic Timing Analyzer Settings** page of the **Settings** dialog box (Assignments menu).

For more information about the Quartus II TimeQuest Timing Analyzer and the Classic Timing Analyzer, refer to the *Timing Analysis* section of volume 3 of the *Quartus II Handbook*.

Accepting Fitter Placements—Back-Annotating Assignments

To create assignments for all Fitter-placed pins into your project **.qsf** file, perform the following steps:

1. On the Processing menu, click **Start Compilation**, or on the Processing menu, point to **Start** and click **I/O Assignment Analysis**.
2. On the Assignments menu, click **Pin Planner**. The Pin Planner appears.
3. On the View menu, point to **Show** and click **Show Fitter Placements**. You can also access this command from the Pin Planner toolbar or on the right-click menu in the Package View.
4. Review the Fitter placements.
5. To create location assignments for these Fitter placements, perform the following steps:
   a. On the Assignments menu, click **Back-Annotate Assignments**. The **Back-Annotate Assignments** dialog box appears.
   b. Select **Pin & device assignments** (Figure 5–58).
   c. Click **OK**.

Figure 5–58. Back-Annotate Assignments Dialog Box

To create assignments for a selection of the Fitter-placed pins, perform the following steps:

1. On the Processing menu, click **Start Compilation**, or on the Processing menu, point to **Start**, and click **I/O Assignment Analysis**.
2. On the Assignments menu, click **Pin Planner**.
3. On the View menu, point to **Show** and click **Show Fitter Placements**. Review the placements.
4. In the Pin Planner, select one or more Fitter-placed pins for which you want to create assignments.
5. Right-click one of the selected pins and click **Back Annotate**.
6. On the File menu, click **Save Project**. The Assignments are written to the **.qsf** file.

For more information about how the Quartus II software writes and updates the **.qsf** file, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

I/O Timing Analysis

Timing analysis is usually run during a full compilation of your design or with early timing estimate runs. You can also run timing analysis independently after full compilation from the Processing menu. For example, if you change the slew rates or current strengths of some I/O pins as ECOs, you do not have to recompile the entire design, but only run timing analysis to verify the timing of your design.

As part of I/O planning in your FPGA design, you must understand I/O timing results in the Quartus II software that are reported after performing timing analysis on your design. If you have all your design files complete and have completed full compilation, all the timing checks related to I/O timing are covered during timing analysis of your design. Static timing analysis is performed when you compile your design in the Quartus II software. You must understand I/O timing and what factors affect I/O timing paths in your design. One important factor that counts greatly in I/O timing results is how accurately you specify the output loads at the output and bidirectional pins in your FPGA design. Incomplete I/O constraints can affect your I/O timing results.

The Quartus II software supports three different methods of I/O timing analysis:

- I/O timing using a default or user-specified capacitive load with no signal integrity analysis (default)

  The Quartus II TimeQuest Timing Analyzer and the Quartus II Classic Timing Analyzer create timing reports that measure t_{CO} to an I/O pin using a default or user-specified value for a capacitive load.

- The Quartus II software **Enable Advanced I/O Timing** option utilizing a user-defined board trace model to produce enhanced timing reports from accurate, "board-aware" simulation models

  The Quartus II software **Enable Advanced I/O Timing** option enables you to configure a complete board trace model for each I/O standard or pin used in your design. With **Enable Advanced I/O Timing** turned on, the Quartus II TimeQuest Timing Analyzer uses the results of simulations of the I/O buffer, package, and board trace model to generate more accurate I/O delays and extra reports to give insight into signal behavior at the system level. You can use these advanced timing reports as a guide to make changes to your I/O assignments and board design to improve timing and signal integrity.

- Full board routing simulation in third-party tools using Altera-provided or Quartus II software generated IBIS or HSPICE I/O models

  The creation of simulation model files for use by third-party board simulation tools is achieved with the IBIS and HSPICE Writers. The IBIS and HSPICE Writers in the Quartus II software can export accurate simulation models for use in applications such as Mentor Graphics HyperLynx and Synopsys HSPICE.

This section describes the first and second methods.

I/O timing using a default or user-specified capacitive load is not supported for Stratix III and Cyclone III devices. Use the **Enable Advanced I/O Timing** option for Stratix III and Cyclone III devices.

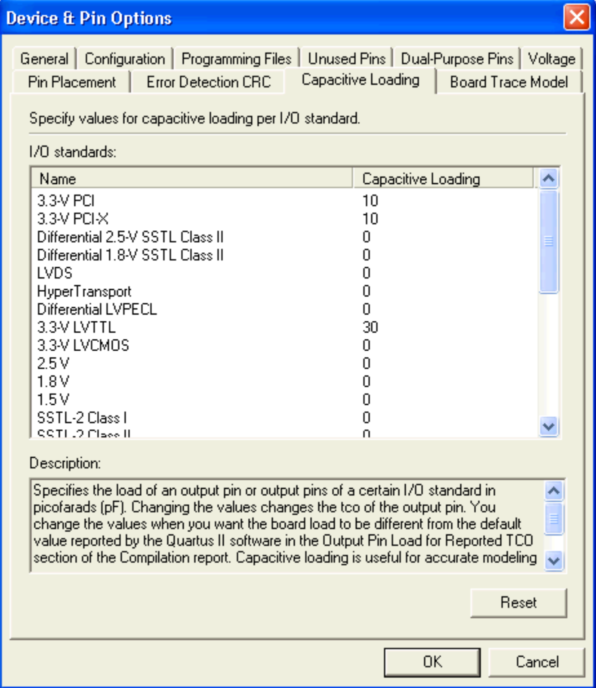
For information about creating IBIS and HSPICE models with the Quartus II software and integrating those models into HyperLynx and HSPICE simulations, refer to the *Signal Integrity Analysis with Third Party Tools* chapter in volume 2 of the *Quartus II Handbook*.

© March 2009 Altera Corporation

I/O Timing and Power with Capacitive Loading

When calculating t_{CO} and power for output and bidirectional pins, the Quartus II TimeQuest Timing Analyzer, the Quartus II Classic Timing Analyzer, and power analysis use a bulk capacitive load. This is the default method for these pins. You can adjust the value of the capacitive load per I/O standard to get t_{CO} and power measurements that more accurately reflect the behavior of the output or bidirectional net on your PCB. Input pins ignore this setting. To adjust the value of the capacitive load, on the Assignments menu, click **Device**. Click **Device & Pin Options** and click the **Capacitive Loading** tab (Figure 5–59).

Figure 5–59. Capacitive Tab of the Device and Pin Options Dialog Box



All of the available I/O standards for your selected device are listed with their default loading values in picofarads (pF). Adjust the loading values as desired for the I/O standards used in your design. Power and t_{CO} measurements in the Compilation Report are adjusted based on the settings.

You can also adjust the load on any individual pin in the Groups list or All Pins list in the Pin Planner by adding the **Output Pin Load** column. Right-click anywhere in either list and select **Customize Columns**. Select **Output Pin Load** from the list of available custom columns and add it to the list of visible columns. You can customize the load for individual pins or multiple pins with different I/O standards.

For more information about capacitive loading, the devices that support it, and how t_{CO} and power are adjusted based on the setting, refer to the Quartus II Help.

Advanced I/O Timing in the Quartus II Software

As part of I/O planning, especially with high-speed designs, you should take board-level signal integrity and timing into account. When adding an FPGA device with high-speed interfaces to a board design, the quality of the signal at the far end of the board route, as well as the propagation delay in getting there, is vital for proper system operation.

Enabling and Configuring Advanced I/O Timing

With the Quartus II software **Enable Advanced I/O Timing** option turned on, you can expand upon the basic timing and power measurements made with the **Capacitive Loading** settings. The **Enable Advanced I/O Timing** option gives you the ability to fully define not only the capacitive load, but also any termination components and trace impedances in the board routing for any output pin or bidirectional pin in output mode. You can configure an overall board trace model for each I/O standard as well as customize the model for specific pins using a graphical interface.

When the **Enable Advanced I/O Timing** option is turned on, the board trace model replaces the **Capacitive Loading** tab settings because the load is included in the model. For timing measurements, the entire board trace model is taken into account when calculating I/O delays. For power measurements, an effective capacitive load is used based on the sum of the capacitive elements in the model. This includes the **Near capacitance**, **Far capacitance**, and **Transmission line distributed capacitance** elements of the model.

For Stratix III and Cyclone III devices, advanced I/O timing is the only way to measure I/O timing. Advanced I/O timing is supported for Stratix II devices also. All other devices use capacitive loading for I/O t_{CO} and power measurements. Check the Altera website at www.altera.com to determine which devices are supported in newer versions of the Quartus II software.

Before you configure a board trace model for advanced I/O timing, you must turn on **Enable Advanced I/O Timing** if your selected device supports it. All devices in each supported family work with advanced I/O timing.

Turn on **Enable Advanced I/O Timing**. If the **Settings** dialog box is not currently open, on the Assignments menu, click **Settings**. In the **Category** list, click the “+” icon to expand **Timing Analysis Settings**. Select **TimeQuest Timing Analyzer**. The **TimeQuest Timing Analysis** page appears. Turn on **Enable Advanced I/O Timing**.

For Stratix III and Cyclone III devices, the **Enable Advanced I/O Timing** option is turned on by default and is always performed when you run the Quartus II TimeQuest Timing Analyzer.

Define Overall Board Trace Models

You can now define an overall board trace model for each I/O standard in your design. This is the default model for all pins that use a particular I/O standard. After configuring the overall board trace model, customize the model for specific pins using the Board Trace Model view in the Pin Planner.

With the **Settings** dialog box open, in the **Category** list, click **Device**. Click **Device & Pin Options** and click the **Board Trace Model** tab (Figure 5–60).

Figure 5–60. Board Trace Model Tab of the Device and Pin Options Dialog Box

You can still click the **Capacitive Loading** tab. However, because you can configure all capacitive loading settings as part of the board trace model, the tab indicates that you must use the settings in the **Board Trace Model** tab.

All of the I/O standards available to the device are listed. Select any I/O standard from the list. The **Board trace model** list displays the names and values of all configurable components of the board trace for the selected I/O standard. Components of the model are initially set to **short**, **open**, or a numeric value depending on the component. The default settings for components in the model for each I/O standard are device-specific and match the default test model used for calculating delay when the **Enable Advanced I/O Timing** option is turned off. In this way, default delay measurements are the same whether or not the **Enable Advanced I/O Timing** option is used.

For information about the default models used for measuring I/O delay, refer to the *DC & Switching Characteristics* chapter in the relevant device handbook.

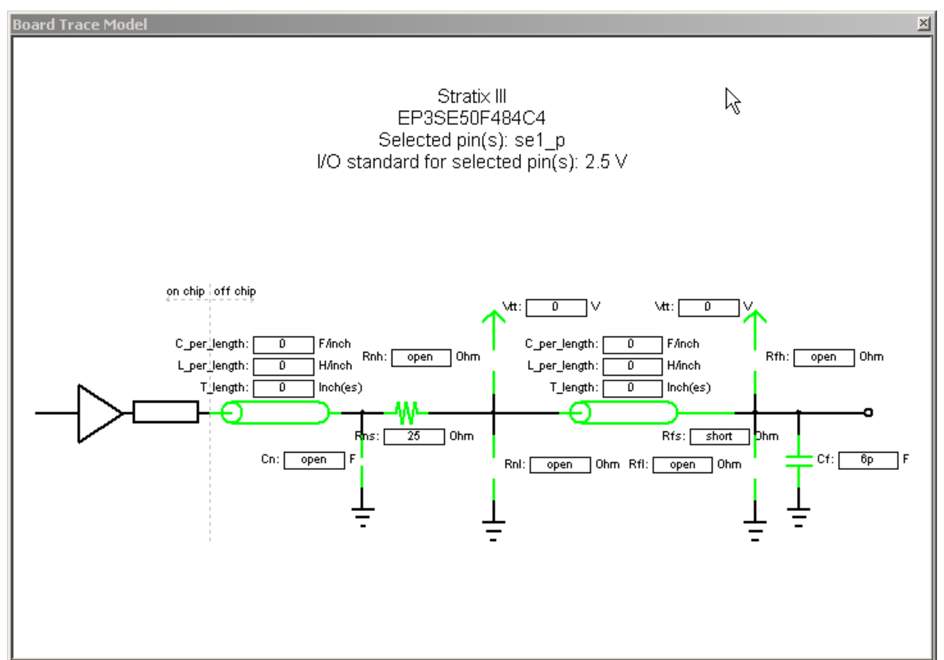
All of the component values listed in Figure 5–60 are adjustable. For differential I/O standards, the component values you set are used for both the positive and negative signals of a differential pair. An additional component, **Far differential resistance**, is also included. To reset individual settings to their defaults, leave the setting blank. If you want all the settings for an I/O standard to revert to their original settings, click **Reset**. Click **OK** to close the **Device & Pin Options** dialog box. Click **OK** again to close the **Settings** dialog box.

Any component value changes made in the **Board Trace Model** tab for a particular I/O standard are reflected in the Board Trace Model view in the Pin Planner of all pins assigned with the same I/O standard (described in "Customize the Board Trace Model in the Pin Planner"). However, custom component value changes made to selected pins in the Board Trace Model view in the Pin Planner take priority and are not affected by changes made to an I/O standard in the **Board Trace Model** tab.

Customize the Board Trace Model in the Pin Planner

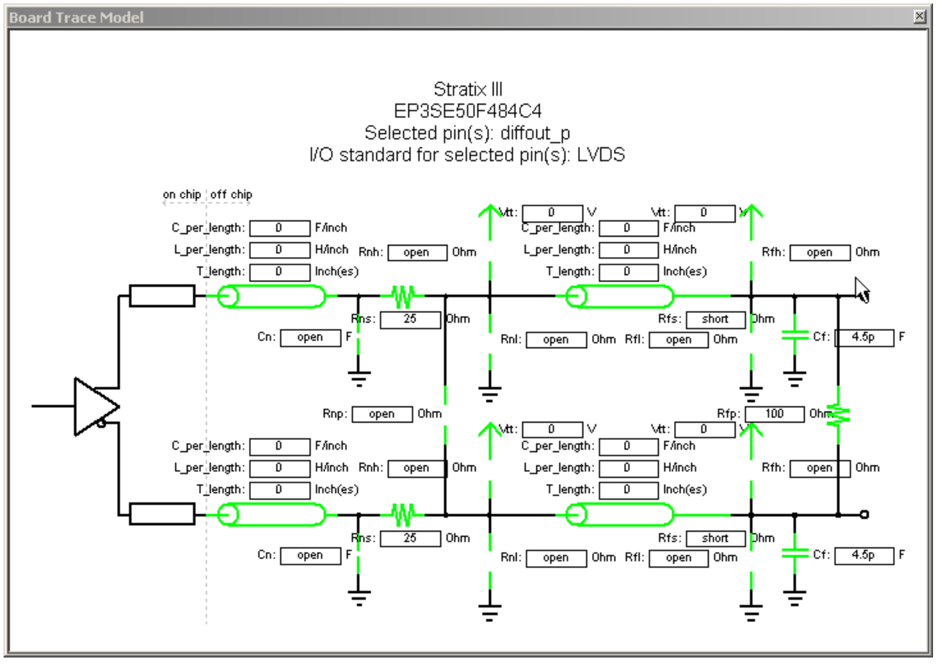
In addition to the views available in the Package View in the Pin Planner, you can also view a graphical representation of the board trace model you have configured using the Board Trace Model view. To open the Board Trace Model view, right-click on an output or bidirectional pin in the Groups list, All Pins list, or Package View and click **Board Trace Model**. The Board Trace Model view opens in a floating window (Figure 5–61).

Figure 5–61. Board Trace Model View



For differential signals, the Board Trace Model view displays the routing and components for both the positive and negative signals of the differential pair (Figure 5–62).

Figure 5–62. Differential Board Trace Model View



Any changes made to the Board Trace Model view for a differential signal pair must be performed on the positive signal of the pair. The settings must match between the positive and negative signals of a differential pair, so the changes are automatically reflected in the settings for the negative signal.

Double-click a component value to edit it. For numerical values, use standard unit prefixes such as *p*, *n*, and *k* to represent pico, nano, and kilo, respectively. To short a series component or have an open circuit for a parallel component, double-click the component value and select **short** or **open** from the list.

All the assignments for board trace models you specify in the schematic are saved to the Quartus II Settings File (**.qsf**). You can also enter these Tcl assignment commands in the **.qsf** to specify the board trace parameters for an output or bidirectional pin. The examples in Example 5–8 use Tcl assignments to specify board trace models.

Example 5–8. Specifying Board Trace Models

```
## setting the near end series resistance model of sel_p output pin to 25 ohms
set_instance_assignment -name BOARD_MODEL_NEAR_SERIES_R 25 -to sel_p
## Settting the far end capacitance model for sel_p output signal to 6 picofarads
set_instance_assignment -name BOARD_MODEL_FAR_C 6P -to sel_
```

For more details about configuring component values for a board trace model, including a complete list of the supported unit prefixes and setting the values using Tcl scripts, refer to the Quartus II Help.

To view a display of the model for a particular pin, in the Package View, Groups list, or All Pins list, click on the pin. This changes the Board Trace Model view to display the model of the pin. To select multiple pins that share the same I/O standard, open the Board Trace Model view and edit the model for all of the selected pins. If an input pin or multiple pins with different I/O standards are selected, the Board Trace Model view window indicates that it cannot display the model for the selected pin or pins.

The components in the Board Trace Model view correspond to the components listed in the **Board Trace Model** tab directly and the settings match initially. You can click and edit any value in the Board Trace Model view to customize the model for the selected pin or pins. Changes made in the Board Trace Model view do not affect the settings in the **Board Trace Model** tab.

To configure board trace models for the pins in your design efficiently with these two methods of entry, define the model for each I/O standard in the **Board Trace Model** tab. With the overall model defined, use the Board Trace Model view in the Pin Planner to customize individual pins as required. These customizations take priority over the settings in the **Board Trace Model** tab on a per pin and per model component basis, so they do not affect the settings on any other pin.

Create Signal Integrity Result Reports

After you have turned on **Enable Advanced I/O Timing** and configured board trace models for the pins you want to analyze, compile your project or run the Quartus II TimeQuest Timing Analyzer after a full compilation. The **Enable Advanced I/O Timing** option creates signal integrity subreports under TimeQuest Timing Analyzer in the Compilation Report window.

The Board Trace Model Assignments report (Figure 5–63) summarizes the board trace model component settings for each output and bidirectional signal.

Figure 5–63. Board Trace Model Assignments Report

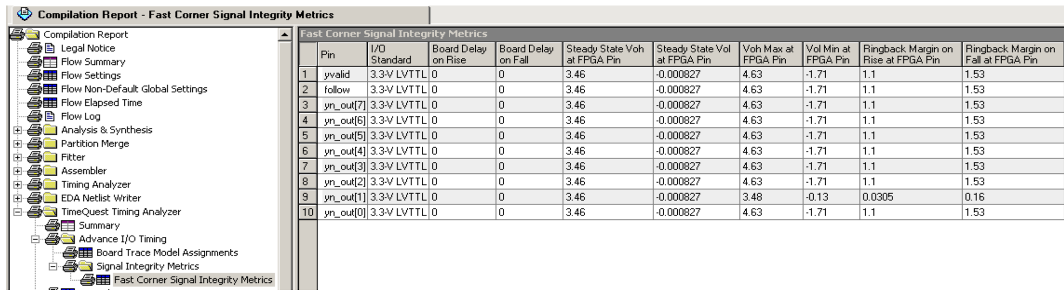
Compilation Report - Board Trace Model Assignments

Board Trace Model Assignments

	Pin	I/O Standard	Near Series R	Near Pull-up R	Near Pull-down R	Near C	Tline Length	Tline L per Length	Tline C per Length	Far Series R	Far Pull-up R	Far Pull-down R
1	yvalid	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
2	follow	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
3	yn_out[7]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
4	yn_out[6]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
5	yn_out[5]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
6	yn_out[4]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
7	yn_out[3]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
8	yn_out[2]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
9	yn_out[1]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open
10	yn_out[0]	3.3-V LVTTL	short	open	open	open	0	0	0	short	open	open

The Signal Integrity Metrics subfolder contains detailed reports listing all of the metrics calculated by the **Enable Advanced I/O Timing** option (Figure 5–64).

Figure 5–64. Example of Slow-Corner Signal Integrity Metrics Report



	Pin	I/O Standard	Board Delay on Rise	Board Delay on Fall	Steady State Voh at FPGA Pin	Steady State Vol at FPGA Pin	Voh Max at FPGA Pin	Vol Min at FPGA Pin	Ringback Margin on Rise at FPGA Pin	Ringback Margin on Fall at FPGA Pin
1	yvalid	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
2	follow	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
3	yn_out[7]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
4	yn_out[6]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
5	yn_out[5]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
6	yn_out[4]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
7	yn_out[3]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
8	yn_out[2]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53
9	yn_out[1]	3.3-V LVTTL	0	0	3.46	-0.000827	3.48	-0.13	0.0305	0.16
10	yn_out[0]	3.3-V LVTTL	0	0	3.46	-0.000827	4.63	-1.71	1.1	1.53

The Slow- and Fast-Corner Signal Integrity Metrics reports are generated by the **Enable Advanced I/O Timing** option. They list, in tabular format, all of the signal integrity metrics calculated by the **Enable Advanced I/O Timing** option, based on the board trace model settings for each output or bidirectional pin. The reports contain many metrics, including measurements at both the FPGA and at the far-end load of board delay, steady state voltages, and rise and fall times.

The Slow- or Fast-Corner Signal Integrity Metrics reports are generated depending on the **Timing Netlist** option in the Quartus II TimeQuest Timing Analyzer. To select whether to create a slow- or a fast-corner report, in the TimeQuest Timing Analyzer on the Netlist menu, click **Create Timing Netlist**. Under **Delay model**, select **Slow corner** or **Fast corner** to create reports of that type.

For complete descriptions of all of the metrics calculated when the **Enable Advanced I/O Timing** option is turned on and diagrams illustrating the metrics on output waveforms, refer to the Quartus II Help. For more information about board-level signal integrity and tips on how to improve signal integrity in your high-speed designs, refer to the Altera Signal Integrity Center.

For information about the configuration and use of the Quartus II TimeQuest Timing Analyzer, refer to the Quartus II Help or *Section III: Timing Analysis* in volume 3 of the *Quartus II Handbook*.

Incorporating PCB Design Tools

Signal and pin assignments are initially made by the FPGA or ASIC designer and it is up to the board designer to transfer these assignments to the symbols used in their system circuit schematics and board layout correctly. As the board design progresses, pin reassignments might be requested or required to optimize the layout. These reassignments must in turn be relayed to the FPGA designer, so that the new assignments can be validated with the I/O Assignment Analyzer and processed through an updated place-and-route of the FPGA.

The Quartus II software interacts with board layout tools by importing and exporting pin information files, including the **.qsf**, **.pin**, and **.fx** files.

For more information about incorporating PCB design tools, refer to the *Cadence PCB Design Tools Support* and *Mentor Graphics PCB Design Tools Support* chapters in volume 2 of the *Quartus II Handbook*.

Conclusion

The Quartus II software provides many tools and features to help you with the I/O planning process. The I/O assignment analysis process offers the ability to validate pin assignments in all design stages, even before the development of the design. The ability to import and export assignments between the Quartus II software and other PCB tools also enables you to make iterative changes efficiently. Finally, the ability to enter a board trace model and create advanced timing reports based on how I/O signals are routed on a board truly makes the Quartus II software "board-aware."

Referenced Documents

The following documents were referenced in this chapter:

- *AN 90: SameFrame Pin-Out Design for FineLine BGA Packages*
- *AN 315: Guidelines for Designing High-Speed FPGA PCBs*
- *AN 366: Understanding I/O Output Timing in Altera Devices*
- *AN 476: Impact of I/O Settings on Signal Integrity in Stratix III Devices*
- *Altera Device Package Information Datasheet*
- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*
- *Cadence PCB Design Tools Support* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Data (DQ) and Data Strobe (DQS) Megafunction User Guide (ALTDQ and ALTDQS)*
- *DC & Switching Characteristics* chapter in volume 1 of the *Stratix II Device Handbook*
- *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*
- *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*
- *Mentor Graphics PCB Design Tools Support* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Support for HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of *Quartus II Handbook*
- *Signal Integrity Analysis with Third Party Tools* chapter in volume 2 of the *Quartus II Handbook*
- *Simultaneous Switching Noise Analysis (SSN) and Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Stratix III Device Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Section III: Timing Analysis* in volume 3 of the *Quartus II Handbook*

© March 2009 Altera Corporation

Document Revision History

Table 5–7 shows the revision history for this chapter.

Table 5–7. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Updated "Pad View Window" on page 5–26 ■ Added new figures: → Figure 5–20 → Figure 5–21 ■ Added new section "SSN Visualization View" on page 5–34 ■ Added new section "Creating Exclusive Group Assignments" on page 5–37	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	■ Changed to 8½" x 11" page size. ■ Reorganized chapter ■ Updated the following sections: → "I/O Planning Overview" on page 5–5 → "Early I/O Planning Using the Pin Planner" on page 5–8 → "Create or Import a Megafunction or IP MegaCore Variation from the Pin Planner" on page 5–9 → "Configure Nodes" on page 5–10 → "I/O Analysis for Designs with Pins Only" on page 5–14 → "Using the Pin Planner" on page 5–20 → "Validating Pin Assignments after Full Compilation" on page 5–76 → "Accepting Fitter Placements—Back-Annotating Assignments" on page 5–76 → "I/O Timing Analysis" on page 5–77	Updated for the Quartus II software version 8.1 release.

Table 5–7. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
May 2008 v8.0.0	■ Reorganized chapter ■ Updated links ■ Updated the following sections: →“Importing and Exporting Pin Assignments” on page 5–16 →“I/O Planning Overview” on page 5–5 →“Creating Pin-Related Assignments” on page 5–19 →“Using Hardware Description Language (HDL)” on page 5–54 →“Validating Pin Assignments” on page 5–56 →“I/O Timing Analysis” on page 5–76 →“.pin File” on page 5–18 →“Import a Megafunction or IP MegaCore Variation from the Pin Planner” on page 5–13 →“Using the Live I/O Check Feature to Validate Pin Assignments” on page 5–57 →“Pin Migration View” on page 5–30 →“altera_attribute” on page 5–55 →“Assigning a Location for Differential Pins” on page 5–37 →“Synthesis Attributes” on page 5–55 ■ Added the following sections: →“Swapping Pin Locations” on page 5–42 →“Using Low-Level I/O Primitives” on page 5–56 →“Advanced I/O Timing in the Quartus II Software” on page 5–78 →“Enabling and Configuring Advanced I/O Timing” on page 5–78 ■ Updated figures to reflect updates to the Quartus II software	Updated for the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

6. Simultaneous Switching Noise (SSN) Analysis and Optimizations

QII52018-9.0.0

Introduction

FPGA design has evolved from small programmable circuits to competing with multimillion-gate ASICs. At the same time, the I/O counts on FPGAs and logic density requirements have increased exponentially. Higher-speed interfaces in FPGAs that include high-speed serial interfaces and memory interfaces require careful interface design on the PCB. The timing and signal integrity requirements of these interfaces are pushing designers to address the effects early in the design development cycle. One example is the effect of simultaneous switching noise (SSN). SSN is defined as a noise voltage induced onto a victim I/O pin of a device due to the switching behavior of other aggressor I/O pins in the device. SSN noise often leads to the degradation of signal integrity by causing signal distortion, thereby reducing the noise margin of a system.

Today's complex FPGA system design is incomplete without addressing the integrity of signals coming in and out of the FPGA. SSN is one of the signal integrity problems faced during FPGA design. Altera recommends that you perform SSN analysis early in your FPGA design and tape out your PCB with complete SSN analysis of your FPGA in the Quartus® II software. This chapter describes the SSN Analyzer and Optimization tool introduced in the Quartus II software version 9.0 and covers the following topics:

- "Definitions"
- "Understanding SSN and Its Effects" on page 6–2
- "SSN Estimation Tools from Altera" on page 6–5
- "Design Factors Affecting SSN Results" on page 6–5
- "Using the SSN Analyzer in the Quartus II Software" on page 6–5
- "SSN Analyzer Usage Models" on page 6–14
- "Scripting Support" on page 6–18
- "Run Time Considerations in SSN Analysis" on page 6–18
- "SSN Optimization" on page 6–20

Definitions

The terminology used in this chapter includes:

Aggressor: Output or bidirectional signal that contributes to the noise for a victim.

PDN: Power Distribution Network

QH: Quiet High signal level on a pin

QHN: Quiet High Noise on a pin in volts

QL: Quiet Low signal level on a pin

QLN: Quiet Low Noise on a pin in volts

SI: Signal Integrity (a superset of SSN covering all noise sources)

SSN: Simultaneous Switching Noise

SSOs: Simultaneous Switching Outputs (which are either the output or bidirectional pins)

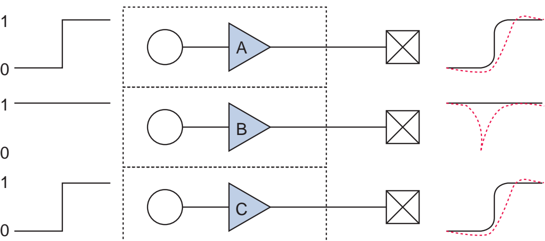
Victim: An input, output, or bidirectional pin that is analyzed during SSN analysis. During SSN analysis, each pin is analyzed as a victim. If it is an output or bidirectional pin, the same pin acts as an aggressor for other pins.

Understanding SSN and Its Effects

SSN is defined as a noise voltage induced onto a single victim I/O pin in your FPGA device. The voltage is induced by the switching behavior of other aggressor I/O pins in the device. SSN can be divided into two types of noise: voltage noise and timing noise.

Figure 6–1 shows a system with three pins. Two of the pins (A and C) are switching, while one pin (B) is quiet. If the pins are driven in isolation, the voltage waveforms at the output of the buffers appear as the solid curves at the left of the figure. However, when the pins are switched simultaneously, the noise generated by pins A and C switching is injected onto the other pins, manifesting itself as a voltage noise on pin B and a timing noise on pins A and C, as shown by the dotted curves in the figure.

Figure 6–1. System with Three Pins



Voltage noise is measured as the worst-case change in voltage of a signal due to SSN. When a signal is quiet high (QH), it is measured as the change in voltage toward 0 V. When a signal is quiet low (QL), it is measured as the change in voltage toward V_{CC}.

In the Quartus II software, only voltage noise is analyzed. Voltage noise can be caused by SSOs under two worst-case conditions:

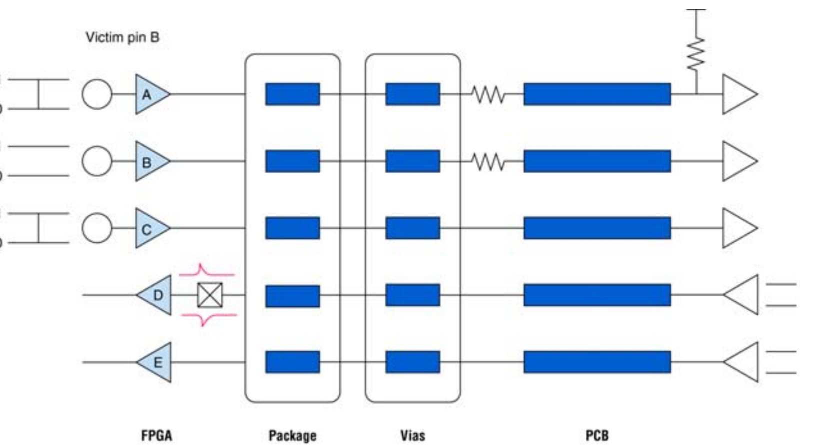
- Victim pin is high and aggressors (SSOs) are switching from low to high
- Victim pin is low and aggressors (SSOs) are switching from high to low

For outputs, the noise is computed at the far-end receiver as shown in Figure 6–2 for pin B.

Figure 6–2. Quiet High Output Noise Estimation

For inputs, the noise is computed at the FPGA bumps as shown in Figure 6–3 for pin D.

Figure 6–3. Quiet Low Input Noise Estimation



SSN can occur in any system, but the induced noise does not always result in failures. The voltage functional errors are caused by SSN on quiet victims only when the voltage values on the quiet pins change by a large enough voltage such that the logic listening to that signal reads a change in the logic value. For QH signals, noise events that cause the voltage on those signals to fall below V_{IH} are a voltage functional error. Similarly, for QL signals, noise events that cause the voltage to rise above V_{IL} are a voltage functional error (Figure 6–4). Because V_{IH} and V_{IL} are different for different I/O standards and signals have different quiet voltage values, the absolute amount of SSN in volts cannot be used to determine if a voltage failure occurs. Instead, to quantify whether an SSN event will cause a voltage error, the Quartus II software uses the amount of noise as a percent of signal margin when reporting noise margins in SSN analysis (Figure 6–4).

Figure 6–4. Reporting Noise Margins

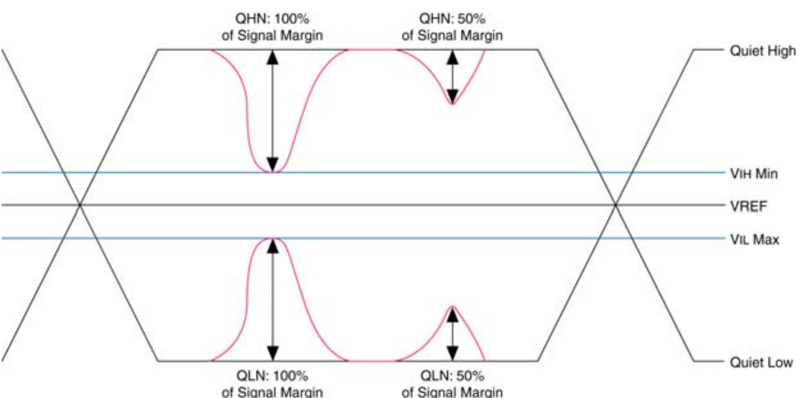
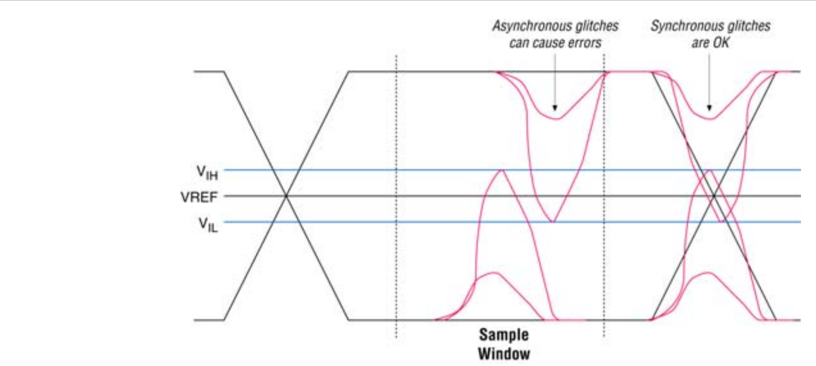


Figure 6–4 shows four noise events, two on QH signals and two on QL signals. The two noise events on the right-side of the figure consume 50% of the signal margin and do not cause voltage functional errors. However, the two noise events on the left side of the figure consume 100% of the signal margin and can cause a voltage functional error.

Another situation where a voltage noise cannot result in an error in your system is when the voltage noise happens synchronously, as shown in Figure 6–5. If noise or glitches, caused by aggressors, are synchronously related to the victim and occur outside of the sampling window of a receiver, the switching time of a victim can be affected but should not be considered as an input threshold violation failure.

Figure 6–5. Synchronous Voltage Noise



When you perform SSN analysis in the Quartus II software, there are a number of design factors that affect the noise margins. These design factors are described in "Design Factors Affecting SSN Results".

SSN Estimation Tools from Altera

Not addressing SSN early in your FPGA design and PCB layout could result in a respin of your board and lost time, which can impact your time to market. Therefore, the best approach is to address SSN early in your system design. Altera provides many tools for SSN analysis and estimation, including SSN characterization reports, an Early SSN Estimator (ESE) tool, and the SSN Analyzer in the Quartus II software. The ESE tool is available for various device families.

You can get more information on the spreadsheet tool and device support at Altera's Signal Integrity Center.

The SSN Analyzer and Optimization tool is available in the Quartus II software version 9.0 and later. In the Quartus II software version 9.0, only Stratix® III devices are supported. To get the latest information on device support for the SSN Analyzer with the Quartus II software you are using, refer to its help.

The ESE tool is a good starting point with which to estimate SSN in your FPGA design. To get more accurate results, you must use the SSN Analyzer tool in the Quartus II software to analyze SSN. Table 6–1 compares some of the differences between the SSN spreadsheet tool and the SSN Analyzer tool.

Table 6–1. Comparison of SSN Spreadsheet Tool and SSN Analyzer Tool

SSN Spreadsheet Tool	SSN Analyzer Tool
Is not integrated with the Quartus II software.	Integrated with the Quartus II software, allowing you to perform what-if analysis for SSN while making I/O assignment changes in the Quartus II software.
QL and QH levels are computed assuming a worst case pattern of I/O placements.	QL and QH levels are computed based on the I/O placements provided by the user or Fitter.
No support for entering board information.	Supports board trace models and board layer information that result in a more accurate SSN analysis.
No visualization feature.	Integrated with the Quartus II software Pin Planner, in which an SSN map shows the QL and QH levels on victim pins.
Good for doing an early SSN estimate. Does not require you to use the Quartus II software for early SSN estimate.	Requires you to create a Quartus II software project and provide the top-level port information.

Design Factors Affecting SSN Results

To understand what contributes to SSN voltage noise in your FPGA design, refer to *AN 472: Stratix II GX SSN Design Guidelines* and *AN 508: Cyclone III Simultaneous Switching Noise (SSN) Design Guidelines*.

Using the SSN Analyzer in the Quartus II Software

The SSN Analyzer introduced in the Quartus II software version 9.0 enables you to estimate the SSN (QLN and QHN) levels for your FPGA pins. The SSN optimization feature helps you optimize your design for SSN when you are compiling your design. The following sections explain the user requirements and how to use the tool to get the results you want.

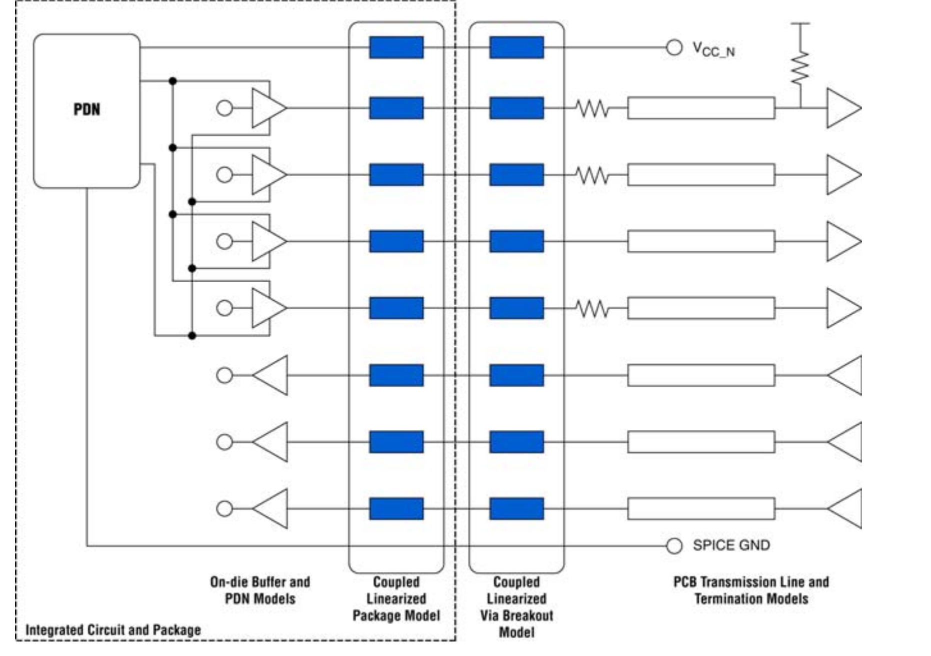
Tools Overview

The Altera® Quartus II SSN Analyzer gives you much flexibility in precisely defining the system to obtain accurate SSN results. Because the SSN Analyzer is integrated into the Quartus II software, it can automatically set up a system topology that matches a given Quartus II design. The tool accounts for different I/O standards and slew rate settings for each buffer in the design, as well as modelling different board traces for each signal. Furthermore, it correctly models the state of the unused pins in the design. These features leverage the previously existing Quartus II Advanced I/O Timing (AIOT) assignments that allowed custom board traces to be specified in the Quartus II software.

The SSN Analyzer tool also models the package and vias in the design. Models for the different packages that Altera FPGAs support are integrated into the Quartus II software. With respect to the via models, the tool supports the ability to specify different layers on which signals break out, each with its own thickness, and then specify which signal breaks out on which layer.

After automatically constructing the correct circuit topology as shown in Figure 6–6, the SSN Analyzer uses a simulation-based methodology to determine the SSN for each victim pin in the design.

Figure 6–6. Circuit Topology for SSN Analysis



I/O Standards Supported in the Quartus II SSN Analyzer

Altera device families support a wide range of I/O standards. To learn more about the I/O standards, refer to the device handbooks at www.altera.com. The Quartus II SSN Analyzer supports most I/O standards in a device family, such as LVTTL, LVCMOS, HSTL, and SSTL. Differential standards, such as LVDS and its variations, are not supported, because these standards contribute a small amount of SSN, which needs to be accounted for in your design.

For more information about the I/O standard support with your version of the Quartus II software, refer to the Quartus II Help.

Tool Inputs

The SSN Analyzer uses circuit models while performing SSN analysis. As shown in Figure 6–6, the circuit topology is incomplete if board trace information and layer information are not entered. To compute the SSN accurately in your FPGA device, you must describe these parameters in your FPGA design. However, if you do not specify some or all of the board trace parameters and PCB layer information, the Quartus II software uses default parameters during SSN analysis. These default parameters are listed in the Confidence Metric report.

For more information about the Confidence Metric Report, refer to "Confidence Metric Details Report" on page 6–13.

Board Trace Models

The board trace models required for the SSN Analyzer include the board trace termination resistors, pin loads (capacitance), and transmission line parameters. You can enter the board circuit models, which are also known as board trace models in the Quartus II software. The board trace model settings are shared with the models used during AIOT analysis.

For more information about AIOT analysis, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

In the Quartus II software, you can specify the board trace models using the board trace schematic template, the Pin Planner's All Pins list, or Tcl assignments. For every I/O standard, there is a built-in template that you can use to fill transmission line parameters, far and near end resistances, far and near end capacitances, and more. To open the board trace model schematic, right-click a pin in the Pin Planner's All Pins list and click **Board Trace Model**. Alternatively, you can enter the parameters in the All Pins list columns. The parameters entered in the board trace model schematic or the Pin Planner's columns are saved as Tcl assignments in the **.qsf** file and are also used in Advanced I/O Timing analysis (AIOT) with TimeQuest in the Quartus II software. If you have already specified the board trace models for AIOT, the same parameters are used during SSN analysis. Following are some examples of Tcl assignments that are used to specify the transmission line parameters:

```
set_instance_assignment -name BOARD_MODEL_TLINE_L_PER_LENGTH "3.041E-7" -to e[0]
set_instance_assignment -name BOARD_MODEL_TLINE_LENGTH 0.1391 -to e[0]
set_instance_assignment -name BOARD_MODEL_TLINE_C_PER_LENGTH "1.463E-10" -to e[0]
```

You can also create or edit the Tcl assignment directly in the **.qsf** file. If you do not specify the board trace parameters, the Quartus II software uses default parameters during SSN analysis. The default parameters used are listed in the Confidence Metric Details Report.

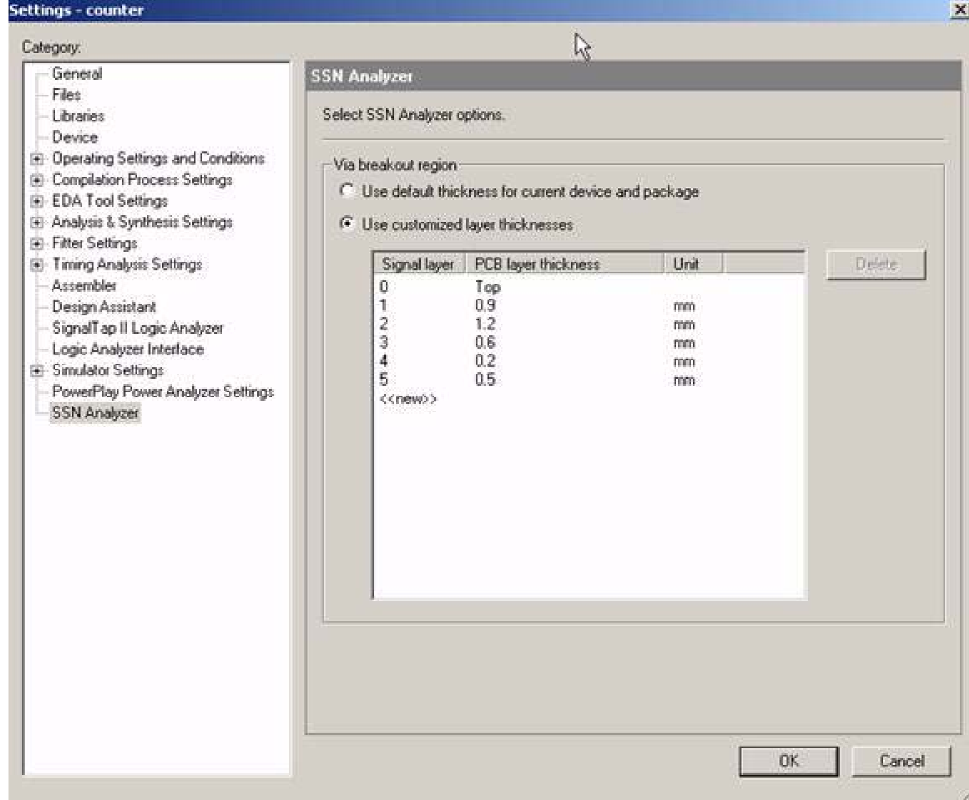
The best way to calculate transmission line parameters is to use a 2D-solver to estimate the line's inductance per inch and capacitance per inch. The termination resistor topology information can be obtained from the PCB schematics. The near-end and far-end pin load (capacitance) values can be obtained from the PCB schematic and other device data sheets. For example, if you know that an FPGA pin is driving a DIMM or some other package, you can get the loading information at the far end by looking at the data sheet of that device.

PCB Layers and PCB Layer Thickness

Every PCB board is fabricated using a number of layers. You can specify the number of layers and their thickness in the Quartus II software. The PCB layer information is used only during SSN analysis and is not required in other parts of the Quartus II software. The **SSN Analyzer** page in the **Settings** dialog box contains controls for setting values for a custom via breakout region (Figure 6–7). You can double-click on **<<new>>** to add a new signal layer. When you click on **<<new>>**, the signal layer number is automatically entered and you can enter a thickness value. To delete a layer, press the **Delete** key.

If a custom PCB breakout region is not described, you can select the default thickness, causing a single-layer PCB breakout region to be used during SSN analysis. Selecting the default option disables the maximum layer edit control and the list box.

Figure 6–7. Specifying the Number of Board Layers and Layer Thickness



Layers are numbered sequentially, starting from 0 (layer 0 is the top layer). Each layer has a thickness associated with it. You can specify the thickness on the **SSN Analyzer** page in the **Settings** dialog box. The **Unit** column shown in Figure 6–7 can be mils or millimeters (mm). For example, in Figure 6–7, layer 3 has a thickness of 0.6 mm. You must specify a thickness value greater than 0.

The Quartus II software saves the GUI settings as Tcl assignments in the **.qsf** file. You can also create or edit the Tcl assignments in the **.qsf** file. The number of layers is not fixed, but the layers must be consecutive. There is no maximum number of layers. For example, your **.qsf** file might contain the following assignments:

```
set_global_assignment -name PCB_LAYER_THICKNESS 0.00099822M -section_id 1
set_global_assignment -name PCB_LAYER_THICKNESS 0.00034036M -section_id 2
set_global_assignment -name PCB_LAYER_THICKNESS 0.00034036M -section_id 3
set_global_assignment -name PCB_LAYER_THICKNESS 0.00055372M -section_id 4
set_global_assignment -name PCB_LAYER_THICKNESS 0.00034036M -section_id 5
set_global_assignment -name PCB_LAYER_THICKNESS 0.00034036M -section_id 6
set_global_assignment -name PCB_LAYER_THICKNESS 0.00082042M -section_id 7
```

These statements tell the Quartus II software that there are 7 layers in the design. In each assignment, the letter `M` is the unit of thickness and stands for millimeters.

The assignments in the **.qsf** file must contain thickness information for consecutive layers. For example, suppose the **.qsf** file contains only the following assignments:

```
set_global_assignment -name PCB_LAYER_THICKNESS 0.00099822M -section_id 1
set_global_assignment -name PCB_LAYER_THICKNESS 0.00082042M -section_id 7
```

The Quartus II software generates the following error message when you start the SSN Analyzer:

```
Error: Nonconsecutive board stackup layers have been specified.
```

Signal Breakout Layers

Each user I/O pin in your FPGA device can break out at different layers on your PCB. In the Quartus II software, you can specify on which layers the I/O pins in your design break out. This information is only used during SSN analysis and is not required for other parts of the Quartus II software.

You can use the Pin Planner’s All Pins list or Tcl assignments to specify how pins break out. In the Pin Planner’s All Pins list, enter the layer number for the corresponding signal in the **PCB Layer** column. This action specifies the connection of that signal to that layer. The number you enter in the All Pins list is stored as a Tcl assignment in the **.qsf** file as follows:

```
set_instance_assignment -name PCB_LAYER 10 -to e[2]
set_instance_assignment -name PCB_LAYER 3 -to e[3]
```

You can also specify this information directly in the Tcl assignments in the **.qsf** file. If you do not specify this information for a pin, the Quartus II software breaks the signal out at the bottommost layer. While you can specify the number of layers using the GUI or Tcl assignments, the **PCB Layer** column in the All Pins list allows you to enter a layer that you may not yet have defined in the GUI. If there is a **.qsf** assignment that specifies a pin to break out at a layer that does not exist, the Quartus II software gives you a warning that the layer does not exist and it uses the bottommost layer:

```
Warning: Pin "e[2]" has an invalid assignment "10" made on the board stackup breakout
layer, using the bottommost layer for this pin.
```

I/O Assignments

The I/O assignments in the Quartus II software are also known as pin assignments. These assignments are required in FPGA design and are also used during SSN analysis in the Quartus II software. Each input, output, or bidirectional signal in your design is assigned a physical pin location on the device using pin location assignments. Each signal has a physical I/O buffer that has a specific I/O standard, pin location, current strength, and slew rate.

Altera device families support a wide variety of I/O standards. You can specify the I/O standard using the Pin Planner, Assignment Editor, or Tcl assignments.

To learn more about specifying the I/O standard, current strength, slew rate, and pin location, refer to the Pin Advisor in the Quartus II software, the Quartus II Help, or the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Automatic Aggressor Identification

The Quartus II software looks for the following situations to determine whether a pin can be an aggressor for a specific victim pin:

- If the aggressor pin is a complement of the victim pin, it is not considered an aggressor for the victim. This is the case for pins that have differential standards.
- If the aggressor pin is a programming or JTAG pin, it does not aggress on any victim pin because it is not active in user mode.
- If the victim pin is a bidirectional pin and analyzed for SSN as an input, the pins that have the same Output Enable (OE) cannot be aggressors, because those other pins also act as inputs and cannot aggress at the same time. Refer to "Group Assignments" for information about grouping bidirectional pins.
- If the victim pin is an output pin and belongs to a synchronous group, the pins that are specified in the same synchronous group cannot be aggressors for that victim pin. Refer to "Group Assignments" for information about grouping output pins.

Group Assignments

In the absence of any specific timing information, SSN analysis must assume worst-case conditions. Typically, this involves assuming that all pins act as aggressors on all possible victim pins. Similarly, all aggressor pins are assumed to be switching with the worst possible timing relationship. In reality, there are many relationships between I/O pins that make this assumption very pessimistic. The following relationships can help lessen that pessimism:

- **Common output enable signals**—If all the pins in a group are always either inputs or outputs, it is impossible for an output pin in the group to cause SSN noise on an input pin in the group. To do so violates the restriction that all pins are either inputs or outputs.
- **Synchronously related signals**—I/O pins that are part of a synchronous group (signals that switch at the same time) may cause SSN, but do not result in any failures because the noise glitch occurs during the switching period of the signal. The noise, therefore, does not occur in the sampling window of that signal.

 © March 2009 Altera Corporation

In some cases, the Quartus II software can detect the grouping for bidirectional pins by looking at the OE of the bidirectional pins. However, Altera recommends that you explicitly specify the bidirectional groups and output groups using Tcl assignments for your design.

You can specify a bidirectional group with the following Tcl assignment in the **.qsf** file:

```
set_instance_assignment -name OUTPUT_ENABLE_GROUP 1 -to DATAINOUT
```

where DATAINOUT is a bidirectional bus.

You can specify the an output group with the following SYNCHRONOUS_GROUP assignment:

```
set_instance_assignment -name SYNCHRONOUS_GROUP 1 -to PCI_AD_io
```

In this case, the PCI_AD_io bus may have 32 pins that all belong to the same group. In a real operation, the bus switches at the same time, so any voltage noise induced by a pin on its groupmate does not matter, because it does not fall in the sampling window. If this assignment is not used, the other 31 pins can act as aggressors for the first pin in that group, leading to higher QL and QH noise levels in SSN analysis. Specifying the correct grouping yields less pessimistic results in your SSN analysis.

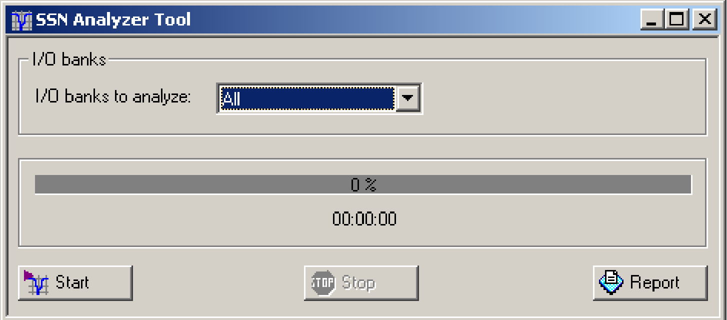
You can use the Assignment Editor to make these assignments. You can also use the Node Finder utility in the Quartus II software to find bidirectional or output buses.

Running the SSN Analyzer

You can start the SSN analysis in several ways:

- On the Processing menu in the Quartus II software, click **SSN Analyzer Tool**. The **SSN Analyzer Tool** dialog box appears (Figure 6–8).

Figure 6–8. SSN Analyzer Tool Dialog Box



- On the Processing menu, point to **Start** and click **Start SSN Analyzer**.
- On the command line, type the following commands:

```
quartus_si <project revision> ↵
quartus_si counter ↵
```

where counter is the project revision.

On the command line, you can run just one I/O bank, as follows:

```
quartus_si <project revision> <--bank = bank id> ↵
quartus_si counter --bank=2A ↵
```

To learn more about I/O bank numbering, refer to the package view for the device in the Pin Planner or refer to the device handbook.

For more information about the `quartus_si` package, type `-quartus_si -h` in the console window or command prompt, refer to the Quartus II Help, or refer to the *Quartus II Scripting Reference Manual*.

Understanding the SSN Reports

At the end of SSN analysis, various reports are printed in the Compilation Report section. You can view the reports by clicking the Compilation Report button on the Quartus II toolbar.

Settings Report

The Settings Report states whether or not smart compilation was used. To learn more about smart compilation, refer to the Quartus II Help.

Summary Report

The Summary Report summarizes the SSN analysis run and gives information such as whether or not the SSN run was successful, which Quartus II software version was used, the revision of the project used, and so forth. The report also rates the SSN Analyzer confidence level as low, medium, or high. The confidence level depends on how completely you have specified the user assignments described in “Tool Inputs” on page 6–7.

The more assignments you complete, the higher the confidence level. However, the confidence level does not always contribute to the accuracy of the QL and QH levels you get on a victim pin. The accuracy of QH and QL noise levels depends on how accurately you have defined your user assignments.

Input Pins Report

The Input Pins Report lists all of the input pins and bidirectional pins that are treated as inputs during SSN analysis, their location assignments on the FPGA device, the QL and QH noise in volts, and what percentage the QL and QH margins are for the I/O standard used for that signal. The QH and QL noise margins that fall in the critical range (> 90%) are shown in red. The QH and QL noise margins that fall in the range of 70% to 90% are shown in gray. You cannot change the color settings in this report.

Output Pins Report

The Output Pins Report lists all of the output pins and bidirectional pins that are treated as output pins during SSN analysis of your design, their location assignments on the FPGA device, the QL and QH noise in volts, and what percentage the QL and QH margins are for the I/O standard used for that signal. The QH and QL noise margins that fall in the critical range (> 90%) are shown in red. The QH and QL noise margins that fall in the range of 70% to 90% are shown in gray. You cannot change the color settings in this report.

Confidence Metric Details Report

The SSN Analyzer confidence level is reported in the Summary Report. The Confidence Metric Details Report lists the I/O, board, and PCB assignments that have not been specified by the user and the value that was used in its place by the SSN Analyzer.

Unanalyzed Pins Report

In the Quartus II software version 9.0, not all pins are analyzed for SSN analysis. The following pins are not analyzed and are reported in the Unanalyzed Pins Report:

- LVDS pins and any pins that have LVDS variations, such as mini-LVDS
- Pins created in the migration flow that cover power and supply pins in other packages
- The negative terminals of pseudo-differential standards; the noise on differential standards is reported as the differential noise and is reported on the positive terminal

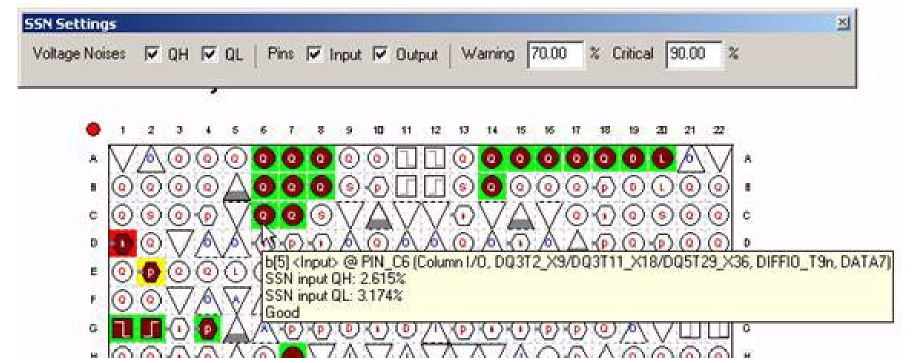
Visualizing SSN in the Pin Planner

After SSN analysis finishes, the results can be analyzed in the Quartus II Pin Planner. You can quickly identify the SSN hotspots in the package view of your device by using the Pin Planner's package view. In addition to viewing the QL and QH results in the reports, you can see the QL and QH noise levels in the Pin Planner. The QL and QH results for each pin are displayed with a color. This color representation is also referred to as the SSN map of your FPGA device. Besides the visualization, the integration of the SSN Analyzer with the Pin Planner provides you a what-if SSN analysis where you change I/O assignments and board trace information and rerun the SSN Analyzer.

Invoking the SSN Map

To view the SSN map of your device pins, right-click in the package window in the Pin Planner and click **Show SSN Analyzer Results**, as shown in Figure 6–9.

Figure 6–9. SSN Map



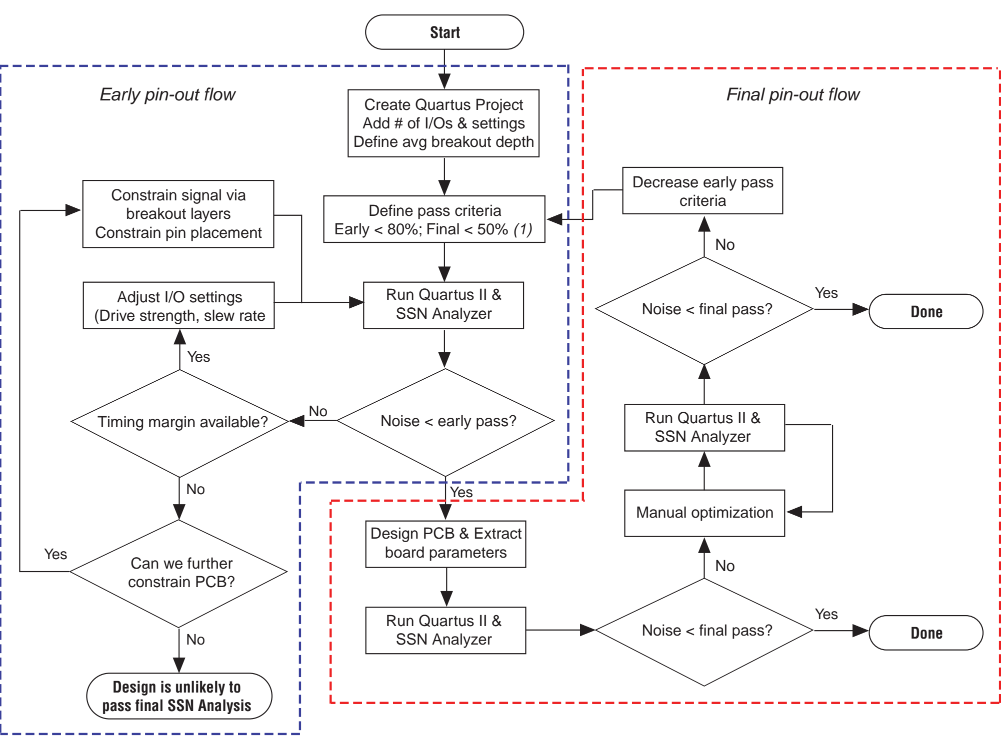
Along with the SSN map, the SSN toolbar is displayed. The toolbar lets you select only input pins, only output pins, or both, and to view QH, QL, or both noise levels. It also lets you change the threshold levels for QH and QL noise voltages. For example, by default noise levels that consume more than 90% of the signal margin of a pin are shown in red. Similarly, noise levels that consume 70% to 90% of the signal margin are shown in yellow and levels below 70% are shown in green. If you change this threshold, the results are updated in the SSN map. Changing the threshold levels in the SSN toolbar does not change the threshold levels in the SSN reports. The threshold levels in the SSN reports are fixed. For example, if you change the critical threshold level in the Pin Planner to be 80% instead of 90% (which is the default), some pins might appear as red instead of yellow, but the SSN reports will still show their noise levels in gray. When you hover your mouse over a pin, the QH and QL noise levels are displayed, as shown in Figure 6–9.

SSN Analyzer Usage Models

Based on which stage your design cycle is in, you can run the SSN Analyzer at a very early stage of your design cycle or at a stage where your PCB design is almost complete. Altera recommends that you start your SSN analysis early in the design (an early pin-out analysis) and later do a fully constrained SSN analysis with complete information about your board (a final pin-out analysis).

The basic methodology of early pin-out and final pin-out analysis, shown in Figure 6–10, assumes conservative design rules initially, then lets you analyze the design and iteratively apply tighter design rules until SSN analysis indicates a passable design. You must define a pass criterion for the SSN analysis as a percent of signal margin for both the early and final analysis. The early pass criterion may be higher than the final pass criterion, so that you do not spend too much time optimizing the on-FPGA portions of your design when the SSN metrics for the design may improve after the design is fully specified.

Figure 6–10. Pin-Out Analysis



Note to Figure 6–10:

(1) Pass criteria to be determined by customer requirements.

Early Pin-Out SSN Analysis

Early pin-out SSN analysis occurs before you have placed I/Os in your Quartus II project. In this SSN analysis, you might not have all the design files ready but you have the interface information ready. If you know what I/O standards and signaling standards are to be applied to those interfaces, you can use either the early SSN spreadsheet tool or the SSN Analyzer in the Quartus II software to perform an initial SSN evaluation of your design.

Early Pin-Out SSN Analysis Using the Early SSN Estimator Spreadsheet

The Altera Early SSN Estimator (ESE) spreadsheet provides basic early pin-out SSN analysis using simple equation-based SSN models. To learn more about the SSN spreadsheet tool, refer to the Signal Integrity Center on the Altera website.

Early Pin-Out SSN Analysis Using the Quartus II SSN Analyzer

The integration of the SSN Analyzer in the Quartus II software enables you to do an early SSN analysis without having any design files ready. If you have complete information for your top-level interface, you can enter that information in the Quartus II software and run the SSN Analyzer to view the early results.

If you plan to use the SSN Analyzer, you must create the Quartus II software project and have at least the information for all the top-level ports or interfaces of your FPGA design. Your top-level port information can be entered in the Quartus II software project in a number of ways. You can use the schematic entry method, or if you have the top-level design file in HDL, you can use that in the Quartus II software directly. If you have a top-level file, you can generate a top-level wrapper file in the Quartus II software in HDL.

To learn more about how to generate a top-level file in the Quartus II software, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

To learn more about how to create projects, refer to the Quartus II Help or the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

When you have the top-level ports ready in any of these forms, you can analyze and synthesize your Quartus II project. In an early pin-out analysis, pin location assignments are assumed not to exist. In the Pin Planner, you can make other I/O assignments such as I/O standard assignments for the top-level ports.

The SSN Analyzer requires you to run the Fitter. In an early pin-out analysis, you may not have all the design files and timing constraints complete. You can run I/O Assignment analysis to place all the I/Os in your FPGA device. During I/O Assignment analysis, the Fitter places all the unplaced pins on the device, and all the I/O placement rules are checked. After the I/O Assignment completes successfully, you are ready to start the SSN analysis.

The I/O rules and their validation process are discussed in the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

In an early pin-out analysis, you might not have all the board information, such as board trace parameters, layers information, and on which layer the pins break out. You can run the SSN Analyzer without this information, but the SSN Analyzer confidence level will be low. At this point the SSN Analyzer can be run and the results analyzed. If the noise amounts are larger than the early pass criteria, you can check whether the SSN noise violations are true failures or false failures. Although the Quartus II SSN Analyzer can sometimes determine whether pins are switching synchronously and use that information to filter false positives, it may not be able to determine all the synchronous groups. You can help the SSN Analyzer by entering assignments that indicate which pins are switching synchronously, such as large buses, and rerun the SSN Analyzer.

After filtering out false positives, if the SSN results are still larger than the pass criterion, you can change design settings to improve the design.

If timing margin is available on various signals, those signal edge rates can be slowed down by changing their drive strength or slew rate setting and rerunning the SSN Analyzer. If timing margin is not available, you can further constrain the PCB breakout depth or adjust the pin placements. With an early SSN analysis, you can see what QL and QH noise levels exist in your FPGA device based on your existing I/O

placements and assignments. The integration of the SSN Analyzer with the Quartus II software allows you to perform a what-if analysis to see how I/O placement changes and I/O assignment changes vary the QL and QH levels in your FPGA device. This capability allows you to manage SSN in your device and system early in the PCB development because you know the noise margin on your FPGA pins.

When you change any I/O assignments in the Pin Planner, you must run I/O Assignment analysis or full compilation before you perform SSN analysis.

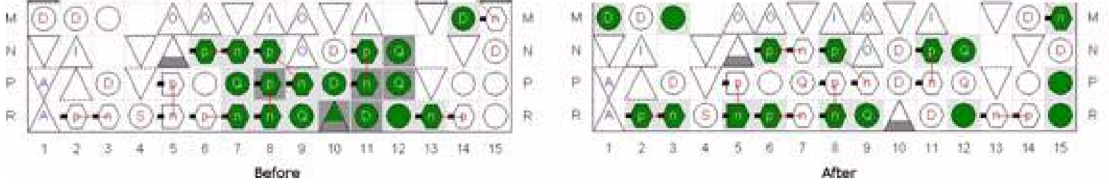
SSN Aware Fitter

In the early pin-out SSN analysis, when you perform I/O assignment analysis or full compilation, if you have not assigned physical locations for your I/Os, the Fitter places the I/Os in various pin locations such that I/O rules are not violated. In the Quartus II software version 9.0 and later, you can run the I/O assignments analysis or full compilation in SSN aware mode to provide an I/O placement that reduces SSN. Refer to “SSN Optimization” on page 6–20 for information about using the SSN aware Fitter.

You can choose an effort level for SSN aware fitting. The Fitter tries to spread out the pins to minimize the worst-case SSN-induced noise. If your I/O pin locations are assigned, the Fitter does not change the pin locations. For the Fitter to spread out the pins for SSN, you must either not have assignments made or if the assignments exist, you must delete the pin location assignments from the **.qsf** file and rerun I/O assignment analysis or full compilation.

With regard to pin placements, the Quartus II software has an SSN aware Fitter that automatically adjusts the pin placements to reduce the amount of SSN in the design. Figure 6–11 shows an example design run through the SSN Analyzer before and after the SSN aware Fitter is used.

Figure 6–11. SSN Results Before and After Using the SSN aware Fitter



Default Assignments Used in Early SSN Analysis

Early in your design cycle, you may not have any board trace model or board layer information. When you run SSN analysis without this information, the Quartus II software uses default board trace models and board layer information. This information is reported in the Confidence Metric report. After the design has been optimized such that it passes the early criteria, the PCB can be designed.

Final Pin-Out Analysis: Fully Constrained Design SSN Analysis

A final pin-out analysis is performed when you or the Fitter have placed the I/O in your design and you want to perform an SSN analysis with complete information of board traces and layers. As a pre-tapeout check for your PCB, the board parameters can be extracted, including the PCB layer thicknesses, the layers on which different signals break out, and the board trace topologies and parameters, and reanalyzed by the SSN Analyzer. If sufficient margin is available, given the early criteria, the extraction of the PCB parameters is optional and can be skipped.

These parameters should be entered into the Quartus II software using both the **Settings** dialog box and the **Board Trace Model Settings** dialog box, as described in "Tool Inputs" on page 6–7. The Quartus II software allows for the specification of near- and far-end loads, near- and far-end pull-up and pull-down resistors, and near- and far-end series resistors, as well as specifying the parameters of near- and far-end transmission lines that can model striplines and micro-striplines.

After entering the parameters, you can run the SSN Analyzer again. If the results pass the final criteria, the design is complete. If the design does not pass the criteria, the design must be micro-optimized by changing the board and design parameters and rerunning the SSN Analyzer. After all of these optimizations, if the design still does not pass the criteria, the early pass criteria should be reduced, and the process restarted. By reducing the early pass criteria, there is a larger emphasis placed on reducing the SSN through I/O settings and I/O placement that will then allow the design to pass the final SSN criteria after the actual PCB board parameters have been specified.

Scripting Support

To run SSN analysis using the command line, use the `quartus_si` package that is provided with the Quartus II software. You can use the Tcl console in the Quartus II software or type the following command to start the SSN Analyzer:

```
-quartus_si <project revision> ↵
```

The Quartus II software provides several packages to compile your design and run I/O assignments for analysis and fitting. You can create a custom Tcl script that maps the design and runs SSN analysis on your design.

For more information about Tcl scripting, refer to the Quartus II Help or the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Run Time Considerations in SSN Analysis

FPGA designs are getting larger in density, logic, and I/O count. The time it takes to complete a process of either synthesis or SSN analysis affects your development time. Faster run times can reduce your design cycle time. Following are some guidelines to consider when performing SSN analysis to reduce the run time.

Running SSN Analyzer with Multi-CPU machines

The Quartus II software has many algorithms that are multi-threaded. The SSN Analyzer in the Quartus II software is also multi-threaded, and can use two or more CPUs in a machine. The SSN run time analysis scales down directly with the number of CPUs used during the analysis.

Running the Complete Design for SSN Analysis after I/O Assignment Analysis

If your design files and constraints are ready but you do not want to run a full compilation, you can run SSN analysis after performing I/O assignment analysis in the Quartus II software. This can save you time if you are interested in looking at the SSN results early in the design and want to perform what-if analysis for your I/O placements.

Running the Complete Design for SSN Analysis after a Full Fit

When you run full compilation, the Quartus II Fitter runs all the tasks for fitting your design. You should always perform a full compilation to get complete and accurate I/O assignments validation and SSN results. However, full compilation of your design can take more time, depending on the logic density and timing requirements of your design. You can also choose to run just the Fitter but not the assembler and timing analysis if you want to run SSN analysis after just the fitting process.

Making ECO Changes and Rerunning SSN Analysis

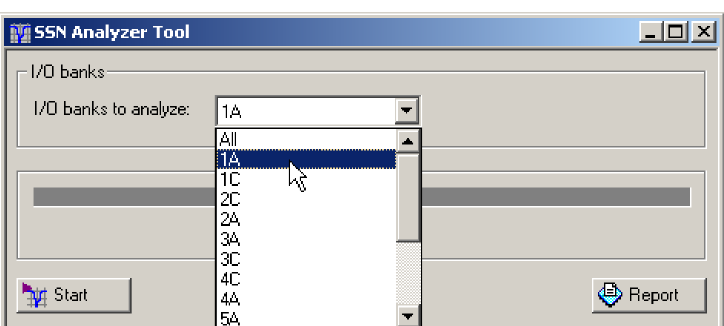
If you have performed SSN analysis after a full compilation and you want to rerun SSN analysis to see what happens to QL and QH noise levels after some changes to the I/O assignments, the best way is to perform ECOs on your design. ECOs do not compile the whole design. Instead, they compile the design for the new changes only. This can save a lot of compilation time.

For more information about performing ECOs on your design, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Running SSN Analysis for One I/O Bank

You can run SSN analysis on the full design or on one I/O bank only. Figure 6–12 shows how to select a particular I/O bank before running SSN analysis with the SSN Analyzer tool.

Figure 6–12. Selecting an I/O Bank before Running SSN Analysis



The following Tcl command line runs SSN analysis for one I/O bank:

```
quartus_si <project revision> <--bank=bank id>
```

If you know the problem area for SSN is within one bank and you are performing I/O placement or assignment changes to only that bank, running SSN analysis for just that one bank can save you run time.

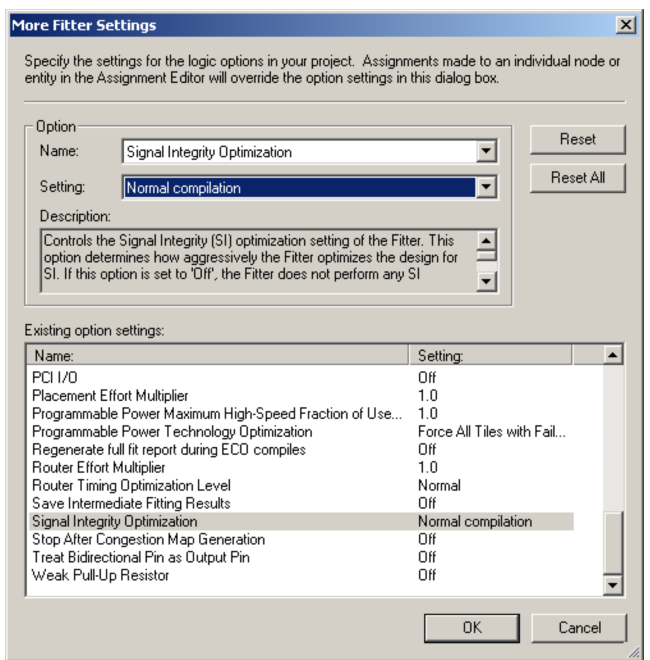
SSN Optimization

The Quartus II software has a built-in feature to optimize your design for SSN. To select an effort level for SSN optimization, in the **Settings** dialog box, select **Fitter Settings** from the **Category** list (Figure 6–13). Click **More Settings** to bring up the **More Fitter Settings** dialog box shown as in Figure 6–14.

Figure 6–13. Fitter Settings Dialog Box

Settings - counter
Category:
General
Files
Libraries
Device
Operating Settings and Conditions
Compilation Process Settings
EDA Tool Settings
Analysis & Synthesis Settings
Fitter Settings
Timing Analysis Settings
Assembler
Design Assistant
SignalTap II Logic Analyzer
Logic Analyzer Interface
Simulator Settings
PowerPlay Power Analyzer Settings
SSN Analyzer
Fitter Settings
Specify options for fitting.
Timing-driven compilation
Optimize hold timing: All Paths
Optimize multi-corner timing
PowerPlay power optimization: Normal compilation
Fitter effort
Standard Fit (highest effort)
Fast Fit (up to 50% faster compilation / may reduce fmax)
Auto Fit (reduce Fitter effort after meeting timing requirements)
Desired worst case slack (margin): 0 ns
Limit to one fitting attempt
Seed: 1
More Settings...
Description:
OK
Cancel

Figure 6–14. SSN Optimization Settings



In the **More Fitter Settings** dialog box, set the **Signal Integrity Optimization** option. The I/O placements in your design may be affected when you use this feature. Choosing the **Normal compilation** level does not affect the f_{MAX} of your design during compilation. Choosing the **Extra effort** level might impact the f_{MAX} of your design during compilation.

In the Quartus II software, you can assign your signal to the pin locations on the device using the Pin Planner or Assignment Editor. When you assign the pin locations, the information is saved as pin location assignments in the **.qsf** file.

You must not set any user location assignments for your pins; instead, let the Fitter place the pins while compiling your design. During compilation, if you have not made any pin location assignments in your project, the Fitter places the pins to meet the timing performance of your design. The pins that are placed by the Fitter can be viewed in the Pin Planner's package view by selecting **Show Fitter Placements**. When the Fitter places the pins automatically, no pin location assignments are created in the **.qsf** file unless you back-annotate them.

There are various optimization options available in the Quartus II software. For more information about these optimization features, refer to the Quartus II Help or the *Area, Timing and Power Optimization* section in volume 2 of the *Quartus II Handbook*.

You can also specify the effort level by using the following Tcl command:

```
set_global_assignment -name OPTIMIZE_SIGNAL_INTEGRITY "Normal Compilation"
```

Back-Annotating the Fitter Results

Back-annotating after the Fitter finishes its task saves the Fitter-placed results in the **.qsf** file. There are various options available when back-annotating. When you back-annotate the Fitter placements, the pin location assignments are saved in the **.qsf** file. To learn more about back-annotation, refer to the Quartus II Help.

SSN Optimization in Your System

This chapter discussed various tools available in the Quartus II software to analyze SSN and optimize SSN with SSN aware Fitter and I/O assignments settings. There are other optimization techniques to manage SSN in your PCB.

To learn more about managing SSN in your system, refer to *AN 472: Stratix II GX SSN Design Guidelines*, *AN 508: Cyclone III Simultaneous Switching Noise (SSN) Design Guidelines*, and high-speed board design guidelines available at www.altera.com.

Conclusion

In the Quartus II software version 9.0 and later, you can estimate SSN in your design using the fast and accurate SSN Analyzer. There are tools in the Quartus II software that allow you to estimate the SSN performance of your FPGA both early in the design cycle and when your PCB is complete. The SSN methodology discussed in this chapter gives you confidence that your FPGA design meets your SSN requirements.

Referenced Documents

This chapter references the following documents:

- *AN 472: Stratix II GX SSN Design Guidelines*
- *AN 508: Cyclone III Simultaneous Switching Noise (SSN) Design Guidelines*
- *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 6–2 shows the revision history for this chapter.

Table 6–2. Document Revision History

Date / Revision	Changes Made	Summary of Changes
March 2009 v9.0.0	Initial release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

7. Signal Integrity Analysis with Third-Party Tools

QII53020-9.0.0

Introduction

With the ever-increasing operating speed of interfaces in traditional FPGA design, the timing and signal integrity margins between the FPGA and other devices on the board must be within specification and tolerance before a single PCB is built. If the board trace is designed poorly or the route is too heavily loaded, noise in the signal can cause data corruption, while overshoot and undershoot can potentially damage input buffers over time.

As FPGA devices are used in high-speed applications, signal integrity and timing margin between the FPGA and other devices on the printed circuit board (PCB) are important aspects to consider to ensure proper system operation. To avoid time-consuming redesigns and expensive board respins, the topology and routing of critical signals must be simulated. The high-speed interfaces available on current FPGA devices must be modeled accurately and integrated into timing models and board-level signal integrity simulations. The tools used in the design of an FPGA and its integration into a PCB must be "board-aware"—able to take into account properties of the board routing and the connected devices on the board.

This chapter contains the following topics:

- "I/O Model Selection: IBIS or HSPICE" on page 7–3
- "FPGA to Board Signal Integrity Analysis Flow" on page 7–3
- "Simulation with IBIS Models" on page 7–7
- "Simulation with HSPICE Models" on page 7–17

The Quartus® II software provides methodologies, resources, and tools to ensure good signal integrity and timing margin between Altera® FPGA devices and other components on the board. Three types of analysis are possible with the Quartus II software:

- I/O timing with a default or user-specified capacitive load and no signal integrity analysis (default)
- The Quartus II **Enable Advanced I/O Timing** option utilizing a user-defined board trace model to produce enhanced timing reports from accurate "board-aware" simulation models
- Full board routing simulation in third-party tools using Altera-provided or generated Input/Output Buffer Information Specification (IBIS) or HSPICE I/O models

I/O timing using a specified capacitive test load requires no special configuration other than setting the size of the load. I/O timing reports from the Quartus II TimeQuest or the Quartus II Classic Timing Analyzer are generated based only on point-to-point delays within the I/O buffer and assume the presence of the capacitive test load with no other details about the board specified. The default size of the load is based on the I/O standard selected for the pin. Timing is measured to the FPGA pin with no signal integrity analysis details.

The **Enable Advanced I/O Timing** option expands the details in I/O timing reports by taking board topology and termination components into account. A complete point-to-point board trace model is defined and accounted for in the timing analysis. This ability to define a board trace model is an example of how the Quartus II software is "board-aware."

In this case, timing and signal integrity metrics between the I/O buffer and the defined far end load are analyzed and reported in enhanced reports generated by the Quartus II TimeQuest Timing Analyzer.

For more information about defining capacitive test loads or how to use the **Enable Advanced I/O Timing** option to configure a board trace model, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

This chapter focuses on the third type of analysis. The Quartus II software can export accurate HSPICE models with the built-in HSPICE Writer. You can run signal integrity simulations with these complete HSPICE models in Synopsys HSPICE. IBIS models of the FPGA I/O buffers are also created easily with the Quartus II IBIS Writer. You can integrate IBIS models into any third-party simulation tool that supports them, such as the Mentor Graphics® Hyperlynx software. With the ability to create industry-standard model definition files quickly, you can build accurate simulations that can provide data to help improve board-level signal integrity.

The I/O's IBIS and HSPICE model creation available in the Quartus II software can help prevent problems before a costly board respin is required. In general, creating and running accurate simulations is difficult and time consuming. The tools in the Quartus II software automate the I/O model setup and creation process by configuring the models specifically for your design. With these tools, you can set up and run accurate simulations quickly and acquire data that helps guide your FPGA and board design.

The information about signal integrity in this chapter refers to board-level signal integrity based on I/O buffer configuration and board parameters, not simultaneous switching noise (SSN), also known as ground bounce or V_{CC} sag. SSN is a product of multiple output drivers switching at the same time, causing an overall drop in the voltage of the chip's power supply. This can cause temporary glitches in the specified level of ground or V_{CC} for the device.

For a more information about SSN and ways to prevent it, refer to *AN 315: Guidelines for Designing High-Speed FPGA PCBs*.

This chapter is intended for FPGA and board designers and includes details about the concepts and steps involved in getting designs simulated and how to adjust designs to improve board-level timing and signal integrity. Also included is information about how to create accurate models from the Quartus II software and how to use those models in simulation software.

The information in this chapter is meant for those who are familiar with the Quartus II software and basic concepts of signal integrity and the design techniques and components in good PCB design. Finally, you should know how to set up simulations and use your selected third-party simulation tool.

For information about basic signal integrity concepts and signal integrity details pertaining to Altera FPGA devices, refer to the Altera Signal Integrity Center.

I/O Model Selection: IBIS or HSPICE

The Quartus II software can export two different types of I/O models that are useful for different simulation situations. IBIS models define the behavior of input or output buffers through the use of voltage-current (V-I) and voltage-time (V-t) data tables. HSPICE models, often referred to as HSPICE decks, include complete physical descriptions of the transistors and parasitic capacitances that make up an I/O buffer along with all the parameter settings required to run a simulation. The HSPICE decks generated by the Quartus II software are preconfigured with the I/O standard, voltage, and pin loading settings for each pin in your design.

The choice of I/O model type is based on many factors. Table 7–1 shows a detailed comparison of the two I/O model types and information and examples of situations in which they might be used.

Table 7–1. IBIS and HSPICE Model Comparison

Feature	IBIS Model	HSPICE Model
I/O Buffer Description	**Behavioral**—I/O buffers are described by voltage-current and voltage-time tables in typical, minimum, and maximum supply voltage cases.	**Physical**—I/O buffers and all components in a circuit are described by their physical properties, such as transistor characteristics and parasitic capacitances, as well as their connections to one another.
Model Customization	**Simple and limited**—The model completely describes the I/O buffer and does not usually have to be customized.	**Fully customizable**—Unless connected to an arbitrary board description, the description of the board trace model must be customized in the model file. All parameters of the simulation are also adjustable.
Simulation Set Up and Run Time	**Fast**—Simulations run quickly after set up correctly.	**Slow**—Simulations take time to set up and take longer to run and complete.
Simulation Accuracy	**Good**—For most simulations, accuracy is sufficient to make useful adjustments to the FPGA and/or board design to improve signal integrity.	**Excellent**—Simulations are highly accurate, making HSPICE simulation almost a requirement for any high-speed design where signal integrity and timing margins are tight.
Third-Party Tool Support	**Excellent**—Almost all third-party board simulation tools support IBIS.	**Good**—Most third-party tools that support SPICE support HSPICE. However, Synopsys HSPICE is required for simulations of Altera's encrypted HSPICE models.

For more information about IBIS files created by the Quartus II IBIS Writer and IBIS files in general, as well as links to websites with detailed information, refer to *AN 283: Simulating Altera Devices with IBIS Models*.

FPGA to Board Signal Integrity Analysis Flow

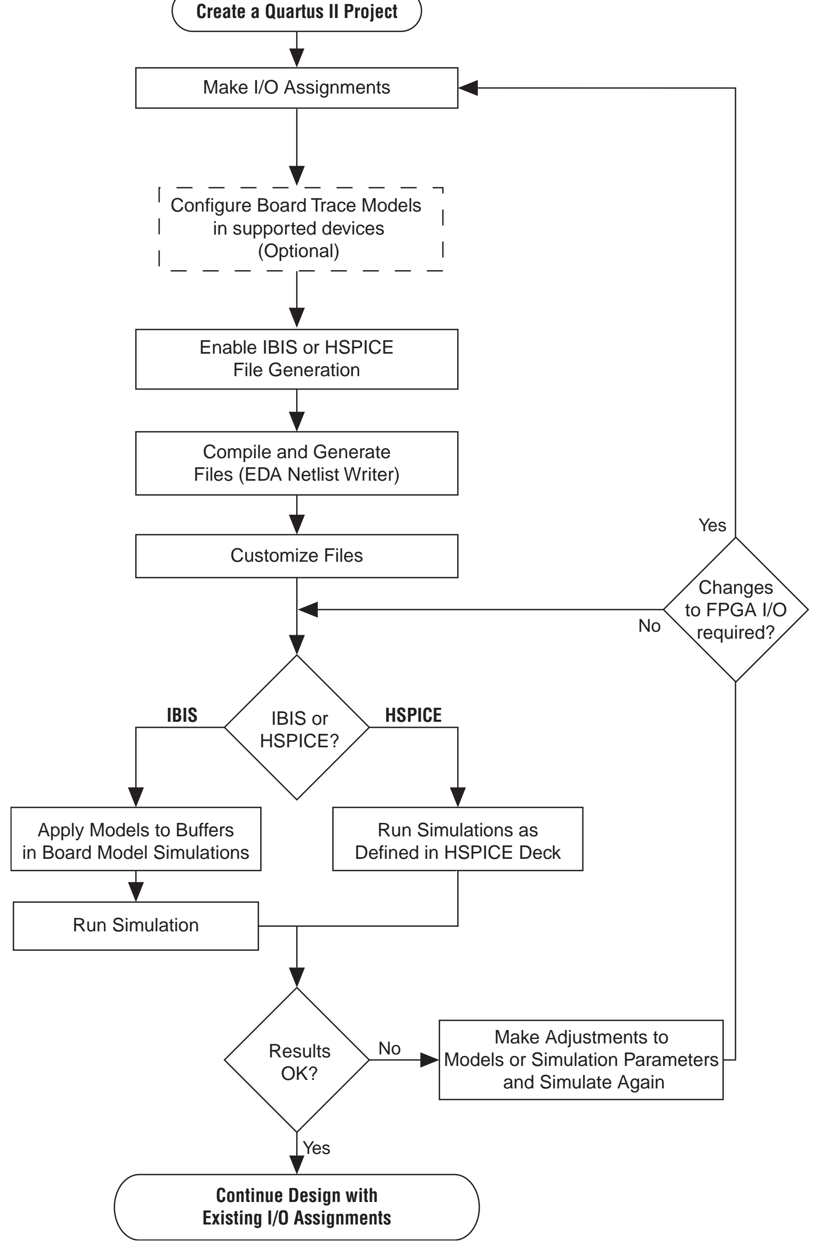
Board signal integrity analysis can take place at any point in the FPGA design process and is often performed before and after board layout. If it is performed early in the process as part of a pre-PCB layout analysis, the models used for simulations can be more generic and can be changed as much as required to see how adjustments improve timing or signal integrity and help with the design and routing of the PCB.

Simulations and the resulting changes made at this stage allow you to analyze "what if" scenarios to plan and implement your design better. To assist with early board signal integrity analysis, you can download generic IBIS model files for each device family and obtain HSPICE buffer simulation kits from the "Board Level Tools" section of the Download center on the Altera website at www.altera.com.

Typically, if board signal integrity analysis is performed late in the design, it is used for a post-layout verification. The inputs and outputs of the FPGA are defined, and required board routing topologies and constraints are known. Simulations can help you find problems that might still exist in the FPGA or board design before fabrication and assembly. In either case, a simple process flow illustrates how to create accurate IBIS and HSPICE models from a design in the Quartus II software and transfer them to third-party simulation tools. Figure 7–1 shows this flow.

This chapter is organized around the type of model, IBIS or HSPICE, that you use for your simulations. When you understand the steps in the analysis flow, refer to the section of this chapter that corresponds to the model type you are using.

Figure 7–1. Third-Party Board Signal Integrity Analysis Flow



Create I/O and Board Trace Model Assignments

If your design uses a Stratix® III, Stratix II, or Cyclone® III device, you can configure a board trace model for output signals or for bidirectional signals in output mode and automatically transfer its description to HSPICE decks generated by the HSPICE Writer. This helps improve simulation accuracy.

To configure a board trace model, in the **Settings** dialog box, in the **TimeQuest Timing Analyzer** page, turn on the **Enable Advanced I/O Timing** option and configure the board trace model assignment settings for each I/O standard used in your design. You can add series or parallel termination, specify the transmission line length, and set the value of the far-end capacitive load. You can configure these parameters either in the Board Trace Model view of the Pin Planner, or click **Device and Pin Options** in the **Device** page of the **Settings** dialog box.

For information about how to use the **Enable Advanced I/O Timing** option and configure board trace models for the I/O standards used in your design, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

The Quartus II software can generate IBIS models and HSPICE decks without having to configure a board trace model with the **Enable Advanced I/O Timing** option. In fact, IBIS models ignore any board trace model settings other than the far-end capacitive load. If any load value is set other than the default, the delay given by IBIS models generated by the IBIS Writer cannot be used to account correctly for the double counting problem. The load value mismatch between the IBIS delay and the t_{CO} measurement of the Quartus II software prevents the delays from being safely added together. Warning messages displayed when the EDA Netlist Writer runs indicate when this mismatch occurs.

Output File Generation

IBIS and HSPICE model files are not generated by the Quartus II software by default. To generate or update the files automatically during each project compilation, select the type of file to generate and a location where to save the file in the project settings. These settings can also be specified with commands in a Tcl script.

The IBIS and HSPICE Writers in the Quartus II software are run as part of the EDA Netlist Writer during normal project compilation. If either writer is turned on in the project settings, IBIS or HSPICE files are created and stored in the specified location. For IBIS, a single file is generated containing information about all assigned pins. HSPICE file generation creates separate files for each assigned pin. You can run the EDA Netlist Writer separately from a full compilation in the Quartus II software or at the command line. However, you must fully compile the project or perform I/O Assignment Analysis at least once for the IBIS and HSPICE Writers to have information about the I/O assignments and settings in the design.

Customize the Output Files

The files generated by either the IBIS or HSPICE Writer are text files that you can edit and customize easily for design or experimentation purposes. IBIS files downloaded from the Altera website must be customized with the correct RLC values for the specific device package you have selected for your design. IBIS files generated by the IBIS Writer do not require this customization because they are configured automatically with the RLC values for your selected device. HSPICE decks require modification to include a detailed description of your board. With **Enable Advanced I/O Timing** turned on and a board trace model defined in the Quartus II software, generated HSPICE decks automatically include that model’s parameters. However, Altera recommends that you replace that model with a more detailed model that

describes your board design more accurately. A default simulation included in the generated HSPICE decks measures delay between the FPGA and the far-end device. You can make additions or adjustments to the default simulation in the generated files to change the parameters of the default simulation or to perform additional measurements.

Set Up and Run Simulations in Third-Party Tools

When you have generated the files, you can use them to perform simulations in your selected simulation tool. With IBIS models, you can apply them to input, output, or bidirectional buffer entities and quickly set up and run simulations. For HSPICE decks, the simulation parameters are included in the files. Open the files in Synopsys HSPICE and run simulations for each pin as required.

With HSPICE decks generated from the HSPICE Writer, the double counting problem is accounted for, which ensures that your simulations are accurate. Simulations that involve IBIS models created with anything other than the default loading settings in the Quartus II software must take the change in the size of the load between the IBIS delay and the Quartus II t_{CO} measurement into account. Warning messages during compilation alert you to this change.

Interpret Simulation Results

If you encounter timing or signal integrity issues with your high-speed signals after running simulations, you can make adjustments to I/O assignment settings in the Quartus II software. These could include such things as drive strength or I/O standard, or making changes to your board routing or topology. After regenerating models in the Quartus II software based on the changes you have made, rerun the simulations to check whether your changes corrected the problem.

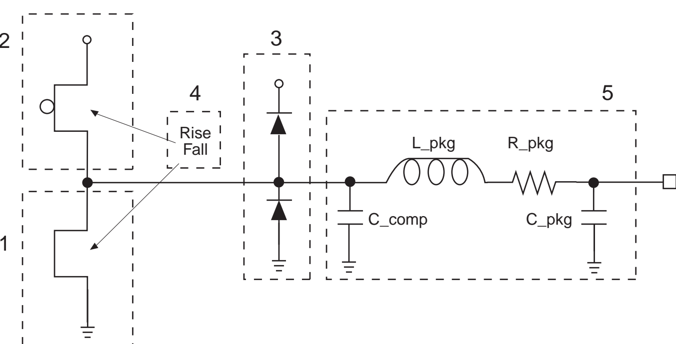
Simulation with IBIS Models

IBIS models provide a way to run accurate signal integrity simulations quickly. IBIS models describe the behavior of I/O buffers with voltage-current and voltage-time data curves. Because of their behavioral nature, IBIS models do not have to include any information about the internal circuit design of the I/O buffer. Most component manufacturers, including Altera, provide IBIS models for free download and use in signal integrity analysis simulation tools. You can download generic device family IBIS models from the Altera website for early design simulation or use the IBIS Writer to create custom IBIS models for your existing design.

Elements of an IBIS Model

An IBIS model file (**.ibs**) is a text file that describes the behavior of an I/O buffer across minimum, typical, and maximum temperature and voltage ranges with a specified test load. The tables and values specified in the IBIS file describe five basic elements of the I/O buffer. Figure 7–2 highlights each of these elements in the I/O buffer model.

Figure 7–2. Five Basic Elements in IBIS Models



The following elements correspond to each numbered block in Figure 7–2.

1. **Pulldown**—A voltage-current table describes the current when the buffer is driven low based on a pull-down voltage range of $-V_{CC}$ to 2 V_{CC}.
2. **Pullup**—A voltage-current table describes the current when the buffer is driven high based on a pull-up voltage range of $-V_{CC}$ to V_{CC}.
3. **Ground and Power Clamps**—Voltage-current tables describe the current when clamping diodes for electrostatic discharge (ESD) are present. The ground clamp voltage range is $-V_{CC}$ to V_{CC}, and the power clamp voltage range is $-V_{CC}$ to ground.
4. **Ramp and Rising/Falling Waveform**—A voltage-time (dv/dt) ratio describes the rise and fall time of the buffer during a logic transition. Optional rising and falling waveform tables can be added to more accurately describe the characteristics of the rising and falling transitions.
5. **Total Output Capacitance and Package RLC**—The total output capacitance includes the parasitic capacitances of the output pad, clamp diodes (if present), and input transistors. The package RLC is device package-specific and defines the resistance, inductance, and capacitance of the bond wire and pin of the I/O.

For more information about IBIS models and Altera-specific features, including links to the official IBIS specification, refer to *AN 283: Simulating Altera Devices with IBIS Models*.

Creating Accurate IBIS Models

There are two methods to obtain Altera device IBIS files for your board-level signal integrity simulations. You can download generic IBIS models from the Altera website or you can use the IBIS writer in the Quartus II software to create design-specific models.

Download IBIS Models

Altera provides IBIS models for almost all FPGA and FPGA configuration devices. Check the Download Center at www.altera.com for information about whether models for your selected device are available. You can use the IBIS models from the website to perform early simulations of the I/O buffers you expect to use in your design as part of a pre-layout analysis.

Downloaded IBIS models have the RLC package values set to one particular device in each device family. To simulate your design with the model accurately, you must adjust the RLC values in the IBIS model file to match the values for your particular device package by performing the following steps:

1. Download and expand the ZIP file (**.zip**) of the IBIS model for the device family you are using for your design. The **.zip** file contains the **.ibs** file along with an IBIS model user guide and a model data correlation report.
2. Download the Package RLC Values spreadsheet for the same device family.
3. Open the spreadsheet and locate the row that describes the device package used in your design.
4. From the package's **I/O** row, copy the minimum, maximum, and typical values of resistance, inductance, and capacitance for your device package.
5. Open the **.ibs** file in a text editor and locate the [Package] section of the file.
6. Overwrite the listed values copied with the values from the spreadsheet and save the file.

The **.ibs** file is now customized for your device package and can be used for any simulation. IBIS models downloaded and used for simulations in this manner are generic. They describe only a certain set of models listed for each device on the IBIS model Download Center page on the Altera website. To create customized models for your design, use the IBIS Writer as described in the next section.

Generate Custom IBIS Models with the IBIS Writer

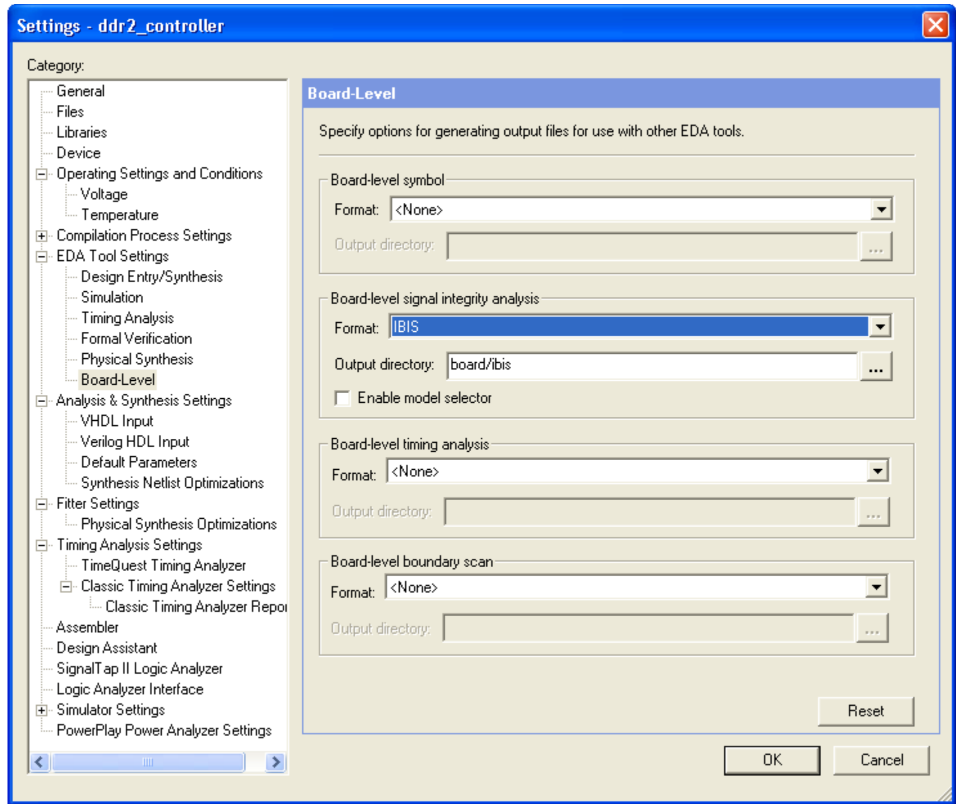
If you have started your FPGA design and have created custom I/O assignments, such as drive strength settings or the enabling of clamping diodes for ESD protection, you can use the Quartus II IBIS Writer to create custom IBIS models to accurately reflect your assignments. IBIS models created with the IBIS Writer take I/O assignment settings into account.

If the **Enable Advanced I/O Timing** option is turned off, the generated **.ibs** files are based on the load value setting for each I/O standard on the **Capacitive Loading** tab of the **Device and Pin Options** dialog box in the **Device** page of the **Settings** dialog box. With the **Enable Advanced I/O Timing** option turned on, IBIS models use an effective capacitive load based on settings found in the board trace model on the **Board Trace Model** tab in the **Device and Pin Options** dialog box or the **Board Trace Model** view in the Pin Planner. The effective capacitive load is based on the sum of the **Near capacitance**, **Transmission line distributed capacitance**, and the **Far capacitance** settings in the board trace model. Resistances and transmission line inductance values are ignored.

If you made any changes from the default load settings, the delay in the generated IBIS model cannot safely be added to the Quartus II t_{CO} measurement to account for the double counting problem. This is because the load values between the two delay measurements do not match. When this happens, the Quartus II software displays warning messages when the EDA Netlist Writer runs to remind you about the load value mismatch.

When the IBIS Writer is enabled in the **Settings** dialog box (Figure 7–3), it generates a custom **.ibs** file whenever the EDA Netlist Writer is run in the Quartus II software.

Figure 7–3. Enabling IBIS Model Generation in the Settings Dialog Box



IBIS models are stored in the *<project directory>***/board/ibis** directory by default. To change the directory, click the browse button next to the **Output directory** box, and browse to the desired location.

If the project has not been compiled, run a full compilation to create a netlist and establish I/O assignments. On the Processing menu, click **Start Compilation**. The **.ibs** file, named *<project name>***.ibs**, is saved in the specified location.

If the project has been compiled before, you only have to run the EDA Netlist Writer to create or update the **.ibs** file. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer**. The **.ibs** file is created or updated in the specified location.

For more information about IBIS model generation, refer to the *AN 283: Simulating Altera Devices with IBIS Models* or to the Quartus II Help.

Design Simulation Using the Mentor Graphics HyperLynx Software

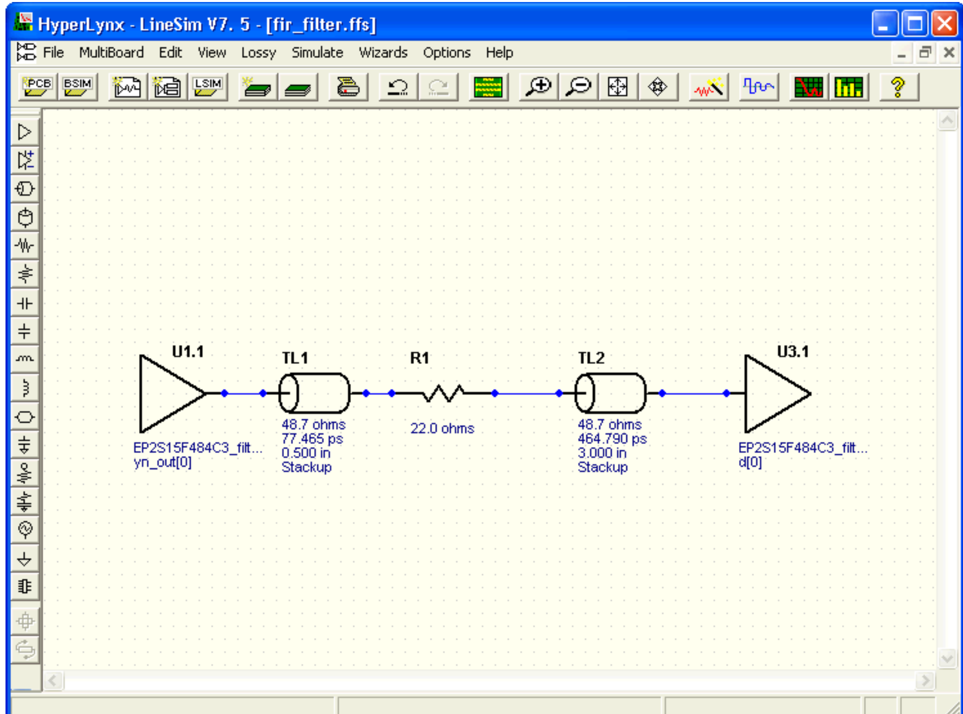
You must integrate IBIS models downloaded from the Altera website (www.altera.com) or created with the Quartus II IBIS Writer into board design simulations to accurately model timing and signal integrity. The HyperLynx software from Mentor Graphics is one of the most popular tools for design simulation. The HyperLynx software makes it easy to integrate IBIS models into simulations.

The HyperLynx software is a PCB analysis and simulation tool for high-speed designs, consisting of two products, LineSim and BoardSim. LineSim is an early simulation tool. Before any board routing takes place, LineSim is used to simulate "what if" scenarios to assist in creating routing rules and defining board parameters. BoardSim is a post-layout tool used to analyze existing board routing. Specific nets are selected from a board layout file and simulated in a manner similar to LineSim. With board and routing parameters, and surrounding signal routing known, highly accurate simulations of the final fabricated PCB are possible. This section focuses on LineSim. Because the process of creating and running simulations is very similar for both LineSim and BoardSim, the details of IBIS model use in LineSim applies to simulations in BoardSim.

Simulations in LineSim are configured using a schematic GUI to create connections and topologies between I/O buffers, route trace segments, and termination components. LineSim provides two methods for creating routing schematics: cell-based and free-form. Cell-based schematics are based on fixed cells consisting of typical placements of buffers, trace impedances, and components. Parts of the grid-based cells are filled with the desired objects to create the topology. A topology in a cell-based schematic is limited by the available connections within and between the cells.

A more robust and expandable way to create a circuit schematic for simulation is to use the free-form schematic format in LineSim as shown in Figure 7–4. The free-form schematic format makes it easy to place parts into any configuration and edit them as required. This section describes the use of IBIS models with free-form schematics, but the process is nearly identical for cell-based schematics.

Figure 7–4. HyperLynx LineSim Free-Form Schematic Editor



When you use HyperLynx software to perform simulations, you typically perform the following steps:

1. Create a new LineSim free-form schematic document and set up the board stackup for your PCB using the Stackup Editor. In this editor, specify board layer properties including layer thickness, dielectric constant, and trace width.
2. Create a circuit schematic for the net you want to simulate. The schematic represents all the parts of the routed net including source and destination I/O buffers, termination components, transmission line segments, and representations of impedance discontinuities such as vias or connectors.
3. Assign IBIS models to the source and destination I/O buffers to represent their behavior during operation.
4. Attach probes from the digital oscilloscope that is built in to LineSim to points in the circuit that you want to monitor during simulation. Typically, at least one probe is attached to the pin of a destination I/O buffer. For differential signals, you can attach a differential probe to both the positive and negative pins at the destination.
5. Configure and run the simulation. You can simulate a rising or falling edge and test the circuit under different drive strength conditions.
6. Interpret the results and make adjustments. Based on the waveforms captured in the digital oscilloscope, you can adjust anything in the circuit schematic to correct any signal integrity issues, such as overshoot or ringing. If necessary, you can make I/O assignment changes in the Quartus II software, regenerate the IBIS file with the IBIS Writer, and apply the updated IBIS model to the buffers in your HyperLynx software schematic.
7. Repeat the simulations and circuit adjustments until you are satisfied with the results. When the operation of the net meets your design requirements, implement changes to your I/O assignments in the Quartus II software and/or adjust your board routing constraints, component values, and placement to match the simulation.

For more information about HyperLynx software, including schematic creation, simulation setup, model usage, product support, licensing, and training, refer to HyperLynx Help or the Mentor Graphics website at www.mentor.com.

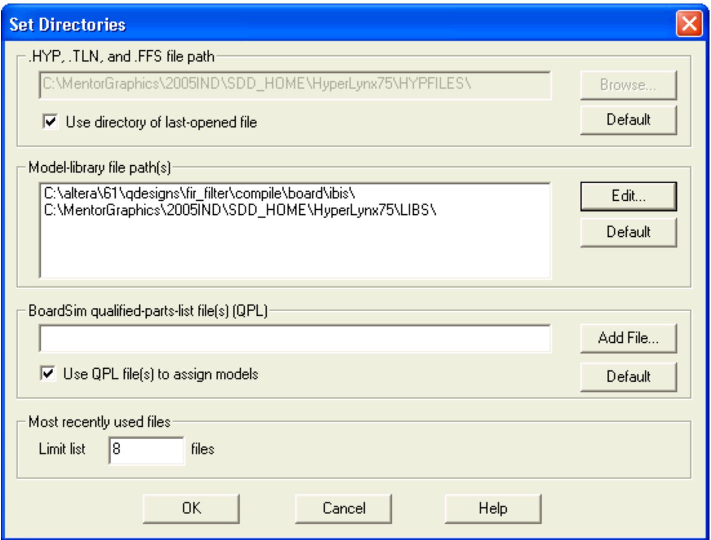
Configuring LineSim to Use Altera IBIS Models

You must configure LineSim to find and use the downloaded or generated IBIS models for your design. To do this, add the location of your **.ibs** file or files to the LineSim Model Library search path. Then you apply a selected model to a buffer in your schematic.

To add the Quartus II software's default IBIS model location, <*project directory*>**/board/ibis**, to the HyperLynx LineSim model library search path, perform the following steps in LineSim:

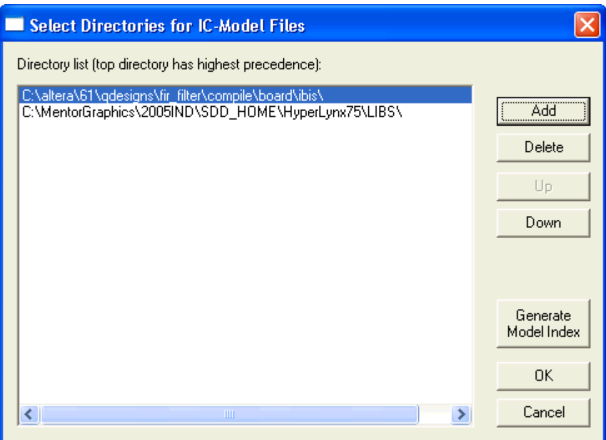
1. From the Options menu, click **Directories**. The **Set Directories** dialog box appears (Figure 7–5). The **Model-library file path(s)** list displays the order in which LineSim searches file directories for model files.

Figure 7–5. LineSim Set Directories Dialog Box



2. Click **Edit**. A dialog box appears where you can add directories and adjust the order in which LineSim searches them (Figure 7–6).

Figure 7–6. LineSim Select Directories Dialog Box



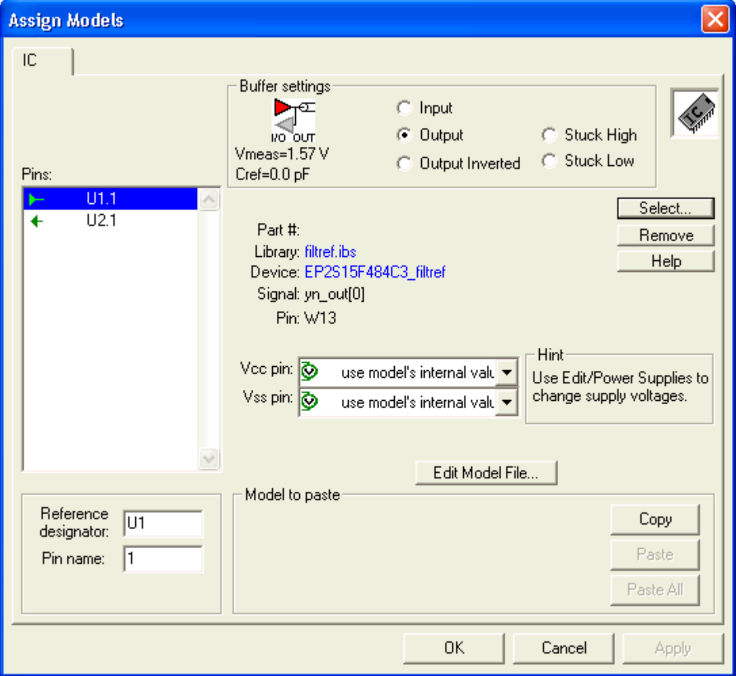
3. Click **Add**
4. Browse to the default IBIS model location, <*project directory*>**/board/ibis**. Click **OK**.
5. Click **Up** to move the IBIS model directory to the top of the list. Click **Generate Model Index** to update LineSim's model database with the models found in the added directory.
6. Click **OK**. The IBIS model directory for your project is added to the top of the Model-library file path(s) list.
7. To close the **Set Directories** dialog box, click **OK**.

Integrating Altera IBIS Models into LineSim Simulations

When the location for IBIS files has been set, you can assign the downloaded or generated IBIS models to the buffers in your schematic. To do this, perform the following steps:

1. Double-click a buffer symbol in your schematic to open the **Assign Models** dialog box (Figure 7–7). You can also click **Assign Models** from the buffer symbol's right-click menu.

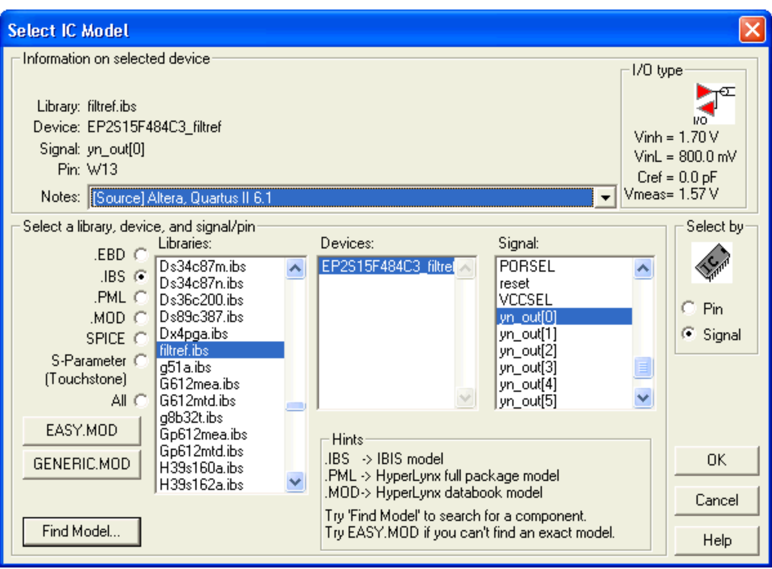
Figure 7–7. LineSim Assign Model Dialog Box



2. The pin of the buffer symbol you selected should be highlighted in the **Pins** list. If you want to assign a model to a different symbol or pin, select it from the list.

3. Click **Select**. The **Select IC Model** dialog box appears (Figure 7–8).

Figure 7–8. LineSim Select IC Model Dialog Box

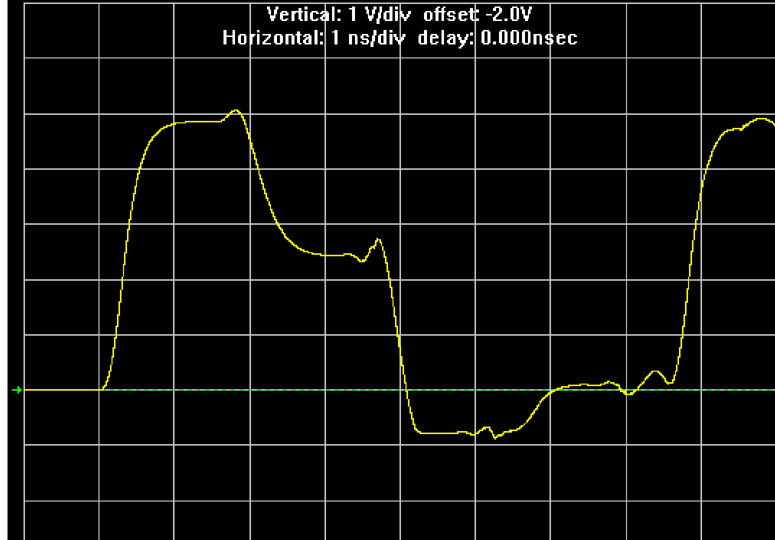


4. To filter the list of available libraries to display only IBIS models, select **.IBS**. Scroll through the **Libraries** list, and click the name of the library for your design. By default, this is <*project name*>**.ibs**.
5. The device for your design should be selected as the only item in the **Devices** list. If not, select your device from the list.
6. From the **Signal** list, select the name of the signal you want to simulate. You can also choose to select by device pin number.
7. Click **OK**. The **Assign Models** dialog box displays the selected **.ibs** file and signal.
8. If applicable to the signal you chose, adjust the buffer settings as required for the simulation.
9. Select and configure other buffer pins from the **Pins** list in the same manner.
10. Click **OK** when all I/O models are assigned.

Running and Interpreting LineSim Simulations

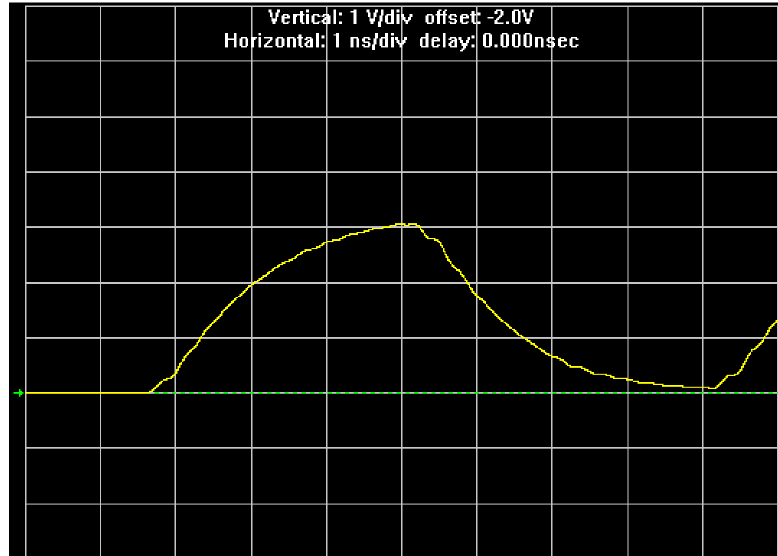
You can now run any desired simulations and make adjustments to the I/O assignments or simulation parameters as required. For example, if you see too much overshoot in the simulated signal at the destination buffer after running a simulation (as shown in Figure 7–9), you could adjust the drive strength I/O assignment setting to a lower value. Regenerate the **.ibs** file, and run the simulation again to verify whether the change fixed the problem.

Figure 7–9. Example of Overshoot in HyperLynx with IBIS Models



If you see a discontinuity or other anomalies at the destination, such as slow rise and fall times (as shown in Figure 7–10), adjust the termination scheme or termination component values. After making these changes, rerun the simulation to check whether your adjustments solved the problem. In this case, it is not necessary to regenerate the **.ibs** file.

Figure 7–10. Example of Signal Integrity Anomaly in HyperLynx with IBIS Models



For more information about board-level signal integrity and to learn about ways to improve it with simple changes to your design, visit the Altera Signal Integrity Center.

Simulation with HSPICE Models

HSPICE decks are used to perform highly accurate simulations by describing the physical properties of all aspects of a circuit precisely. HSPICE decks describe I/O buffers, board components, and all of the connections between them, as well as defining the parameters of the simulation to be run. By their nature, to be effective, HSPICE decks are highly customizable and require a detailed description of the circuit under simulation. For devices that support advanced I/O timing, when **Enable Advanced I/O Timing** is turned on, the HSPICE decks generated by the Quartus II HSPICE Writer automatically include board components and topology defined in the Board Trace Model. Configure the board components and topology in the Pin Planner or in the **Board Trace Model** tab of the **Device and Pin Options** dialog box. All HSPICE decks generated by the Quartus II software include compensation for the double count problem. For more information about the double count problem, refer to "The Double Counting Problem in HSPICE Simulations" on page 7–18. You can simulate with the default simulation parameters built in to the generated HSPICE decks or make adjustments to customize your simulation.

Supported Devices and Signaling

Beginning with Quartus II software version 6.1 and later, the HSPICE Writer supports the devices and signaling defined in Table 7–2. Only Stratix III, Stratix II, and Cyclone III devices support the creation of a board trace model in the Quartus II software for automatic inclusion in an HSPICE deck. Other devices require the board description to be manually added to the HSPICE file.

Table 7–2. HSPICE Writer Device and Signaling Support

Device	Input	Output	Single-Ended	Differential	Automatic Board Trace Model Description
Stratix III	✓	✓	✓	✓	✓
Stratix II GX (non-HSSI pins)	✓	✓	✓	✓	—
Stratix II	✓	✓	✓	✓	✓
HardCopy® II	✓	✓	✓	✓	—
Cyclone III	✓	✓	✓	✓	✓

If you are using a Stratix II device for your design, you can turn on **Enable Advanced I/O Timing** and configure the board trace model for each I/O standard used in your design. Newer families have this feature turned on by default and it cannot be turned off. The HSPICE files include the board trace description you create in the Board Trace Model view in the Pin Planner or the **Board Trace Model** tab in the **Device and Pin Options** dialog box.

For more information about the **Enable Advanced I/O Timing** option and configuring board trace models for the I/O standards in your design, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Accessing HSPICE Simulation Kits

You can access the available HSPICE models at the SPICE Models for Altera Devices web page and also with the Quartus II software's HSPICE Writer tool. The Quartus II software HSPICE Writer tool removes many common sources of user error from the I/O simulation process. The HSPICE Writer tool automatically creates preconfigured I/O simulation spice decks that only require the addition of a user board model. All the difficult tasks required to configure the I/O modes and interpret the timing results are handled automatically by the HSPICE Writer tool.

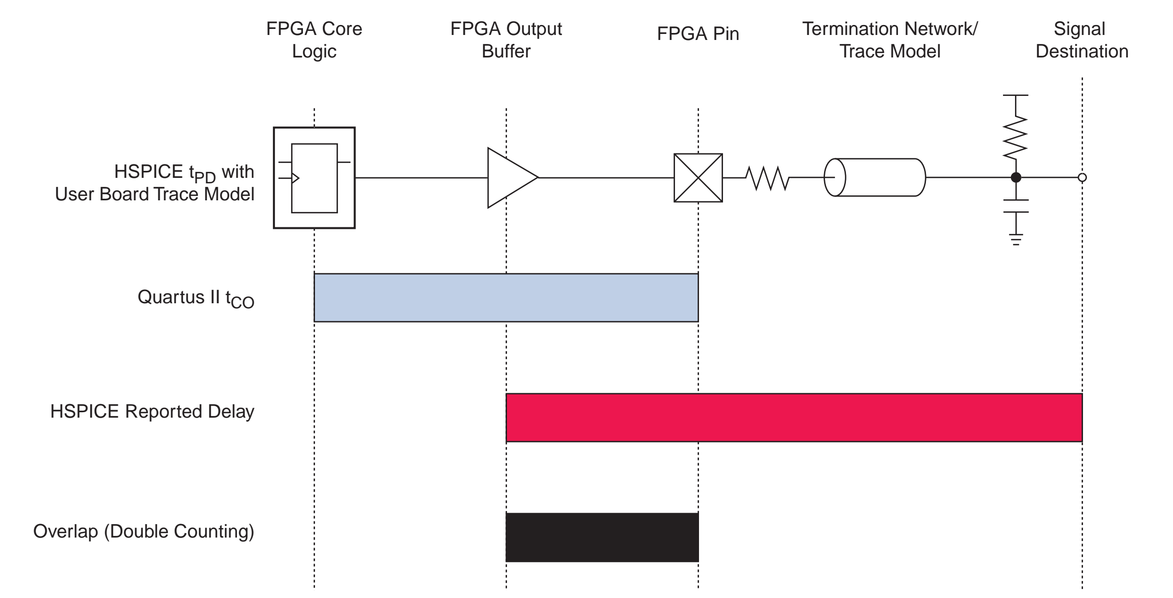
The Double Counting Problem in HSPICE Simulations

Simulating I/Os using accurate models is extremely helpful for finding and fixing FPGA I/O timing and board signal integrity issues before any boards are built. However, the usefulness of such simulations is directly related to the accuracy of the models used and whether the simulations are set up and performed correctly. To ensure accuracy in models and simulations created for FPGA output signals, the timing hand-off between t_{CO} timing in the Quartus II software and simulation-based board delay must be taken into account. If this hand-off is not handled correctly, the calculated delay could either count some of the delay twice or even miss counting some of the delay entirely.

Defining the Double Counting Problem

The double counting problem is inherent to the method output timing is analyzed versus the method used for HSPICE models. The timing analyzer tools in the Quartus II software measure delay timing for an output signal from the core logic of the FPGA design through the output buffer ending at the FPGA pin with a default capacitive load or a specified value for the selected I/O standard. This measurement is the t_{CO} timing variable as shown in Figure 7–11.

Figure 7–11. Double Counting Problem



HSPICE models for board simulation measure t_{PD} (propagation delay) from an arbitrary reference point in the output buffer, through the device pin, out along the board routing, and ending at the signal destination.

It is apparent immediately that if these two delays were simply added together, the delay between the output buffer and the device pin would be counted twice in the calculation. A model or simulation that does not account for this double count would create overly pessimistic simulation results, because the double-counted delay can limit I/O performance artificially. To fix the problem, it might seem that simply subtracting the overlap between t_{CO} and t_{PD} would account for the double count. However, this adjustment would not be accurate because each measurement is based on a different load.

Input signals do not exhibit this problem because the HSPICE models for inputs stop at the FPGA pin instead of at the input buffer. In this case, simply adding the delays together produces an accurate measurement of delay timing.

The Solution to Double Counting

To adjust the measurements to account for the double-counting, the delay between the arbitrary point in the output buffer selected by the HSPICE model and the FPGA pin must be subtracted from either t_{CO} or t_{PD} before adding the results together. The subtracted delay must also be based on a common load between the two measurements. This is done by repeating the HSPICE model measurement, but with the same load used by the Quartus II software for the t_{CO} measurement. This second measurement, called $t_{TESTLOAD}$, is illustrated with the top circuit in Figure 7–12.

Figure 7–12. Common Test Loads Used for Output Timing

FPGA Core Logic
FPGA Output Buffer
FPGA Pin
Quartus Test Load
Termination Network/ Trace Model
Signal Destination
HSPICE Netlist with Quartus Test Load
HSPICE Netlist with User Board Trace Model
Quartus II t_{CO}
HSPICE t_{PD} with User Specified Board Trace Model
Overlap (HSPICE Delay with Test Load)
HSPICE t_{PD} Adjusted by $t_{TESTLOAD}$
Total Delay

With $t_{TESTLOAD}$ known, the total delay for the output signal from the FPGA logic to the signal destination on the board, accounting for the double count, is calculated as shown in Equation 7–1.

Equation 7–1.

$$t_{delay} = t_{CO} + (t_{PD} - t_{TESTLOAD})$$

The preconfigured simulation files generated by the HSPICE Writer in the Quartus II software are designed to account for the double-counting problem based on this calculation automatically. Performing accurate timing simulations is easy without having to make adjustments for double counting manually.

HSPICE Writer Tool Flow

This section includes information to help you get started using the Quartus II software HSPICE Writer tool. The information in this section assumes you have a basic knowledge of the standard Quartus II software design flow, such as project and assignment creation, compilation, and timing analysis.

For additional information about standard design flows, refer to the appropriate sections of the *Quartus II Handbook*.

Applying I/O Assignments

The first step in the HSPICE Writer tool flow is to configure the I/O standards and modes for each of the pins in your design properly. In the Quartus II software, these settings are represented by assignments that map I/O settings, such as pin selection, and I/O standard and drive strength, to corresponding signals in your design.

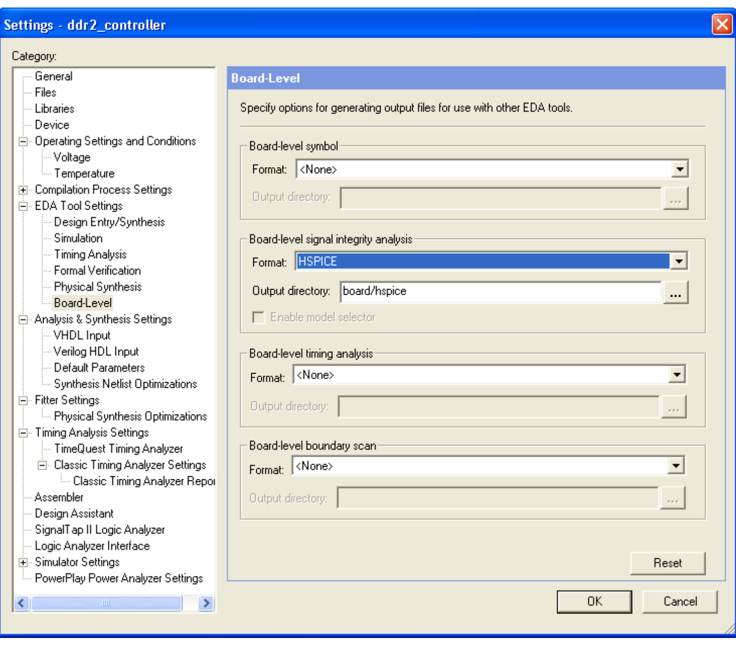
The Quartus II software provides multiple methods for creating these assignments:

- Using the Pin Planner
- Using the assignment editor
- Manually editing the **.qsf** file
- By making assignments in a scripted Quartus II flow using Tcl

Enabling HSPICE Writer

You must enable the HSPICE Writer in the **Settings** dialog box of the Quartus II software (Figure 7–13) to generate the HSPICE decks from the Quartus II software.

Figure 7–13. EDA Tool Settings: Board Level Options Dialog Box



Enabling HSPICE Writer Using Assignments

You can also use HSPICE Writer in conjunction with a scripted Tcl flow. To enable HSPICE Writer during a full compile, include the lines shown in Example 7–1 in your Tcl script.

Example 7–1. Enable HSPICE Writer

```
set_global_assignment -name EDA_BOARD_DESIGN_SIGNAL_INTEGRITY_TOOL \
"HSPICE (Signal Integrity)"

set_global_assignment -name EDA_OUTPUT_DATA_FORMAT HSPICE \
-section_id eda_board_design_signal_integrity

set_global_assignment -name EDA_NETLIST_WRITER_OUTPUT_DIR <output_directory> \
-section_id eda_board_design_signal_integrity
```

As with command-line invocation, specifying the output directory is optional. If not specified, the output directory defaults to **board/hspice**.

Naming Conventions for HSPICE Files

HSPICE Writer automatically generates simulation files and names them using the following naming convention:

*<device>_<pin #>_<pin_name>_<in/out>***.sp**

For bidirectional pins, two spice decks are produced; one with the I/O buffer configured as an input, and the other with the I/O buffer configured as an output.

The Quartus II software supports alphanumeric pin names that contain the underscore (_) and dash (-) characters. Any illegal characters used in file names are converted automatically to underscores.

The contents of the HSPICE files are described in detail in "Sample Output for I/O HSPICE Simulation Deck" on page 7–32 and "Sample Input for I/O HSPICE Simulation Deck" on page 7–28.

Invoking HSPICE Writer

After HSPICE Writer is enabled, the HSPICE simulation files are generated automatically each time the project is completely compiled. The Quartus II software also provides an option to generate a new set of simulation files without having to recompile manually. In the Processing menu, click **Start EDA Netlist Writer** to generate new simulation files automatically.

You must perform both Analysis & Synthesis and Fitting on a design before invoking the HSPICE Writer tool.

Invoking HSPICE Writer from the Command Line

If you use a script-based flow to compile your project, you can create HSPICE model files by including the commands shown in Example 7–2 in your Tcl script (**.tcl** file).

Example 7–2. Create HSPICE Model Files

```
set_global_assignment -name EDA_BOARD_DESIGN_SIGNAL_INTEGRITY_TOOL \
"HSPICE (Signal Integrity)"

set_global_assignment -name EDA_OUTPUT_DATA_FORMAT HSPICE \
-section_ideda_board_design_signal_integrity

set_global_assignment -name EDA_NETLIST_WRITER_OUTPUT_DIR <output_directory> \
-section_id eda_board_design_signal_integrity
```

The <*output_directory*> option specifies the location where HSPICE model files are saved. By default, the <*project directory*>**/board/hspice** directory is used.

To invoke the HSPICE Writer tool through the command line, type the syntax shown in Example 7–3.

Example 7–3. Invoke HSPICE Writer

```
quartus_eda.exe <project_name> --board_signal_integrity=on --format=HSPICE \
--output_directory=<output_directory>
```

<*output_directory*> specifies the location where the generated spice decks will be written (relative to the design directory). This is an optional parameter and defaults to **board/hspice**.

Customizing Automatically Generated HSPICE Decks

HSPICE models generated by the HSPICE Writer can be used for simulation as generated. A default board description is included, and a default simulation is set up to measure rise and fall delays for both input and output simulations, which compensates for the double counting problem. However, Altera recommends that you customize the board description to more accurately represent your routing and termination scheme.

The sample board trace loading in the generated HSPICE model files must be replaced by your actual trace model before you can run a correct simulation. To do this, open the generated HSPICE model files for all pins you want to simulate and locate the section shown in Example 7–4.

Example 7–4. Sample Board Trace Section

```
* I/O Board Trace and Termination Description
* - Replace this with your board trace and termination description
```

You must replace the example load with a load that matches the design of your PCB board. This includes a trace model, termination resistors, and, for output simulations, a receiver model. The spice circuit node that represents the pin of the FPGA package is called **pin**. The node that represents the far pin of the external device is called **load-in** (for output SPICE decks) and **source-in** (for input SPICE decks).

For an input simulation, you must also modify the stimulus portion of the spice file. The section of the file that must be modified is indicated in the comment block shown in Example 7–5.

Example 7–5. Sample Source Stimulus Section

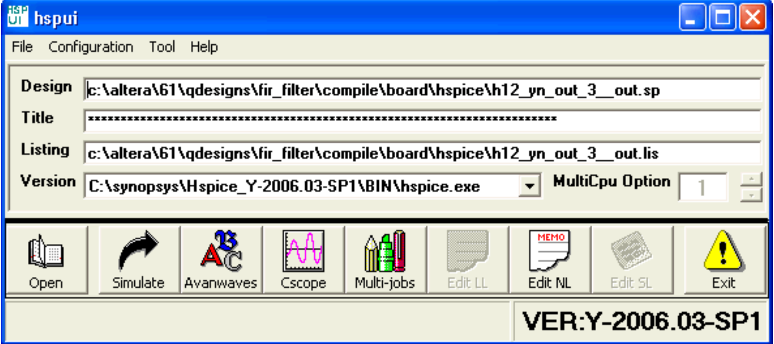
```
* Sample source stimulus placeholder
*  - Replace this with your I/O driver model
```

Replace the sample stimulus model with a model for the device that will drive the FPGA.

Running an HSPICE Simulation

Because simulation parameters are configured directly in the HSPICE model files, running a simulation requires only that you open an HSPICE file in the HSPICE user interface and start the simulation. The HSPICE user interface window is shown in Figure 7–14.

Figure 7–14. HSPICE User Interface Window



Click **Open** and browse to the location of the HSPICE model files generated by the Quartus II HSPICE Writer. The default location for HSPICE model files is <*project directory*>**/board/hspice**. Select the **.sp** file generated by the HSPICE Writer for the signal you want to simulate. Click **OK**.

To run the simulation, click **Simulate**. The status of the simulation is displayed in the window and saved in an **.lis** file with the same name as the **.sp** file when the simulation is complete. Check the **.lis** file if an error occurs during the simulation requiring a change in the **.sp** file to fix.

Interpreting the Results of an Output Simulation

By default, the automatically generated output simulation spice decks are set up to measure three delays for both rising and falling transitions. Two of the measurements, **tpd_rise** and **tpd_fall**, measure the double-counting corrected delay from the FPGA pin to the load pin. To determine the complete clock-edge to load-pin delay, add these numbers to the Quartus II software reported default loading t_{CO} delay.

The remaining four measurements, **tpd_uncomp_rise**, **tpd_uncomp_fall**, **t_dblcnt_rise**, and **t_dblcnt_fall**, are required for the double-counting compensation process and are not required for further timing usage. Refer to "Simulation Analysis" on page 7–32 for a description of these measurements.

Interpreting the Results of an Input Simulation

By default, the automatically generated input simulation SPICE decks are set up to measure delays from the source's driver pin to the FPGA's input pin for both rising and falling transitions. The propagation delay is reported by HSPICE measure statements as `tpd_rise` and `tpd_fall`. To determine the complete source driver pin-to-FPGA register delay, add these numbers to the Quartus II software reported T_H and T_{SU} input timing numbers.

Viewing and Interpreting Tabular Simulation Results

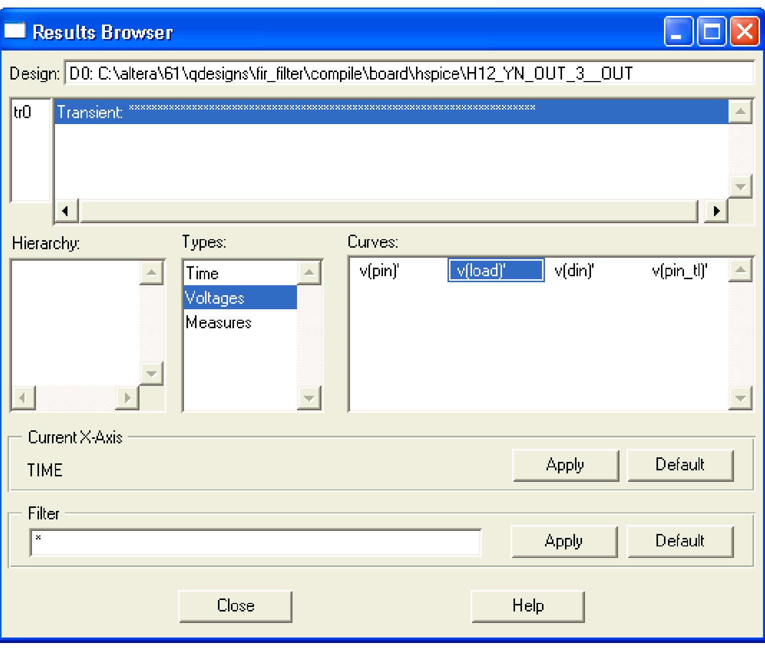
The **.lis** file stores the collected simulation data in tabular form. The default simulation configured by the HSPICE Writer produces delay measurements for rising and falling transitions on both input and output simulations. These measurements are found in the **.lis** file and named `tpd_rise` and `tpd_fall`. For output simulations, these values are already adjusted for the double count. To determine the complete delay from the FPGA logic to the load pin, add either of these measurements to the Quartus II t_{CO} delay. For input simulations, add either of these measurements to the Quartus II t_{SU} and t_H delay values to calculate the complete delay from the far end stimulus to the FPGA logic. Other values found in the **.lis** file, such as `tpd_uncomp_rise`, `tpd_uncomp_fall`, `t_dblcnt_rise`, and `t_dblcnt_fall`, are parts of the double count compensation calculation. These values are not necessary for further analysis.

Viewing Graphical Simulation Results

You can view the results of the simulation quickly as a graphical waveform display using the AvanWaves viewer included with HSPICE. With the default simulation configured by the HSPICE Writer, you can view the simulated waveforms at both the source and destination in input and output simulations.

To see the waveforms for the simulation, in the HSPICE user interface window, click **AvanWaves**. The AvanWaves viewer opens and displays the **Results Browser** as shown in Figure 7–15.

Figure 7–15. HSPICE AvanWaves Results Browser



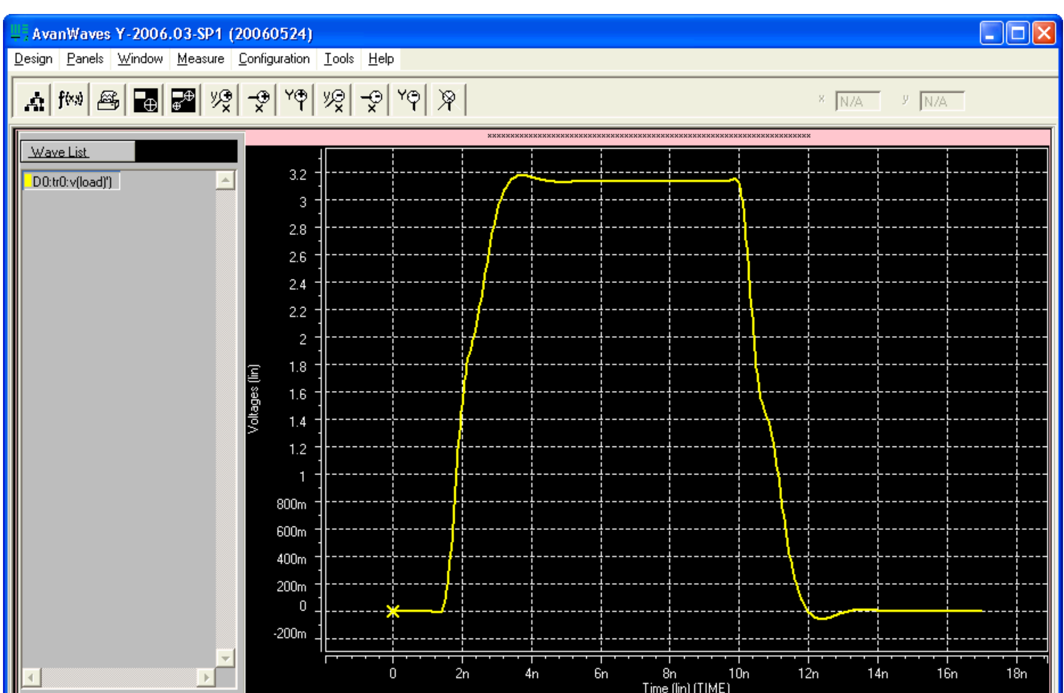
The **Results Browser** lets you select which waveform to view quickly in the main viewing window. If multiple simulations are run on the same signal, the list at the top of the **Results Browser** displays the results of each simulation. Click the simulation description to select which simulation to view. By default, the descriptions are derived from the first line of the HSPICE file, so the description might appear as a line of asterisks.

Select the type of waveform to view, by performing the following steps:

1. To see the source and destination waveforms with the default simulation, from the **Types** list, select **Voltages**.
2. On the **Curves** list, double-click the waveform you want to view. The waveform appears in the main viewing window.

You can zoom in and out and adjust the view as desired (Figure 7–16).

Figure 7–16. AvanWaves Waveform Viewer

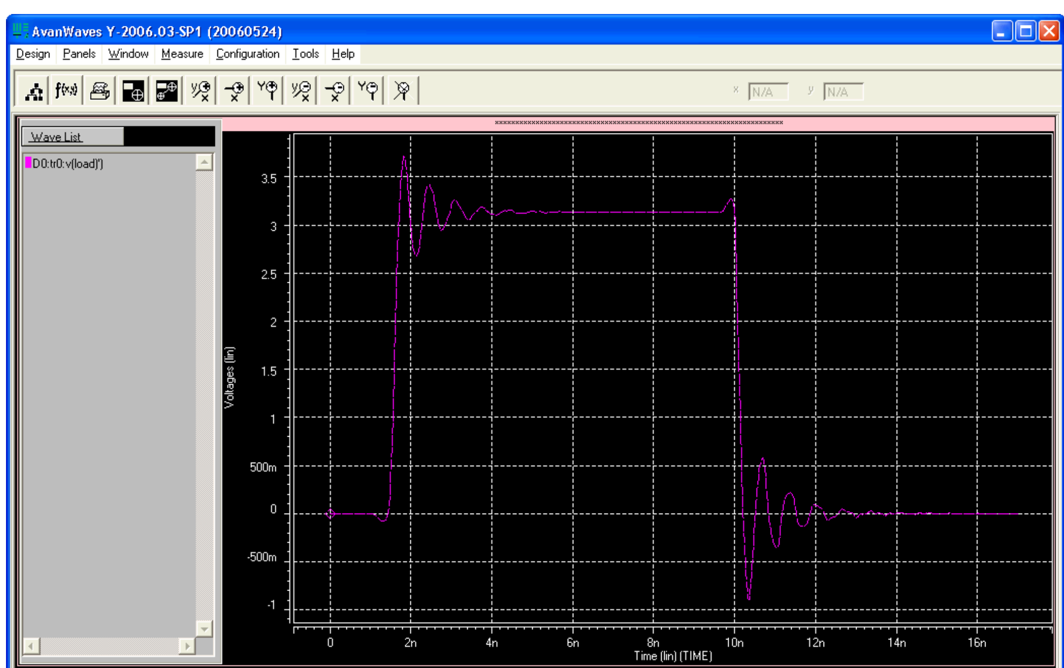


Making Design Adjustments Based on HSPICE Simulations

Based on the results of your simulations, you can make adjustments to the I/O assignments or simulation parameters if required. For example, after you run a simulation and see overshoot or ringing in the simulated signal at the destination buffer as shown in the example in Figure 7–17, you can adjust the drive strength I/O assignment setting to a lower value. Regenerate the HSPICE deck, and run the simulation again to verify that the change fixed the problem.

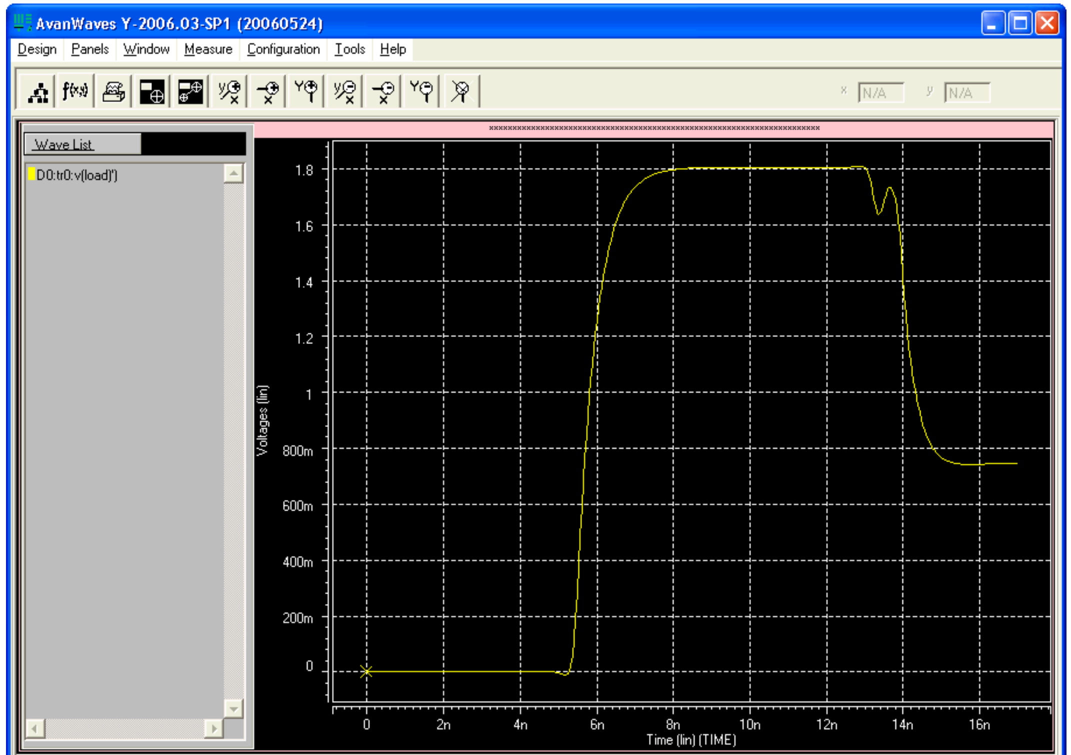
© March 2009 Altera Corporation

Figure 7–17. Example of Overshoot in the AvanWaves Waveform Viewer



If there is a discontinuity or any other anomalies at the destination as shown in the example in Figure 7–18, adjust the board description in the Quartus II Board Trace Model (for Stratix II, Stratix III, or Cyclone III devices) or in the generated HSPICE model files to change the termination scheme or adjust termination component values. After making these changes, regenerate the HSPICE files if necessary, and rerun the simulation to verify whether your adjustments solved the problem.

Figure 7–18. Example of Signal Integrity Anomaly in the AvanWaves Waveform Viewer



For more information about board-level signal integrity and to learn about ways to improve it with simple changes to your FPGA design, refer to the Altera Signal Integrity Center.

Sample Input for I/O HSPICE Simulation Deck

The following sections examine a typical HSPICE simulation spice deck for an I/O of type input. Each section presents the simulation file one block at a time.

Header Comment

The first block of an input simulation spice deck is the header comment. The purpose of this block is to provide an easily readable summary of how the simulation file has been automatically configured by the Quartus II software.

This block has two main components: The first component summarizes the I/O configuration relevant information such as device, speed grade, and so on. The second component specifies the exact test condition that the Quartus II software assumes for the given I/O standard. Example 7–6 shows a header comment block.

Example 7–6. Header Comment Block

```
* Quartus II HSPICE Writer I/O Simulation Deck*

* This spice simulation deck was automatically generated by
* Quartus for the following IO settings:
*
*  Device:       EP2S60F1020C3
*  Speed Grade:  C3
*  Pin:          AA4 (out96)
*  Bank:         IO Bank 6 (Row I/O)
*  I/O Standard: LVTTL, 12mA
*  OCT:          Off
*
* Quartus II's default I/O timing delays assume the following slow
* corner simulation conditions.
*
*  Specified Test Conditions For Quartus II Tco
*    Temperature:      85C (Slowest Temperature Corner)
*    Transistor Model: TT (Typical Transistor Corner)
*    Vccn:             3.135V (Vccn_min = Nominal - 5%)
*    Vccpd:            2.97V (Vccpd_min = Nominal - 10%)
*    Load:             No Load
*    Vtt:              1.5675V (Voltage reference is Vccn/2)
*
* Note: The I/O transistors are specified to operate at least as
*       fast as the TT transistor corner, actual production
*       devices can be as fast as the FF corner. Any simulations
*       for hold times should be conducted using the fast process
*       corner with the following simulation conditions.
*        Temperature:     0C (Fastest Commercial Temperature Corner **)
*        Transistor Model: FF (Fastest Transistor Corner)
*        Vccn:             1.98V (Vccn_hold = Nominal + 10%)
*        Vccpd:            3.63V (Vccpd_hold = Nominal + 10%)
*        Vtt:              0.95V (Vtt_hold = Vccn/2 - 40mV)
*        Vcc:              1.25V (Vcc_hold = Maximum Recommended)
*        Package Model:   Short-circuit from pad to pin (no parasitics)
*
* Warnings:
```

Simulation Conditions

The simulation conditions block loads the appropriate process corner models for the transistors. This condition is automatically set up for the slow timing corner and is modified only if other simulation corners are desired. Example 7–7 shows a simulation conditions block.

Example 7–7. Simulation Conditions Block

```
* Process Settings

.options brief
.inc 'sii_tt.inc' * TT process corner
```

Simulation Options

The simulation options block configures the simulation temperature and configures HSPICE with typical simulation options. Example 7–8 shows a simulation options block.

For a detailed description of these options, consult your *HSPICE* manual.

Example 7–8. Simulation Options Block

```
* Simulation Options

.options brief=0
.options badchr co=132 scale=1e-6 acct ingold=2 nomod dv=1.0
+        dcstep=1 absv=1e-3 absi=1e-8 probe csdf=2 accurate=1
+        converge=1
.temp 85
```

Constant Definition

The constant definition block of the simulation file instantiates the voltage sources that controls the configuration modes of the I/O buffer. Example 7–9 shows a constant definition block.

Example 7–9. Constant Definition Block

```
* Constant Definition

voeb       oeb       0     vc  * Set to 0 to enable buffer output
vopdrain   opdrain   0     0   * Set to vc to enable open drain
vrambh     rambh     0     0   * Set to vc to enable bus hold
vrpullup   rpullup   0     0   * Set to vc to enable weak pullup
vpcdp5     rpcdp5    0     rp5 * Set the IO standard
vpcdp4     rpcdp4    0     rp4
vpcdp3     rpcdp3    0     rp3
vpcdp2     rpcdp2    0     rp2
vpcdp1     rpcdp1    0     rp1
vpcdp0     rpcdp0    0     rp0
vpcdn4     rpcdn4    0     rn4
vpcdn3     rpcdn3    0     rn3
vpcdn2     rpcdn2    0     rn2
vpcdn1     rpcdn1    0     rn1
vpcdn0     rpcdn0    0     rn0
vdin din       0     0
```

Where:

- Voltage source `voeb` controls the output enable of the buffer and is set to *disabled* for inputs.
- `vopdrain` controls the open drain mode for the I/O.
- `vrambh` controls the bus hold circuitry in the I/O.
- `vrpullup` controls the weak pullup.
- The next 11 voltages sources control the I/O standard of the buffer and are configured through a later library call.
- `vdin` is not used on input pins because it is the data pin for the output buffer.

Buffer Netlist

The buffer netlist block (Example 7–10) of the simulation spice deck loads all the load models required for the corresponding input pin.

Example 7–10. Buffer Netlist Block

```
* IO Buffer Netlist

.include 'vio_buffer.inc'
```

Drive Strength

The drive strength block (Example 7–11) of the simulation SPICE deck loads the configuration bits necessary to configure the I/O into the proper I/O standard and drive strengths. Although these settings are not relevant to an input buffer, they are provided to allow the SPICE deck to be modifiable to support bidirectional simulations.

Example 7–11. Drive Strength Block

```
* Drive Strength Settings

.lib 'drive_select_hio.lib' 3p3ttl_12ma
```

I/O Buffer Instantiation

The I/O buffer instantiation block of the simulation SPICE deck instantiates the necessary power supplies and I/O model components that are necessary to simulate the given I/O.

Example 7–12 shows I/O buffer instantiation.

Example 7–12. I/O Buffer Instantiation

```
I/O Buffer Instantiation

* Supply Voltages Settings
.param vcn=3.135
.param vpd=2.97
.param vc=1.15

* Instantiate Power Supplies|
vvcc       vcc      0     vc     * FPGA core voltage
vvss       vss      0     0      * FPGA core ground
vvccn      vccn     0     vcn    * IO supply voltage
vvssn      vssn     0     0      * IO ground
vvccpd     vccpd    0     vpd    * Pre-drive supply voltage

* Instantiate I/O Buffer
xvio_buf din oeb opdrain die rambh
+ rpcdn4 rpcdn3 rpcdn2 rpcdn1 rpcdn0
+ rpcdp5 rpcdp4 rpcdp3 rpcdp2 rpcdp1 rpcdp0
+ rpullup vccn vccpd vcpad0 vio_buf

* Internal Loading on Pad
* - No loading on this pad due to differential buffer/support
*   circuitry

* I/O Buffer Package Model
* - Single-ended I/O standard on a Row I/O
.lib 'lib/package.lib' hio
xpkg die pin hio_pkg
```

Board Trace and Termination

The board trace and termination block of the simulation SPICE deck is provided only as an example (shown in Example 7–13). Replace this block with your own board trace and termination models.

Example 7–13. Board Trace and Termination Block

```
* I/O Board Trace and Termination Description
* - Replace this with your board trace and termination description

wtline pin vssn load vssn N=1 L=1 RLGCMODEL=tlinemodel
.MODEL tlinemodel W MODELTYPE=RLGC N=1 Lo=7.13n Co=2.85p
Rterm2 load vssn 1x
```

Stimulus Model

The stimulus model block of the simulation spice deck is provided only as a place holder example (shown in Example 7–14). Replace this block with your own stimulus model. Options for this include an IBIS or HSPICE model, among others.

Example 7–14. Stimulus Model Block

```
* Sample source stimulus placeholder
* - Replace this with your I/O driver model

Vsource source 0 pulse(0 vcn 0s 0.4ns 0.4ns 8.5ns 17.4ns)
```

Simulation Analysis

The simulation analysis block (Example 7–15) of the simulation file is configured to measure the propagation delay from the source to the FPGA pin. Both the source and end point of the delay are referenced against the 50% V_{CCN} crossing point of the waveform.

Example 7–15. Simulation Analysis Block

```
* Simulation Analysis Setup

* Print out the voltage waveform at both the source and the pin
.print tran v(source)  v(pin)
.tran 0.020ns 17ns

* Measure the propagation delay from the source pin to the pin
* referenced against the 50% voltage threshold crossing point

.measure TRAN tpd_rise TRIG v(source) val='vcn*0.5' rise=1
+ TARG v(pin) val ='vcn*0.5' rise=1
.measure TRAN tpd_fall TRIG v(source) val='vcn*0.5' fall=1
+ TARG v(pin) val ='vcn*0.5' fall=1
```

Sample Output for I/O HSPICE Simulation Deck

The following sections examine a typical HSPICE simulation SPICE deck for an I/O-type output. Each section presents the simulation file one block at a time.

Header Comment

The first block of an output simulation SPICE deck is the header comment, as shown in Example 7–16. The purpose of this block is to provide a readable summary of how the simulation file has been automatically configured by the Quartus II software.

This block has two main components:

- The first component summarizes the I/O configuration relevant information such as device, speed grade, and so on.
- The second component specifies the exact test condition that the Quartus II software assumes when generating t_{CO} delay numbers. This information is used as part of the double-counting correction circuitry contained in the simulation file.

The SPICE decks are preconfigured to calculate the slow process corner delay but can also be used to simulate the fast process corner as well. The fast corner conditions are listed in the header under the notes section.

The final section of the header comment lists any warning messages that you must consider when you use the SPICE decks.

Example 7–16. Header Comment Block

```
* Quartus II HSPICE Writer I/O Simulation Deck
*
* This spice simulation deck was automatically generated by
* Quartus II for the following IO settings:
*
*  Device:       EP2S60F1020C3
*  Speed Grade:  C3
*  Pin:          AA4 (out96)
*  Bank:         IO Bank 6 (Row I/O)
*   I/O Standard: LVTTL, 12mA
*  OCT:          Off
*
* Quartus' default I/O timing delays assume the following slow
* corner simulation conditions.
*  Specified Test Conditions For Quartus II Tco
*    Temperature:      85C (Slowest Temperature Corner)
*    Transistor Model: TT (Typical Transistor Corner)
*    Vccn:             3.135V (Vccn_min = Nominal - 5%)
*    Vccpd:            2.97V (Vccpd_min = Nominal - 10%)
*    Load:             No Load
*    Vtt:              1.5675V (Voltage reference is Vccn/2)
* For C3 devices, the TT transistor corner provides an
* approximation for worst case timing. However, for functionality
* simulations, it is recommended that the SS corner be simulated
* as well.
*
* Note: The I/O transistors are specified to operate at least as
*       fast as the TT transistor corner, actual production
*       devices can be as fast as the FF corner. Any simulations
*       for hold times should be conducted using the fast process
*       corner with the following simulation conditions.
*         Temperature:      0C (Fastest Commercial Temperature Corner
**)
*         Transistor Model: FF (Fastest Transistor Corner)
*         Vccn:             1.98V (Vccn_hold = Nominal + 10%)
*         Vccpd:            3.63V (Vccpd_hold = Nominal + 10%)
*         Vtt:              0.95V (Vtt_hold = Vccn/2 - 40mV)
*         Vcc:              1.25V (Vcc_hold = Maximum Recommended)
*         Package Model:    Short-circuit from pad to pin
* Warnings:
```

Simulation Conditions

The simulation conditions block (Example 7–17) loads the appropriate process corner models for the transistors. This condition is automatically set up for the slow timing corner and must be modified only if other simulation corners are desired.

Two separate corners cannot be simulated at the same time. Instead, simulate the base case using the Quartus corner as one simulation and then perform a second simulation using the desired customer corner. The results of the two simulations can be manually added together.

 © March 2009 Altera Corporation

Example 7–17. Simulation Conditions Block

```
* Process Settings

.options brief
.inc ‘sii_tt.inc’ * typical-typical process corner
```

Simulation Options

The simulation options block (Example 7–18) configures the simulation temperature and configures HSPICE with typical simulation options.

For a detailed description of these options, consult your *HSPICE* manual.

Example 7–18. Simulation Options Block

```
* Simulation Options
.options brief=0
.options badchr co=132 scale=1e-6 acct ingold=2 nomod dv=1.0
+        dcstep=1 absv=1e-3 absi=1e-8 probe csdf=2 accurate=1
+        converge=1
.temp 85
```

Constraint Definition

The constant definition block (Example 7–19) of the output simulation SPICE deck instantiates the voltage sources that controls the configuration modes of the I/O buffer.

Example 7–19. Constant Definition Block

```
* Constant Definition

voeb        oeb       0     0 * Set to 0 to enable buffer output
vopdrain    opdrain   0     0 * Set to vc to enable open drain
vrambh      rambh     0     0 * Set to vc to enable bus hold
vrpullup    rpullup   0     0 * Set to vc to enable weak pullup
vpci        rpci      0     0 * Set to vc to enable pci mode
vpcdp4      rpcdp4    0     rp4  * These control bits set the IO standard
vpcdp3      rpcdp3    0     rp3
vpcdp2      rpcdp2    0     rp2
vpcdp1      rpcdp1    0     rp1
vpcdp0      rpcdp0    0     rp0
vpcdn4      rpcdn4    0     rn4
vpcdn3      rpcdn3    0     rn3
vpcdn2      rpcdn2    0     rn2
vpcdn1      rpcdn1    0     rn1
vpcdn0      rpcdn0    0     rn0
vdin        din       0     pulse(0 vc 0s 0.2ns 0.2ns 8.5ns 17.4ns)
```

Where:

- Voltage source `voeb` controls the output enable of the buffer.
- `vopdrain` controls the open drain mode for the I/O.
- `vrambh` controls the bus hold circuitry in the I/O.
- `vrpullup` controls the weak pullup.
- `vpci` controls the PCI clamp.
- The next ten voltage sources control the I/O standard of the buffer and are configured through a later library call. Stratix III and Cyclone III devices have more bits and so might have more voltage sources listed in the constant definition block. They also have slew rate and delay chain settings.
- `vdin` is connected to the data input of the I/O buffer.
- The edge rate of the input stimulus is automatically set to the correct value by the Quartus II software.

I/O Buffer Netlist

The I/O buffer netlist block (Example 7–20) loads all of the models required for the corresponding pin. These include a model for the I/O output buffer, as well as any loads that might be present on the pin.

Example 7–20. I/O Buffer Netlist Block

```
*IO Buffer Netlist

.include 'hio_buffer.inc'
.include 'lvds_input_load.inc'
.include 'lvds_oct_load.inc'
```

Drive Strength

The drive strength block (Example 7–21) of the simulation spice deck loads the configuration bits for configuring the I/O to the proper I/O standard and drive strength. These options are set by the HSPICE Writer tool and are not changed for expected use.

Example 7–21. Drive Strength Block

```
* Drive Strength Settings

.lib 'drive_select_hio.lib' 3p3ttl_12ma
```

Slew Rate and Delay Chain

Stratix III and Cyclone III devices have sections for configuring the slew rate and delay chain settings (Example 7–22).

Example 7–22. Slew Rate and Delay Chain Settings

```
* Programmable Output Delay Control Settings

.lib 'lib/output_delay_control.lib' no_delay

* Programmable Slew Rate Control Settings

.lib 'lib/slew_rate_control.lib' slow_slow
```

I/O Buffer Instantiation

The I/O buffer instantiation block (Example 7–23) of the output simulation spice deck instantiates the necessary power supplies and I/O model components that are necessary to simulate the given I/O.

Example 7–23. I/O Buffer Instantiation Block

```
* I/O Buffer Instantiation

* Supply Voltages Settings
.param vcn=3.135
.param vpd=2.97
.param vc=1.15

* Instantiate Power Supplies
vvcc       vcc       0     vc     * FPGA core voltage
vvss       vss       0     0      * FPGA core ground
vvccn      vccn      0     vcn    * IO supply voltage
vvssn      vssn      0     0      * IO ground
vvccpd     vccpd     0     vpd    * Pre-drive supply voltage

* Instantiate I/O Buffer
xhio_buf din oeb opdrain die rambh
+ rpcdn4 rpcdn3 rpcdn2 rpcdn1 rpcdn0
+ rpcdp4 rpcdp3 rpcdp2 rpcdp1 rpcdp0
+ rpullup vccn vccpd vcpad0 hio_buf

* Internal Loading on Pad
* - This pad has an LVDS input buffer connected to it, along
*   with differential OCT circuitry. Both are disabled but
*   introduce loading on the pad that is modeled below.
xlvds_input_load die vss vccn lvds_input_load
xlvds_oct_load die vss vccpd vccn vcpad0 vccn lvds_oct_load

* I/O Buffer Package Model
* - Single-ended I/O standard on a Row I/O
.lib 'lib/package.lib' hio
xpkg die pin hio_pkg
```

Board and Trace Termination

The board trace and termination block (Example 7–24) of the simulation SPICE deck is provided only as an example. Replace this block with your specific board loading models.

Example 7–24. Board Trace and Termination Block

```
* I/O Board Trace And Termination Description
* - Replace this with your board trace and termination description

wtline pin vssn load vssn N=1 L=1 RLGCMODEL=tlinemodel
.MODEL tlinemodel W MODELTYPE=RLGC N=1 Lo=7.13n Co=2.85p
Rterm2 load vssn 1x
```

Double-Counting Compensation Circuitry

The double-counting compensation circuitry block (Example 7–25) of the simulation SPICE deck instantiates a second I/O buffer that is used to measure double-counting. The buffer is configured identically to the user I/O buffer but is connected to the Quartus II software test load. The simulated delay of this second buffer can be interpreted as the amount of double-counting between the Quartus II software and HSPICE Writer simulated results.

As the amount of double-counting is constant for a given I/O standard on a given pin, consider separating the double-counting circuitry from the simulation file. In doing so, you can perform any number of I/O simulations while referencing the delay only once. For more information about the double-counting problem, refer to "The Double Counting Problem in HSPICE Simulations" on page 7–18.

Example 7–25. Double-Counting Compensation Circuitry Block

```
* Double Counting Compensation Circuitry
*
* The following circuit is designed to calculate the amount of
* double counting between Quartus II and the HSPICE models. If
* you have not changed the default simulation temperature or
* transistor corner the double counting will be automatically
* compensated by this spice deck. In the event you wish to
* simulate an IO at a different temperature or transistor corner
* you will need to remove this section of code and manually
* account for double counting. A description of Altera's
* recommended procedure for this process can be found in the
* Quartus II HSPICE Writer AppNote.
*

* Supply Voltages Settings
.param vcn_tl=3.135
.param vpd_tl=2.97

* Test Load Constant Definition
vopdrain_tl   opdrain_tl   0     0
vrambh_tl     rambh_tl     0     0
vrpullup_tl   rpullup_tl   0     0

* Instantiate Power Supplies
vvccn_tl      vccn_tl      0     vcn_tl
vvssn_tl      vssn_tl      0     0
vvccpd_tl     vccpd_tl     0     vpd_tl

* Instantiate I/O Buffer
xhio_testload din oeb opdrain_tl die_tl rambh_tl
+ rpcdn4 rpcdn3 rpcdn2 rpcdn1 rpcdn0
+ rpcdp4 rpcdp3 rpcdp2 rpcdp1 rpcdp0
+ rpullup_tl vccn_tl vccpd_tl vcpad0_tl hio_buf

* Internal Loading on Pad
xlvds_input_testload die_tl vss vccn_tl lvds_input_load
xlvds_oct_testload die_tl vss vccpd_tl vccn_tl vcpad0_tl vccn_tl
lvds_oct_load


* I/O Buffer Package Model
* - Single-ended I/O standard on a Row I/O
.lib 'lib/package.lib' hio
xpkg die pin hio_pkg

* Default Altera Test Load
* - 3.3V LVTTL default test condition is an open load
```

Simulation Analysis

The simulation analysis block (Example 7–26) is set up to measure double-counting corrected delays. This is accomplished by measuring the uncompensated delay of the I/O buffer when connected to the user load, and when subtracting the simulated amount of double-counting from the test load I/O buffer.

Example 7–26. Simulation Analysis Block

```
*Simulation Analysis Setup

* Print out the voltage waveform at both the pin and far end load
.print tran v(pin) v(load)
.tran 0.020ns 17ns

* Measure the propagation delay to the load pin. This value will
* include some double counting with Quartus II's Tco
.measure TRAN tpd_uncomp_rise TRIG v(din) val='vc*0.5' rise=1
+ TARG v(load) val='vcn*0.5' rise=1
.measure TRAN tpd_uncomp_fall TRIG v(din) val='vc*0.5' fall=1
+ TARG v(load) val='vcn*0.5' fall=1

* The test load buffer can calculate the amount of double counting
.measure TRAN t_dblcnt_rise TRIG v(din) val='vc*0.5' rise=1
+ TARG v(pin_tl) val='vcn_tl*0.5' rise=1
.measure TRAN t_dblcnt_fall TRIG v(din) val='vc*0.5' fall=1
+ TARG v(pin_tl) val='vcn_tl*0.5' fall=1

* Calculate the true propagation delay by subtraction
.measure TRAN tpd_rise PARAM='tpd_uncomp_rise-t_dblcnt_rise'
.measure TRAN tpd_fall PARAM='tpd_uncomp_fall-t_dblcnt_fall'
```

Advanced Topics

The information in this section describes some of the more advanced topics and methods employed when setting up and running HSPICE simulation files.

PVT Simulations

The automatically generated HSPICE simulation files are set up to simulate the slow process corner using low voltage, high temperature, and slow transistors. To ensure a fully robust link, Altera recommends that you run simulations over all process corners.

To perform process, voltage, and temperature (PVT) simulations, manually modify the spice decks in a two step process:

1. Remove the double-counting compensation circuitry from the simulation file. This is required as the amount of double-counting is dependant upon how the Quartus II software calculates delays and is not based on which PVT corner is being simulated. By default, the Quartus II software provides timing numbers using the slow process corner.
2. Select the proper corner for the PVT simulation by setting the correct HSPICE temperature, changing the supply voltage sources, and loading the correct transistor models.

A more detailed description of HSPICE process corners can be found in the family-specific HSPICE model documentation. This document is available online with the HSPICE models as described in "Accessing HSPICE Simulation Kits" on page 7–18.

Hold Time Analysis

Altera recommends performing worst-case hold time analysis using the fast corner models, which use fast transistors, high voltage, and low temperature. This involves modifying the SPICE decks to select the correct temperature option, change the supply voltage sources, and load the correct fast transistor models. The values of these parameters are located in the header comment section of the corresponding simulation deck files.

For a truly worst-case analysis, combine the HSPICE Writer hold time analysis results with the Quartus II software fast timing model. This requires that you change the double-counting compensation circuitry in the simulations files to also simulate the fast process corners, as this is what the Quartus II software uses for the fast timing model.

This method of hold time analysis is recommended only for globally synchronous buses. Do not apply this method of hold-time analysis to source synchronous buses. This is because the source synchronous clocking scheme is designed to cancel out some of the PVT timing effects. If this is not taken into account, the timing results will not be accurate. Proper source synchronous timing analysis is beyond the scope of this document.

I/O Voltage Variations

Use each of the FPGA family datasheets to verify the recommended operating conditions for supply voltages. For current FPGA families, the maximum recommended voltage corresponds to the fast corner, while the minimum recommended voltage corresponds to the slow corner. These voltage recommendations are specified at the power pins of the FPGA and are not necessarily the same voltage that are seen by the I/O buffers due to package IR drops.

The automatically generated HSPICE simulation files model this IR effect pessimistically by including a 50-mV IR drop on the V_{CCPD} supply when a high drive strength standard is being used.

Correlation Report

Correlation reports for the HSPICE I/O models are located in the family-specific HSPICE I/O buffer simulation kits. Refer to "Accessing HSPICE Simulation Kits" on page 7–18 for additional information.

Conclusion

As FPGA devices are used in more high-speed applications, it becomes increasingly necessary to perform board-level signal integrity analysis simulations. You must run such simulations to ensure good signal integrity between the FPGA and any connected devices. The Quartus II software helps to simplify this process with the ability to automatically generate I/O buffer description models easily with the IBIS and HSPICE Writers. IBIS models can be integrated into a third-party signal integrity analysis workflow using a tool such as Mentor Graphics HyperLynx software,

generating quick and accurate simulation results. HSPICE decks include preconfigured simulations and only require descriptions of board routing and stimulus models to create highly accurate simulation results using Synopsys HSPICE. Either type of simulation helps prevent unnecessary board spins, increasing your productivity and decreasing your costs.

Referenced Documents

This chapter references the following documents:

- *AN 283: Simulating Altera Devices with IBIS Models*
- *AN 315: Guidelines for Designing High-Speed FPGA PCBs*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Handbook*

Document Revision History

Table 7–3 shows the revision history for this chapter.

Table 7–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Was volume 3, chapter 12 in the 8.1.0 release. ■ No change to content.	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	■ Changed to 8-1/2 x 11 page size. ■ Added information for Stratix III devices. ■ Input signals for Cyclone III devices are supported.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Updated "Introduction" on page 12–1. ■ Updated Figure 12–1. ■ Updated Figure 12–3. ■ Updated Figure 12–13. ■ Updated "Output File Generation" on page 12–6. ■ Updated "Simulation with HSPICE Models" on page 12–17. ■ Updated "Invoking HSPICE Writer from the Command Line" on page 12–22. ■ Added "Sample Input for I/O HSPICE Simulation Deck" on page 12–29. ■ Added "Sample Output for I/O HSPICE Simulation Deck" on page 12–33. ■ Updated "Correlation Report" on page 12–41. ■ Added hyperlinks to referenced documents and websites throughout the chapter. ■ Made minor editorial updates.	Updated for the Quartus II software version 8.0.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

8. Mentor Graphics PCB Design Tools Support

QII52015-9.0.0

Introduction

With today's large, high-pin-count and high-speed FPGA devices, good and correct printed circuit board (PCB) design practices are more essential than ever for ensuring correct system operation. Typically, the PCB design takes place concurrently with the design and programming of the FPGA. Signal and pin assignments are initially made by the FPGA or ASIC designer, and the board designer must correctly transfer these assignments to the symbols used in their system circuit schematics and board layout. As the board design progresses, pin reassignments may be needed to optimize the PCB layout. These reassignments must in turn be relayed back to the FPGA designer so that the new assignments can be processed through an updated placement and routing of the FPGA design.

Mentor Graphics® provides tools to support this type of design flow. This chapter discusses how the Quartus® II software interacts with the Mentor Graphics I/O Designer software and the DxDesigner software to provide a completely cyclical FPGA-to-board integration design workflow. This chapter covers the following topics:

- General design flow between the Quartus II software, the Mentor Graphics I/O Designer software, and the DxDesigner software
- Setting up the Quartus II software to create the design flow files
- Creating an I/O Designer database project to incorporate the Quartus II software signal and pin assignment data
- Updating signal and pin assignment changes between the I/O Designer software and the Quartus II software
- Generating symbols in the I/O Designer software
- Creating symbols in the DxDesigner software from the Quartus II software output files without the use of the I/O Designer software

This chapter is intended primarily for board design and layout engineers who want to start the FPGA board integration while the FPGA is still in the design phase. Optionally, the board designer can plan the FPGA pinout and routing requirements in the Mentor Graphics tools and pass the information back to the Quartus II software for place-and-route. In addition, part librarians benefit from learning how to take output from the Quartus II software and use it to create new library parts and symbols.

The procedures in this chapter require the following software:

- The Quartus II software version 5.1 or higher
- DxDesigner software version 2004 or higher

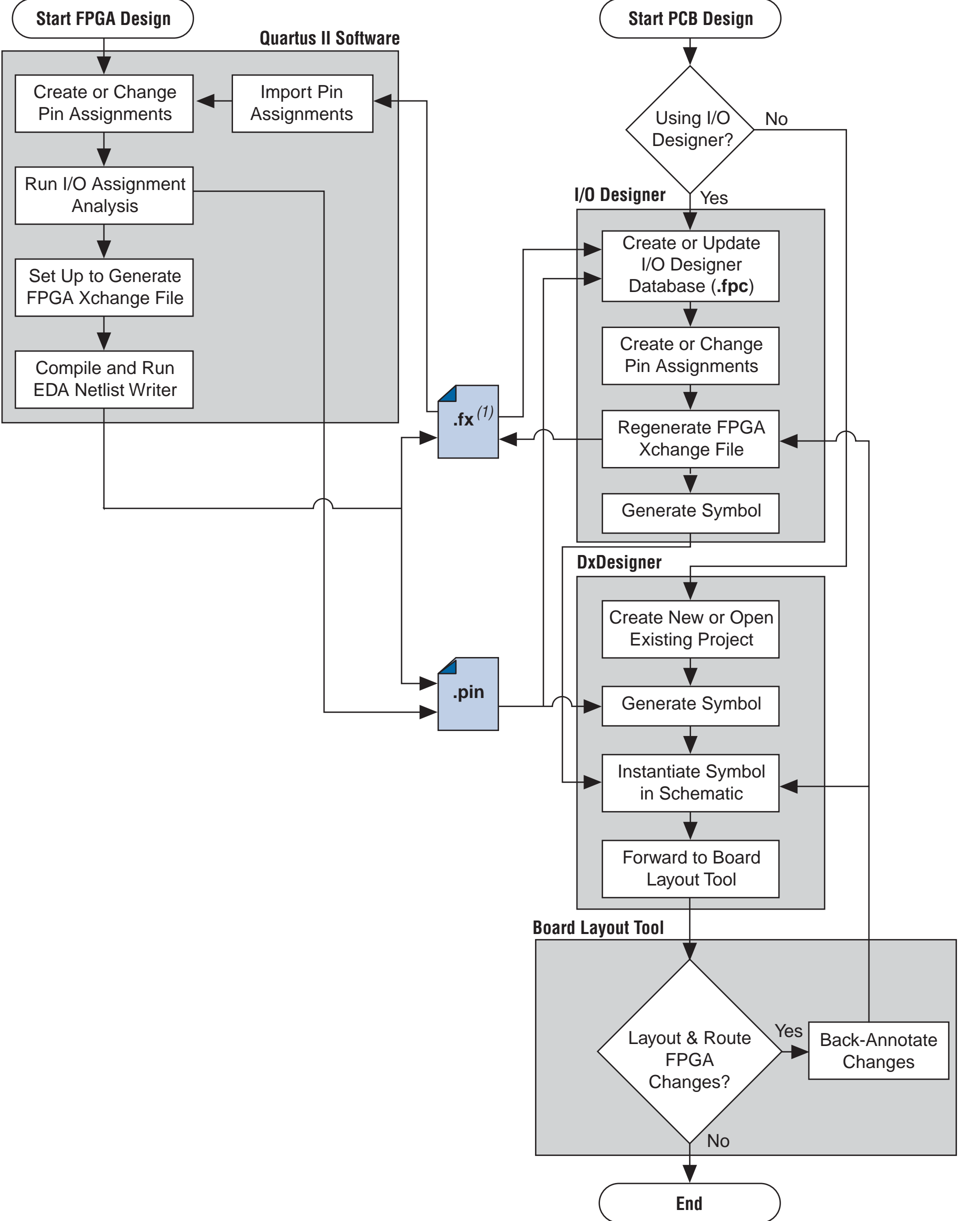
Mentor Graphics I/O Designer software is optional.

To obtain and license the Mentor Graphics tools and obtain product information, support, and training, go to the Mentor Graphics website at www.mentor.com.

FPGA-to-PCB Design Flow

In the examples in this section, you create a design flow integrating an Altera® FPGA design from the Quartus II software, and a circuit schematic in the DxDesigner software. Figure 8–1 shows the design flow with and without the I/O Designer software.

Figure 8–1. Design Flow with and without the I/O Designer Software



Note to Figure 8–1:

(1) The Quartus II software generates the FPGA Xchange file in the output directory you specify in the Board-Level Assignment Settings. However, the Quartus II software and the I/O Designer software can import pin assignments from an FPGA Xchange file located in any directory. Altera recommends that you work with a backup of the FPGA Xchange file to prevent overwriting existing assignments or importing invalid assignments.

The following tasks, which are described in this chapter, describe how to proceed through the design flow shown in Figure 8–1:

- Set up the board-level assignment settings to generate an FPGA Xchange file (**.fx**) for symbol generation in the Quartus II software
- Compile the design and generate the FPGA Xchange file and the Pin-Out file (**.pin**), which are located in the Quartus II project directory
- Create a board design using the DxDesigner software together with the I/O Designer software, which involves the following steps:
  - Create a new I/O Designer database based on the FPGA Xchange file and the Pin-Out file
  - Make adjustments to signal and pin assignments in the I/O Designer software
  - Regenerate the FPGA Xchange file in the I/O Designer software to reflect the I/O Designer software changes in the Quartus II software
  - Generate a single or fractured symbol for use in the DxDesigner software
  - Add the symbol to the **sym** directory of a DxDesigner project, or specify a new DxDesigner project with the new symbol
  - Instantiate the symbol in your DxDesigner schematic and export the design to the board layout tool
  - Back-annotate pin changes created in the board layout tool to the DxDesigner software and back to the I/O Designer software and the Quartus II software
- Create a board design using the DxDesigner software without the I/O Designer software, which involves the following steps:
  - Create a new DxBoardLink symbol using the Symbol Wizard and reference the Pin-Out file output from the Quartus II software in an existing DxDesigner project
  - Instantiate the symbol in your DxDesigner schematic and pass the design to a board layout tool

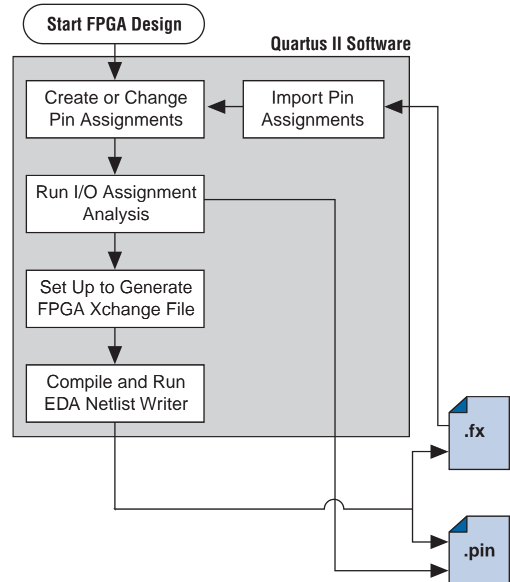
The I/O Designer software allows you to take advantage of the full FPGA symbol design, creation, editing, and back-annotation flow supported by Mentor Graphics tools.

Symbols can be updated with design changes at any point with or without the I/O Designer software. However, if symbols are changed in the DxDesigner software, the I/O Designer software does not see the changes. If you change symbols using the DxDesigner software, you must reimport the symbols into I/O Designer to avoid overwriting your symbol changes.

Setting Up the Quartus II Software

You can transfer pin and signal assignments from the Quartus II software to the Mentor Graphics tools by generating two output files, a Pin-Out file (**.pin**) and an FPGA Xchange file (**.fx**) (Figure 8–2).

Figure 8–2. Pin-Out Files and FPGA Xchange Files *(Note 1)*



Note to Figure 8–2:

(1) Refer to Figure 8–1 for the full design flow, which includes the I/O Designer software, the DxDesigner software, and the board layout tool flowchart details.

The two output files, the Pin-Out file and the FPGA Xchange file, are described in Table 8–1.

Table 8–1. Pin Assignment Output File Format Comparison

File Format	Description
Pin-Out file (**.pin**) *(1)*	An output file generated by the Quartus II Fitter. The file cannot be imported into the Quartus II software to change pin assignments. The file contains a complete list of the device pins including any unused I/O pins, and provides the following basic information fields for each assigned pin on a device: ■ Pin signal name/usage ■ Pin number ■ Signal direction ■ I/O standard ■ Voltage ■ I/O Bank ■ User or Fitter assigned
FPGA Xchange file (**.fx**) *(1)*,*(2)*	An input/output file generated by the Quartus II software and the I/O Designer software that can be imported and exported from both programs. Industry standard with room for future changes and additions. The FPGA Xchange file generated by the Quartus II software lists only assigned pins. The file provides the following advanced information fields for each pin on a device: ■ Pin number ■ I/O Bank ■ Signal name ■ Signal direction ■ I/O standard ■ Drive strength (mA) ■ Termination enabling ■ Slew rate ■ IOB Delay ■ Swap group ■ Differential pair type
	When generated by the I/O Designer software, all pins, including unused pins, are listed and the following fields are added: ■ Swap group ■ Differential pair type ■ Device pin name ■ Pin set ■ Pin set position ■ Pin set group ■ Super pin set group ■ Super pin set position

Notes to Table 8–1:

(1) For additional information about these file formats, refer to the Quartus II Help.

(2) For additional information about the information fields added by the Mentor Graphics software, refer to the Mentor Graphics website at www.mentor.com.

The I/O Designer software can also read from or update a Quartus II Settings File (**.qsf**). The Quartus II Settings File is used in the design flow in a similar manner to the FPGA Xchange file, but does not transfer pin swap group information between the I/O Designer software and the Quartus II software.

The **Quartus II Settings File** also contains additional important information about your project that is not used by the I/O Designer software. Because of this, Altera recommends that you use the FPGA Xchange file instead of the Quartus II Settings File for this design flow.

For more information about the Quartus II Settings File, refer to the *Quartus II Settings File Reference Manual*.

Generating Pin-Out Files

The Quartus II Fitter generates the Pin-Out file whenever you perform a full compilation or I/O Assignment Analysis on your design. The file is generated and placed in your design directory and your file is named <*project name*>**.pin**. The Mentor Graphics tools do not alter this file. The Quartus II software cannot import assignments from an existing Pin-Out file.

Generating FPGA Xchange Files

The FPGA Xchange file is not created automatically. To set up the Quartus II software to create the FPGA Xchange file, follow these steps:

1. Start the Quartus II software. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. Under **EDA Tool Settings**, click **Board-Level**. In the **Board-Level Symbol Format** list, choose **FPGA Xchange**.
3. Set the Output directory to the location where you want to save the file. The default output file path is <*project directory*>/symbols/fpgaxchange. Click **OK**.
4. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer**.

The output directory you selected is created when you generate the FPGA Xchange file.

Both the Quartus II software and the I/O Designer software can export and import an FPGA Xchange file. It is therefore possible to overwrite the FPGA Xchange file and import incorrect assignments into one or both programs. To prevent this occurrence from happening, make a backup copy of the file before importing, and import the copy instead of the file generated by the Quartus II software. In addition, assignments in the Quartus II software can be protected by following the steps in "Protecting Assignments in the Quartus II Software" on page 8–18.

Creating a Backup Quartus II Settings File

To create a backup Quartus II Settings File, perform the following steps:

1. On the Assignments menu, click **Import Assignments**. The **Import Assignments** dialog box appears.

2. In the **Import Assignments** dialog box, browse to your project and turn on **Copy existing assignments into** <*project name*>**.qsf.bak**.
3. Click **OK**.

Following these steps automatically creates a backup Quartus II Settings File of your current pin assignments.

For more information about pin and signal assignment transfer, and files the Quartus II software can import and export, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

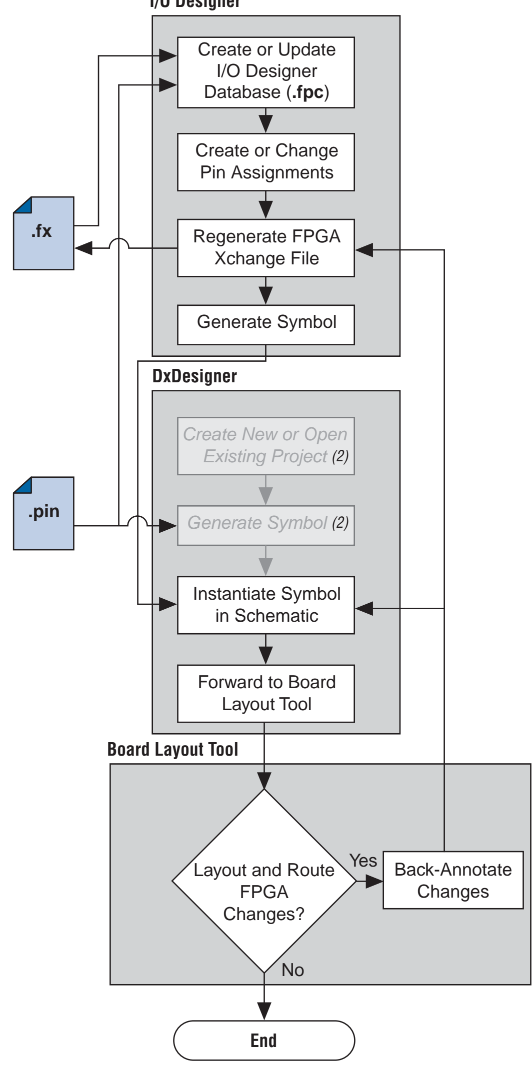
FPGA-to-Board Integration with the I/O Designer Software

The Mentor Graphics I/O Designer software allows you to integrate your FPGA and PCB designs. Pin and signal assignment changes can be made anywhere in the design flow, typically using either the Quartus II Pin Planner or the I/O Designer software. The I/O Designer software facilitates moving these changes, as well as synthesis, placement, and routing changes, between the Quartus II software, an external synthesis tool (if used), and a schematic capture tool such as the DxDesigner software.

This section describes how to use the I/O Designer software to transfer pin and signal assignment information to and from the Quartus II software with the FPGA Xchange file, and how to create symbols for the DxDesigner software.

Figure 8–3 shows the design flow using the I/O Designer software.

Figure 8–3. Design Flow Using the I/O Designer Software *(Note 1)*



Notes to Figure 8–3:

(1) Refer to Figure 8–1 for the full design flow including the Quartus II software flowchart details.

(2) These are DxDesigner software-specific steps in the design flow and are not part of the I/O Designer flow.

For more information about the I/O Designer software, and to obtain usage, support, and product updates, use the Help menu in the I/O Designer software or refer to the Mentor Graphics website at www.mentor.com.

I/O Designer Database Wizard

All I/O Designer project information is stored in an I/O Designer Database (**.fpc**) file. You can create a new database that incorporates the FPGA Xchange file and Pin-Out file information generated by the Quartus II software by using the I/O Designer Database Wizard. You can also create a new, empty database and manually add the assignment information. If there is no signal or pin assignment information currently available, you can create an empty database that contains only a selection of the target device. This is useful if you know the signals in your design and the pins you want to assign. You can transfer this information at a later time to the Quartus II software for place-and-route.

It is possible to create an I/O Designer database with only one type of file or the other. However, if only a Pin-Out file is used, any I/O assignment changes made in the I/O Designer software cannot be imported back into the Quartus II software without first generating an FPGA Xchange file. If only an FPGA Xchange file is used to create the I/O Designer database, the database may not contain a complete picture of all of the I/O assignment information available. The FPGA Xchange file generated by the Quartus II software only lists pins with assigned signals. Since the Pin-Out file lists all device pins—whether signals are assigned to them or not—its use, along with the FPGA Xchange file, produces the most complete set of information for creating the I/O Designer Database.

To create a new I/O Designer database using the Database Wizard, perform the following steps:

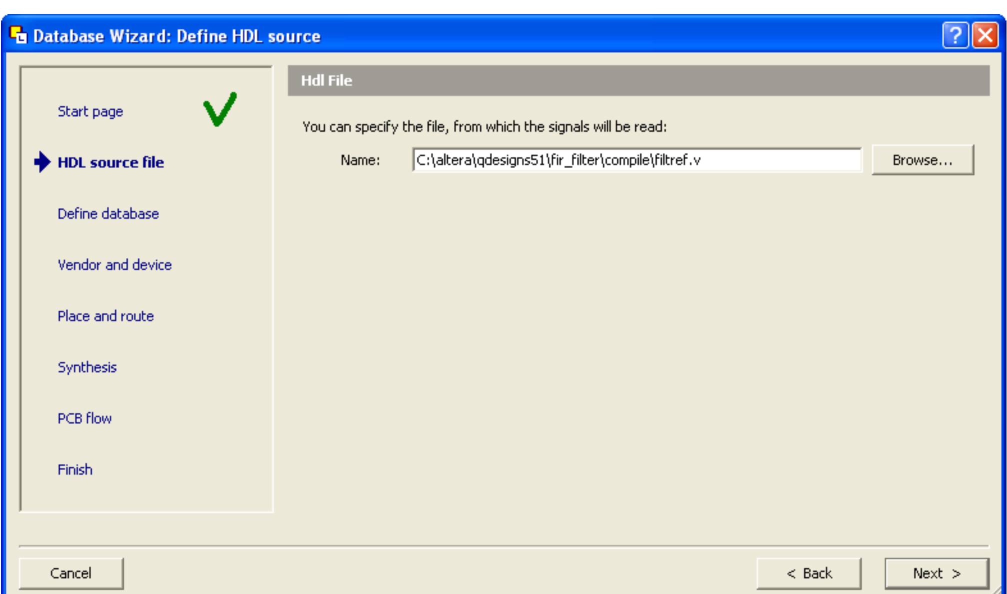
If you skip a step in this process, you can complete the skipped step later, filling in the appropriate information. To return to a skipped step, on the Properties menu, click **File**.

1. Start the I/O Designer software. The **Welcome to I/O Designer** dialog box appears. Select **Wizard to create new database** and click **OK**.

   If the **Welcome to I/O Designer** dialog box is not shown because it was disabled, you can access the Wizard through the menus. To access the Wizard, on the File menu, click **Database Wizard**.

2. Click **Next**. The **Define HDL source file** page opens (Figure 8–4).

Figure 8–4. Database Wizard HDL File Page



For more information about creating and using HDL files in the Quartus II software, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*, or refer to the I/O Designer Help.

If no HDL files are available, or if your signal and pin assignments are already contained in the FPGA Xchange file, you do not have to complete step 3 and can proceed to step 4.

3. If you have created a Verilog HDL or VHDL file in your Quartus II software design, you can enter a top-level Verilog HDL or VHDL file. Adding a file allows you to create functional blocks or get signal names from your design. All physical pin assignments must be created in I/O Designer if no FPGA Xchange file or Pin-Out file is used. Click **Next**. The Database Name page is shown.
4. In the Database Name window, enter your database file name. Click **Next**. The Database Location window is shown.
5. Enter a path to the new database or an existing one in the **Location** field, or browse to a database location. Click **Next**. The **FPGA flow** page is shown.
6. In the Vendor menu, click **Altera**.
7. In the Tool/Library menu, click **Quartus II 5.0**, or a later version of the Quartus II software.
8. Select the appropriate device family, device, package, and speed (if applicable), from the corresponding menus. Click **Next**. The **Place and route** page is shown.

The Quartus II software version selections in the Tool/Library menu may not reflect the version of the Quartus II software currently installed on your system even if you are using the most current version of the I/O Designer software. The version number selection in this window is used in the I/O Designer software to identify the devices that were available or obsolete in that particular version of the Quartus II software. If you are unsure of the version to select, use the most recent version listed in the menu. If the device you are targeting does not appear in the device menu after making this selection, the device may be new and not yet added to the I/O Designer software. For I/O Designer software updates, contact Mentor Graphics or refer to their website at www.mentor.com.

9. In the **FPGAX file name** field, type or browse to the backup copy of the FPGA Xchange file generated by the Quartus II software.
10. In the **Pin report file name** field, type or browse to the Pin-Out file generated by the Quartus II software. Click **Next**.

    In addition, you can select a Quartus II Settings File for update. The I/O Designer software can update the pin assignment information in the Quartus II Settings File without affecting any other information contained in the file.

You can select a Pin-Out file without selecting an FPGA Xchange file for import. The I/O Designer software does not generate a Pin-Out file. To transfer assignment information to the Quartus II software, select an additional file and file type. Altera recommends selecting an FPGA Xchange file in addition to a Pin-Out file for transferring all of the assignment information contained within both types of files.

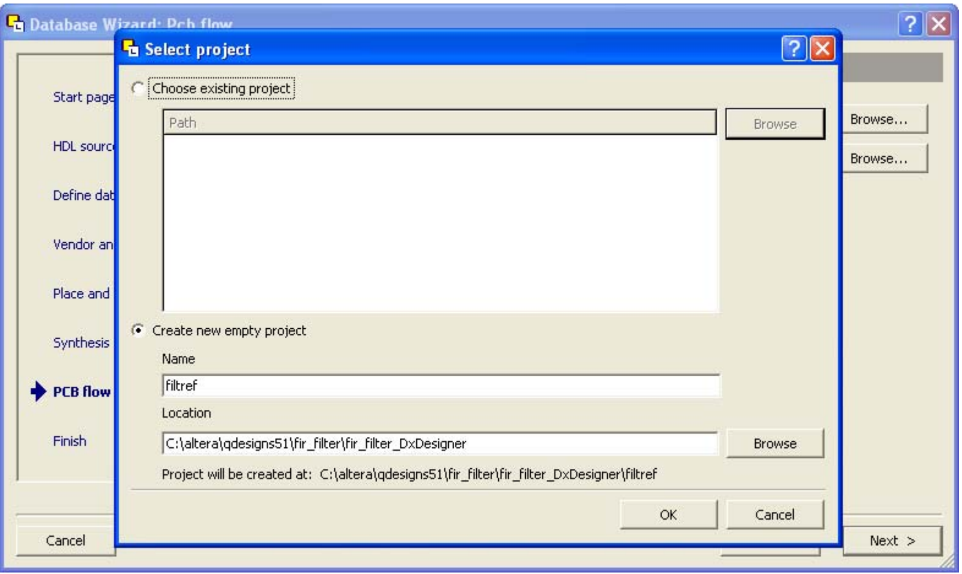
In some versions of the I/O Designer software, the standard file picker may incorrectly look for a Pin-Out file instead of an FPGA Xchange file. In this case, select **All Files (*.*)** from the **Save as type** list and select the file from the list.

11. The **Synthesis** page displays. On the **Synthesis** page, you can specify an external synthesis tool and a synthesis constraints file for use with the tool. If you do not use an external synthesis tool, click **Next**.

For more information about third-party synthesis tools, refer to *Volume 3: Verification* of the *Quartus II Handbook*.

12. On the **PCB Flow** page, you can select an existing schematic project or create a new project as a destination for symbol information.
    - To select an existing project, select **Choose existing project** and click **Browse** after the Project Path field. The **Select project** dialog box appears. Select the project.
    - To create a new project, in the **Select project** dialog box, select **Create new empty project**. Enter the project file name in the **Name** field and browse to the location where you want to save the file (Figure 8–5). Click **OK**.

Figure 8–5. Select Project Dialog Box

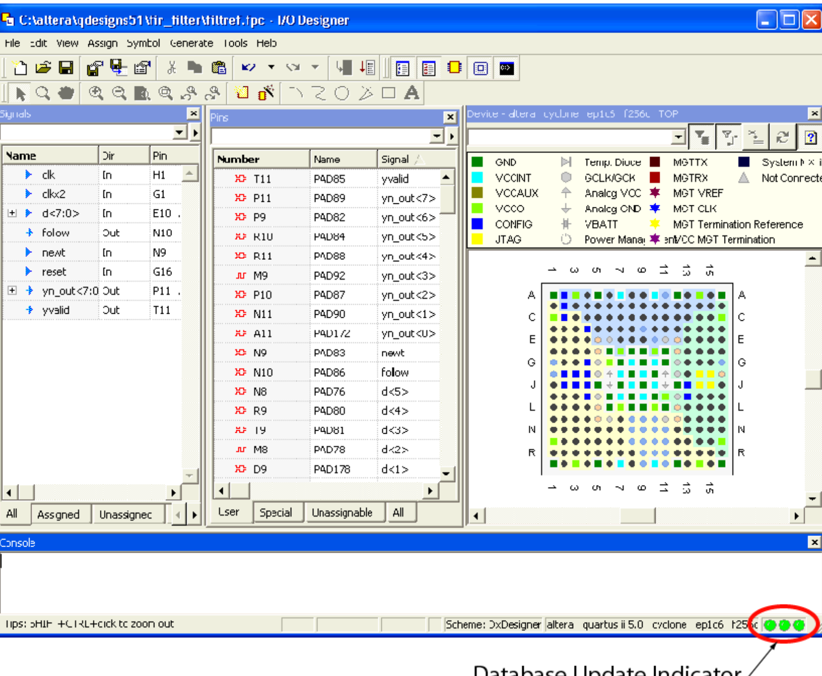


If you have not specified a design tool to which you can send symbol information in I/O Designer, click **Advanced** in the **PCB Flow** page and select your design tool. If the DxDesigner software is selected, you have the option of specifying a Hierarchical Occurrence Attributes (**.oat**) file to import into the I/O Designer software. Click **Next**, then click **Finish** to create the database.

In I/O Designer version 2005 or later, the Update Wizard (refer to Figure 8–9 on page 8–16) is shown when you finish creating the database using the database wizard. Use the Update Wizard to confirm creation of the I/O Designer database using the selected FPGA Xchange and Pin-Out files.

Use the I/O Designer software and your newly created database to make pin assignment changes, create pin swap groups, or adjust signal and pin properties in the I/O Designer GUI (Figure 8–6).

Figure 8–6. I/O Designer Main Window



For more information about using the I/O Designer software and the DxDesigner software, refer to the Mentor Graphics website at www.mentor.com or refer to the I/O Designer software or the DxDesigner Help.

Updating Pin Assignments from the Quartus II Software

As the design process continues, the FPGA designer may need to make changes to the logic design in the Quartus II software that place signals on different pins after the design is recompiled, or manually by using the Quartus II Pin Planner. These types of changes must be carried forward to the circuit schematic and board layout tools to ensure that signals are connected to the correct pins on the FPGA. Updating the FPGA Xchange file and the Pin-Out file in the Quartus II software facilitates this flow (Figure 8–7).

Figure 8–7. Updating the I/O Designer Pin Assignments in the Design Flow *(Note 1)*

Note to Figure 8–7:

(1) Refer to Figure 8–1 for the full design flow, which includes the Quartus II software, the DxDesigner software, and the board layout tool flowchart details.

To update the FPGA Xchange file and the Pin-Out file in the Quartus II software after making changes to the design, run a full compilation, or on the Start menu, point to Processing and click **Start EDA Netlist Writer**. The FPGA Xchange file in your selected output directory and the Pin-Out file in your project directory are updated. You must rerun the I/O Assignment Analyzer whenever you make I/O changes in the Quartus II software. To rerun the I/O Assignment Analyzer, on the Processing menu, click **Start Compilation**, or to run a full compilation, on the Processing menu, point to **Start** and click **Start I/O Assignment Analysis**.

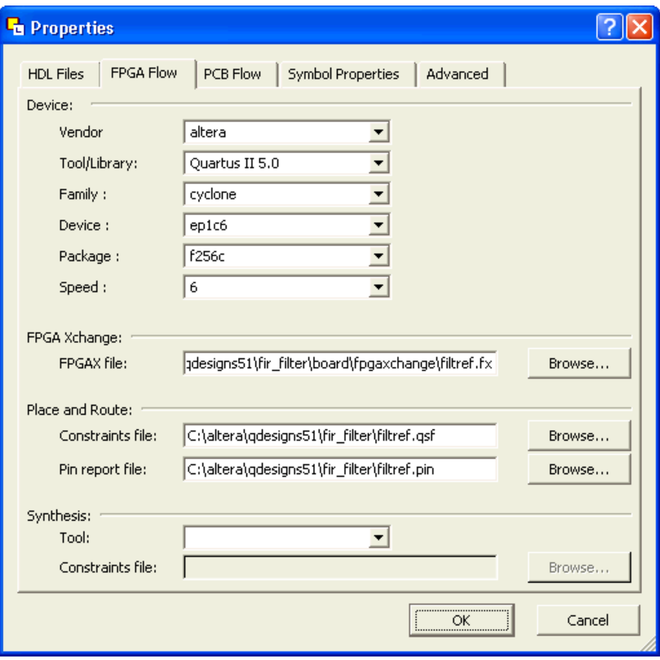
Refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook* for more information about setting up the FPGA Xchange file and running the I/O Assignment Analyzer.

If your I/O Designer database points to the FPGA Xchange file generated by the Quartus II software instead of a backup copy of the file, updating the file in the Quartus II software overwrites any changes made to the file by the I/O Designer software. If there are I/O Designer assignments in the FPGA Xchange file that you want to preserve, create a backup copy of the file before updating it in the Quartus II software, and verify that your I/O Designer database points to the backup copy. To point to the backup copy, perform the steps in the following section.

Whenever the FPGA Xchange file or the Pin-Out file is updated in the Quartus II software, the changes can be imported into the I/O Designer database. You must set up the locations for the files in the I/O Designer software.

1. To set up the file locations if they are not already set, on the File menu, click **Properties**. The project **Properties** dialog box appears (Figure 8–8).

Figure 8–8. Project Properties Dialog Box



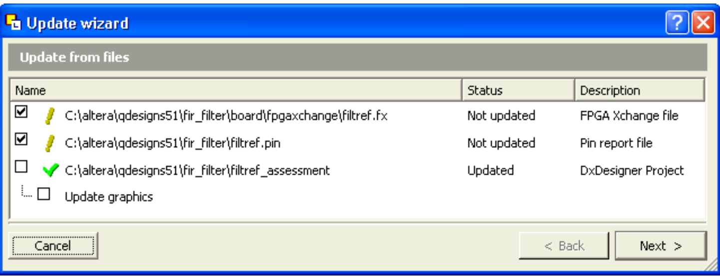
2. Under **FPGA Xchange**, click **Browse** to select the FPGA Xchange file name and file location.
3. To specify a Pin report file, under **Place and Route**, click **Browse** to select the Pin-Out file name and file location.

After you have set up these file locations, the I/O Designer software monitors these files for changes. If the FPGA Xchange file or Pin-Out file changes during the design flow, three indicators flash red in the lower right-hand corner of the I/O Designer main window (see Figure 8–6 on page 8–13). You can continue working or click on the indicators to open the I/O Designer Update Wizard. If you have made changes to your design in the Quartus II software that result in an updated FPGA Xchange file or Pin-Out file and the update indicators do not flash or you have previously canceled an indicated update, manually open the Update Wizard. To open the Wizard, on the File menu, click **Update**.

In versions of the I/O Designer software before version 2005, instead of using flashing indicators, the I/O Designer software displays a dialog box asking if you want to open the Update Wizard.

The I/O Designer Update Wizard lists the updated files associated with the database (Figure 8–9).

Figure 8–9. Update Wizard Dialog Box



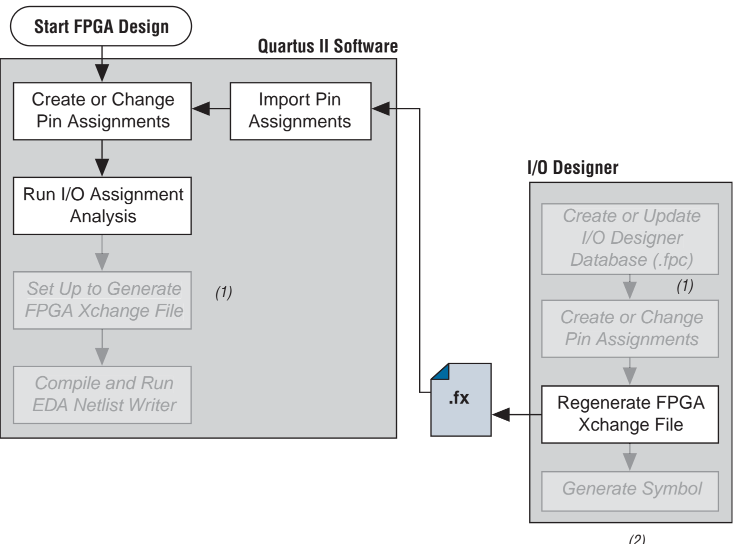
The paths to the updated files have yellow exclamation points and the **Status** column shows **Not updated**, indicating that the database has not yet been updated with the newer information contained in the files. A checkmark to the left of any updated file indicates that the file will update the database. Turn on any files you want to use to update the I/O Designer database, and click **Next**. If you are not satisfied with the database update, on the Edit menu, click **Undo**.

You can update the I/O Designer database using both the FPGA Xchange file and the Pin-Out file at the same time. Turning on both the FPGA Xchange file and the Pin-Out file for update causes the Update Wizard to provide options for using assignments from one file or the other exclusively or merging the assignments contained in both files into the I/O Designer database. Versions of the I/O Designer software older than version 2005 simply merge assignments contained in multiple files.

Sending Pin Assignment Changes to the Quartus II Software

In the same way that the FPGA designer can make adjustments that affect the PCB design, the board designer can make changes to optimize signal routing and layout that must be applied to the FPGA. The FPGA designer can take these required changes back into the Quartus II software to refit the logic to match the adjustments to the pinout. The I/O Designer software can accommodate this reverse flow as shown in Figure 8–10.

Figure 8–10. Updating the Quartus II Pin Assignments in the Reverse Design Flow



Notes to Figure 8–10:

(1) These are software-specific steps in the design flow and are not necessary for the reverse flow steps of the design.

(2) Refer to Figure 8–1 for the full design flow, which includes the complete I/O Designer software, the DxDesigner software, and the board layout tool flowchart details.

Pin assignment changes are made directly in the I/O Designer software, or the software automatically updates changes made in a board layout tool that are back-annotated to a schematic entry program such as the DxDesigner software. You must update the FPGA Xchange file to reflect these updates in the Quartus II software. To perform this update in the I/O Designer software, on the Generate menu, click **FPGA Xchange File**.

If your I/O Designer database points to the FPGA Xchange file generated by the Quartus II software instead of a backup copy, updating the file from the I/O Designer software overwrites any changes that may have been made to the file by the Quartus II software. If there are assignments from the Quartus II software in the file that you want to preserve, make a backup copy of the file before updating it in the I/O Designer software, and verify that your I/O Designer database points to the backup copy. To point to the backup copy, perform the steps in "Updating Pin Assignments from the Quartus II Software" on page 8–13.

After the FPGA Xchange file is updated, you must import it into the Quartus II software. To import the file, perform the following steps:

1. Start the Quartus II software and open your project.
2. On the Assignments menu, click **Import Assignments**.
3. In the File name box, click **Browse** and from the **Files of type** list, select **FPGA Xchange Files (*.fx)**.
4. Select the FPGA Xchange file and click **Open**.
5. Click **OK**.

Both the Quartus II software and the I/O Designer software can export and import an FPGA Xchange file. It is therefore possible to overwrite the FPGA Xchange file and import incorrect assignments into one or both programs. To prevent this occurrence from happening, make a backup copy of the file before importing, and import the copy instead of the file generated by the Quartus II software. In addition, assignments in the Quartus II software can be protected by following the steps in "Protecting Assignments in the Quartus II Software".

Protecting Assignments in the Quartus II Software

To protect assignments in the Quartus II software, perform the following steps:

1. Start the Quartus II software.
2. On the Assignments menu, click **Import Assignments**. The **Import Assignments** dialog box appears.
3. Turn on **Copy existing assignments into** <*project name*>**.qsf.bak before importing** before importing the FPGA Xchange file. This action automatically creates a backup Quartus II constraints file that contains all of your current pin assignments.

Generating Symbols for the DxDesigner Software

Along with circuit simulation, circuit board schematic creation is one of the first tasks required in the design of a new PCB. Schematics are required to understand how the PCB will work, and to generate a netlist that is passed to a board layout tool for board design and routing. The I/O Designer software provides the ability to create schematic symbols based on the FPGA design exported from the Quartus II software.

Most FPGA devices contain hundreds of pins, requiring large schematic symbols that may not fit on a single schematic page. Symbol designs in the I/O Designer software can be split or fractured into a number of functional blocks, allowing multiple part fractures on the same schematic page or across multiple pages. In the DxDesigner software, these part fractures are joined together with the use of the `HETERO` attribute.

The I/O Designer software can generate symbols for use in a number of Mentor Graphics schematic entry tools, and can import changes back-annotated by board layout tools to update the database and feed updates back to the Quartus II software using the FPGA Xchange file. This section discusses symbol creation specifically for the DxDesigner software.

Schematic symbols are created in the I/O Designer software in the following ways:

- Manually
- Using the I/O Symbol Wizard
- Importing previously created symbols from the DxDesigner software

The I/O Designer Symbol Wizard can be used as a design base that allows you to quickly create a symbol for manual editing at a later time. If you have already created symbols in a DxDesigner project and want to apply a different FPGA design to them, you can manually import these symbols from the DxDesigner project. To import the symbols, open the I/O Designer software, and on the File menu, click **Import Symbol**.

For more information about importing symbols from the DxDesigner software into an I/O Designer database, refer to the I/O Designer Help.

Symbols created in the I/O Designer software are either functional, physical (PCB), or a combination of functional and physical. A functional symbol is based on signals imported into the database, usually from Verilog HDL or VHDL files. No physical device pins must be associated with the signals to generate a functional symbol. This section focuses on board-level PCB symbols with signals directly mapped to physical device pins through assignments in either the Quartus II Pin Planner or in the I/O Designer database.

For information about manually creating symbols, importing symbols, and editing symbols in the I/O Designer software, as well as the different types of symbols the software can generate, refer to the I/O Designer Help.

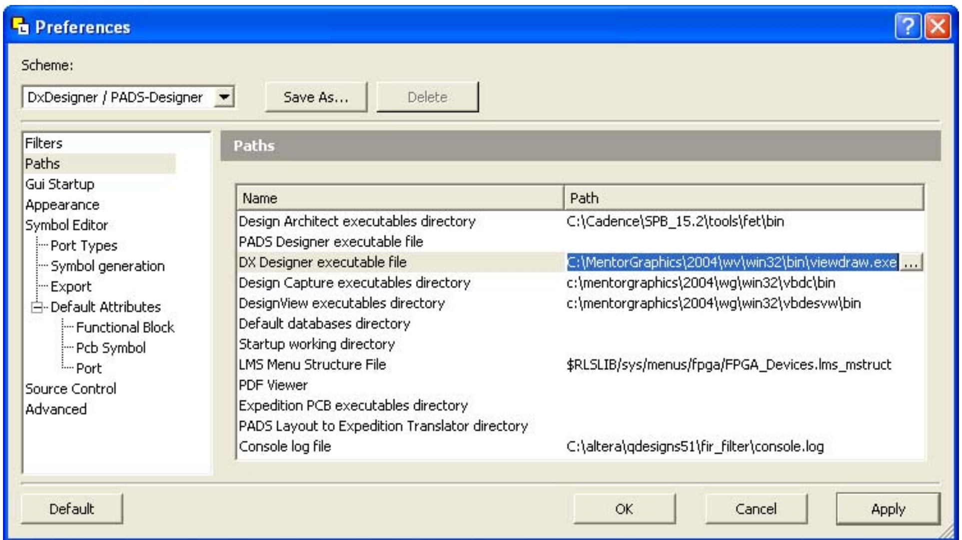
Setting Up the I/O Designer Software to Work with the DxDesigner Software

If you created your I/O Designer database using the Database Wizard, you may already be set up to export symbols to a DxDesigner project. To verify this, or to manually set up the I/O Designer software to work with the DxDesigner software, you must set the path to the DxDesigner executable, set the export type to DxDesigner, and set the path to a DxDesigner project directory.

To set these options, perform the following steps:

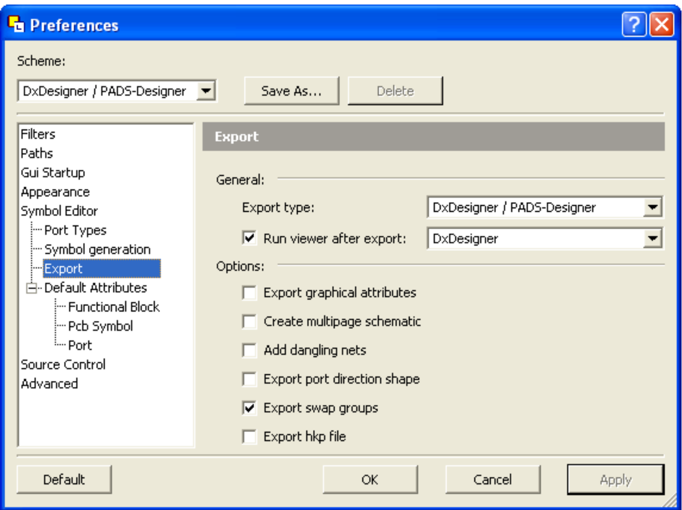
1. Start the I/O Designer software.
2. On the Tools menu, click **Preferences**. The **Preferences** dialog box appears.
3. Click **Paths**, double-click on the DxDesigner executable file path field, and click **Browse** to select the location of the DxDesigner application (Figure 8–11).
4. Click **Apply**.

Figure 8–11. Path Preferences Dialog Box



5. Click **Symbol Editor** and click **Export**. In the Export type menu, under **General**, select **DxDesigner/PADS-Designer** (Figure 8–12).
6. Click **Apply** and click **OK**.

Figure 8–12. Symbol Editor Export Preferences



7. On the File menu, click **Properties**. The project **Properties** dialog box appears.
8. Click the **PCB Flow** tab and click **Path to a DxDesigner project directory**.
9. Click **OK**.

If you did not create a new DxDesigner project in the Database Wizard and you do not already have a DxDesigner project, you must create a new database using the DxDesigner software, and point the I/O Designer software to this new project.

For information about creating and working with DxDesigner projects, refer to the DxDesigner Help.

Create Symbols with the Symbol Wizard

FPGA symbols based on Altera devices can be created, fractured, and edited using the I/O Designer Symbol Wizard. To create a symbol based on a selected Altera FPGA device:

1. Start the I/O Designer software.
2. Click **Symbol Wizard** in the toolbar, or on the Symbol menu, click **Symbol Wizard**. The **Symbol Wizard (1 of 6)** page is shown (Figure 8–13).

Figure 8–13. Symbol Wizard

3. On the first Symbol Wizard page, in the **Symbol name** field, enter the symbol name. The **DEVICE** and **PKG_TYPE** fields are populated with the device and package information automatically. Under **Symbol type**, click **PCB**. Under **Use signals**, click **All**.
4. Click **Next**. The **Symbol Wizard (2 of 6)** page is shown.

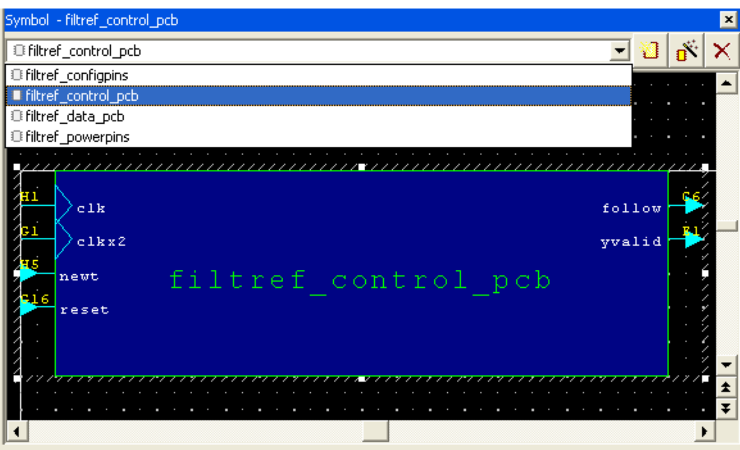
If the **DEVICE** and **PKG_TYPE** fields are blank or incorrect, cancel the Symbol Wizard and select the correct device information. On the File menu, click **Properties**. In the Properties window, click the **FPGA Flow** tab and enter the correct device information.

5. On page 2 of the Symbol Wizard, select fracturing options for your symbol. If you are using the Symbol Wizard to edit a previously created fractured symbol, you must turn on **Reuse existing fractures** so that your current fractures are not altered. Select other options on this page as appropriate for your symbol.
6. Click **Next**. The **Symbol Wizard (3 of 6)** page is shown.
7. Additional fracturing options are available on page 3 of the Symbol Wizard. After selecting the desired options, click **Next**. The **Symbol Wizard (4 of 6)** page is shown.
8. On page 4 of the Symbol Wizard, select the options for how the symbols will look. Select the desired options and click **Next**. The **Symbol Wizard (5 of 6)** page is shown.
9. On page 5 of the Symbol Wizard, define what information will be labeled for the entire symbol and for individual pins. Select the desired options and click **Next**. The **Symbol Wizard (6 of 6)** page is shown.

10. On the final page of the Symbol Wizard, add additional signals and pins that have not already been placed in the symbol. Click **Finish** when you complete your selections.

Your symbol is complete. You can view your symbol and any fractures you created using the Symbol Editor (Figure 8–14). You can edit parts of the symbol, delete fractures, or rerun the Symbol Wizard.

Figure 8–14. The I/O Designer Symbol Editor



If assignments in the I/O Designer database are updated, the symbols created in the I/O Designer software automatically reflect these changes. Assignment changes can be made within the I/O Designer software, with an updated FPGA Xchange file from the Quartus II software, or from a back-annotated change in your board layout tool.

Export Symbols to the DxDesigner Software

After you have completed your symbols, export the symbols to your DxDesigner project. To generate all the fractures of a symbol, on the Generate menu, click **All Symbols**. To generate a symbol for the currently displayed symbol in Symbol Editor, click **Current Symbol Only**. Each symbol in the database is saved as a separate file in the /sym directory in your DxDesigner project. The symbols can be instantiated in your DxDesigner schematics.

For more information about working with DxDesigner projects, refer to the DxDesigner Help.

Scripting Support

The I/O Designer software features a command line Tcl interpreter. All commands issued through the GUI in the I/O Designer software are translated into Tcl commands that are run by the tool. You can view the generated Tcl commands and run scripts, or enter individual commands in the I/O Designer Console window.

The following section includes commands that perform some of the operations described in this chapter.

If you want to change the FPGA Xchange file from which the I/O Designer software updates assignments, type the following command at an I/O Designer Tcl prompt:

```
set_fpga_xchange_file <file name>
```

After the FPGA Xchange file is specified, use the following command to update the I/O Designer database with assignment updates made in the Quartus II software:

```
update_from_fpga_xchange_file
```

Use the following command to update the FPGA Xchange file with changes made to the assignments in the I/O Designer software for transfer back into the Quartus II software:

```
generate_fpga_xchange_file
```

If you want to import assignment data from a Pin-Out file created by the Quartus II software, use the following command:

```
set_pin_report_file -quartus_pin <file name>
```

Run the I/O Designer Symbol Wizard with the following command:

```
symbolwizard
```

Set the DxDesigner project directory path where symbols are saved with the following command:

```
set_dx_designer_project -path <path>
```

For more information about Tcl scripting and Tcl scripting with the Quartus II software, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about the Tcl scripting capabilities of the I/O Designer software as well as a list of all the commands available, refer to the I/O Designer Help.

FPGA-to-Board Integration with the DxDesigner Software

The Mentor Graphics DxDesigner software is a design entry tool for schematic capture. You can use it to create flat circuit schematics for all types of PCB design. You can also use the DxDesigner software to create hierarchical schematics that facilitate design reuse and a team-based design. You can use the DxDesigner software in the design flow alone or in conjunction with the I/O Designer software. However, if you use the DxDesigner software without the I/O Designer software, the design flow is one-way, using only the Pin-Out file generated by the Quartus II software.

Signal and pin assignment changes can be made only in the Quartus II software and are reflected in updated symbols in a DxDesigner schematic. You cannot back-annotate changes made in a board layout tool or in a DxDesigner symbol to the Quartus II software. Figure 8–15 shows the design flow when the I/O Designer software is not used.

Figure 8–15. Design Flow Without the I/O Designer Software *(Note 1)*

DxDesigner
.pin
Create New or Open Existing Project
Generate Symbol
Instantiate in Schematic
Forward to Board Layout Tool

Note to Figure 8–15:

(1) Refer to Figure 8–1 for the full design flow, which includes the Quartus II software, the I/O Designer software, and the board layout tool flowchart details.

For more information about the DxDesigner software, including usage, support, training, and product updates, refer to the Mentor Graphics web page at www.mentor.com, or choose Schematic Design Help Topics in the DxDesigner Help.

DxDesigner Project Settings

New projects in the DxDesigner software are already set up to create FPGA symbols by default. However, for complete support and compatibility with the I/O Designer software, if it is used with the DxDesigner software, you should enable the DxBoardLink Flow options.

You can enable the DxBoardLink flow design configuration while creating a new DxDesigner project or after a project is created.

To enable the DxBoardLink flow design configuration when creating a new DxDesigner project, perform the following steps:

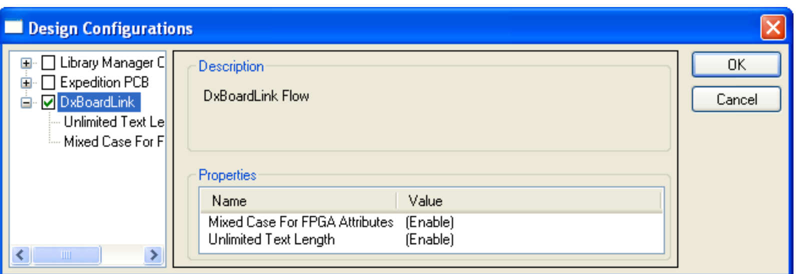
1. Start the DxDesigner software.
2. On the File menu, click **New** and click the **Project** tab. The **New** dialog box appears (Figure 8–16).

Figure 8–16. New Project Dialog Box

3. Click **More**. Turn on **DxBoardLink** (Figure 8–16).

To enable the DxBoardLink Flow design configuration in an existing project, click **Design Configurations** in the Design Configuration toolbar and turn on **DxBoardLink** (Figure 8–17).

Figure 8–17. DxBoardLink Design Configuration



DxDesigner Symbol Wizard

In addition to circuit simulation, circuit board schematic creation is one of the first tasks required in the design of a new PCB. Schematics are required to understand how the PCB will work, and to generate a netlist that is passed on to a board layout tool for board stackup design and routing.

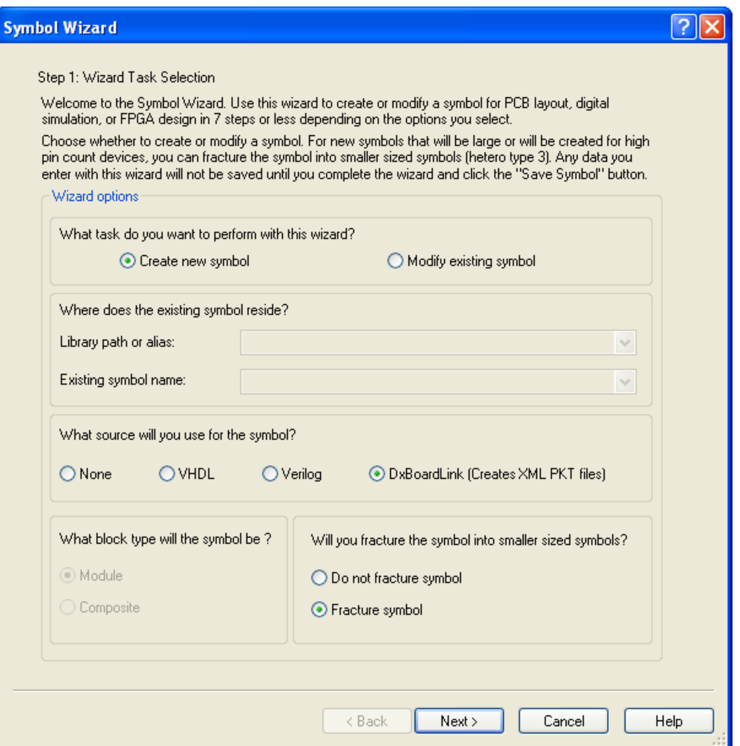
You can create schematic symbols using the DxDesigner software based on FPGA designs exported from the Quartus II software through the Pin-Out file for instantiation in DxDesigner schematic design files. Most FPGA devices are physically large with hundreds of pins, requiring large schematic symbols that may not fit on a single schematic page. You can split or fracture symbols created in the DxDesigner software into a number of functional blocks, allowing multiple part fractures on the same schematic page or across multiple pages. In the DxDesigner software, these part fractures are joined together with the use of the `HETERO` attribute.

You can create schematic symbols in the DxDesigner software manually or with the Symbol Wizard. The DxDesigner Symbol Wizard is similar to the I/O Designer Symbol Wizard, but with fewer fracturing options.

FPGA symbols based on Altera devices can be created, fractured, and edited using the DxDesigner Symbol Wizard. To start the Symbol Wizard, perform the following steps:

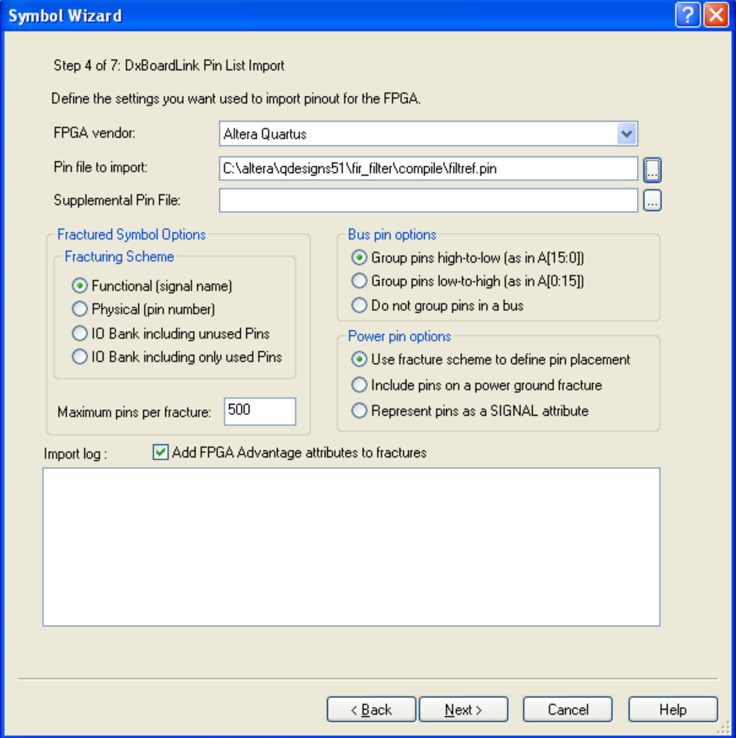
1. Start the DxDesigner software.
2. Click **Symbol Wizard** in the toolbar, or on the File menu, click **New**. The **New** window is shown. Click the **File** tab and create a new file of type **Symbol Wizard**.
3. Enter the new symbol name in the name field and click **OK**. The **Symbol Wizard** page is shown (Figure 8–18).

Figure 8–18. Wizard Task Selection



4. On the **Wizard Task Selection** page, choose to create a new symbol or modify an existing symbol. If you are modifying an existing symbol, specify the library path or alias, and select the existing symbol. If you are creating a new symbol, select DxBoardLink for the symbol source. The DxDesigner block type defaults to Module because the FPGA design does not have an underlying DxDesigner schematic. Define whether or not to fracture the symbol. After making your selections, click **Next**. The **New Symbol and Library Name** page is shown.
5. On the **New Symbol and Library Name** page, enter a name for the symbol, an overall part name for all of the symbol fractures, and a library name for the new library created for this symbol. By default, the part and library names are the same as the symbol name. Click **Next**. The **Symbol Parameters** page is shown.
6. On the **Symbol Parameters** page, decide how the generated symbol will look and how it will match up with the grid you have set in your DxDesigner project schematic. After making your selections, click **Next**. The **DxBoardLink Pin List Import** page is shown (Figure 8–19).

Figure 8–19. DxBoardLink Pin List Import



7. On the **DxBoardLink Pin List Import** page, in the **FPGA vendor** list, select **Altera Quartus**. In the Pin-Out file to import field, browse to and select the Pin-Out file from your Quartus II design project directory. Additionally, select choices from the Fracturing Scheme options, Bus pin options, and Power pin options. After you make your selections, click **Next**. The **Symbol Attributes** page is shown.
8. On the **Symbol Attributes** page, select to create or modify symbol attributes for use in the DxDesigner software. After you make your selections, click **Next**. The **Pin Settings** page is shown.

9. On the **Pin Settings** page, make any final adjustments to pin and label location and information. Each tabbed spreadsheet represents a fracture of your symbol. After you make your selections, click **Save Symbol**.

After you save the symbol, you can examine and place any fracture of the symbol in your schematic. When you are finished with the Symbol Wizard, all the fractures you created are saved as separate files in the library you specified or created in the **/sym** directory in your DxDesigner project. You can add the symbols to your schematics or you can edit the symbols manually or with the Symbol Wizard.

Symbols created in the DxDesigner software can be edited and updated with newer versions of the Pin-Out file generated by the Quartus II software. However, symbol fracturing is fixed, and the symbol cannot be fractured again. To create new fractures for your design, create a new symbol in the Symbol Wizard, and follow the steps in "DxDesigner Symbol Wizard" on page 8–25.

For more information about creating, editing, and instantiating component symbols in DxDesigner, choose Schematic Design Help Topics from the Help menu in the DxDesigner software.

Conclusion

Transferring a complex, high-pin-count FPGA design to a PCB for prototyping or manufacturing is a daunting process that can lead to errors in the PCB netlist or design, especially when multiple engineers are working on different parts of the project. The design workflow available when the Quartus II software is used in conjunction with the Mentor Graphics toolset assists the FPGA designer and the board designer in preventing errors and focusing their attention on the design.

Referenced Documents

This chapter references the following documents:

- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Volume 3: Verification* of the *Quartus II Handbook*

 © March 2009 Altera Corporation

Document Revision History

Table 8–2 shows the revision history for this chapter.

Table 8–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Was chapter 6 in the 8.1.0 release. ■ Removed Figures that were numbered 6-4, 6-6, 6-7, and 6-8 in v8.1.0.	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	Changed to 8½” × 11” page size. No change to content.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	Updated references.	Updated for the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

9. Cadence PCB Design Tools Support

QII52014-9.0.0

Introduction

With today's large, high-pin-count and high-speed FPGA devices, good PCB design practices are more essential than ever to ensure the correct operation of your system. Typically, the PCB design takes place concurrently with the design and programming of the FPGA. Signal and pin assignments are initially made by the FPGA or ASIC designer, and it is up to the board designer to correctly transfer these assignments to the symbols used in their system circuit schematics and board layout. As the board design progresses, pin reassignments might be requested or required to optimize the layout. These reassignments must in turn be relayed to the FPGA designer so that the new assignments can be processed through the FPGA using updated place-and-route.

Cadence provides tools to support this type of design flow. This chapter addresses how the Quartus® II software interacts with the Cadence Allegro Design Entry HDL software and the Allegro Design Entry CIS (Component Information System) software (also known as OrCAD Capture CIS) to provide a complete FPGA-to-board integration design workflow. This chapter provides information about the following topics:

- Cadence tool description, history, and comparison
- The general design flow between the Quartus II software and the Cadence Allegro Design Entry HDL software and the Cadence Allegro Design Entry CIS software
- Generating schematic symbols from your FPGA design for use in the Cadence Allegro Design Entry HDL software
- Updating Design Entry HDL symbols when signal and pin assignment changes are made in the Quartus II software
- Creating schematic symbols in the Cadence Allegro Design Entry CIS software from your FPGA design
- Updating symbols in the Cadence Allegro Design Entry CIS software when signal and pin assignment changes are made in the Quartus II software
- Using Altera®-provided device libraries in the Cadence Allegro Design Entry CIS software

This chapter is intended primarily for board design and layout engineers who want to begin the FPGA board integration process while the FPGA is still in the design phase. In addition, part librarians benefit from learning how to take output from the Quartus II software and use it to create new library parts and symbols.

The instructions in this chapter require the following software:

- The Quartus II software version 5.1 or later
- The Cadence Allegro Design Entry HDL or the Cadence Allegro Design Entry CIS software version 15.2 or later
- If you are using the OrCAD Capture software, you must have version 10.3 or later (CIS is optional)

Because the Cadence Allegro Design Entry CIS software is based on OrCAD Capture, these programs are very similar. For this reason, this chapter refers to the Allegro Design Entry CIS software in directions; however, these directions also apply to OrCAD Capture unless otherwise noted.

To obtain and license the Cadence tools described in this chapter, and for product information, support, and training, refer to the Cadence website, www.cadence.com. For information about OrCAD Capture and the CIS option, refer to the Cadence website. For Cadence and OrCAD support and training, refer to the EMA Design Automation website, www.ema-eda.com.

Product Comparison

The design tools described in this chapter have similar functionality, but there are differences both in their use and where to access product information. Table 9–1 lists the products described in this chapter and provides information about changes, product information, and support.

Table 9–1. Cadence and OrCAD Product Comparison

	Cadence Allegro Design Entry HDL	Cadence Allegro Design Entry CIS	OrCAD Capture CIS
Former Name	Concept HDL Expert	Capture CIS Studio	—
History	More commonly known by its former name, Cadence renamed all board design tools in 2004 under the Allegro name.	Based directly on OrCAD Capture CIS, this tool is still developed by OrCAD but sold and marketed by Cadence. EMA provides support and training.	The basis for Design Entry CIS is still developed by OrCAD for continued use by existing OrCAD customers. EMA now provides support and training for all OrCAD products.
Vendor Design Flow	Cadence Allegro 600 series, formerly known as Expert Series, for high-end, high-speed design.	Cadence Allegro 200 series, formerly known as Studio Series, for small- to medium-level design.	—
Information and Support	www.cadence.com www.ema-eda.com	www.cadence.com www.ema-eda.com	www.cadence.com www.ema-eda.com

FPGA-to-PCB Design Flow

In the examples in this section, you create a design flow integrating an Altera FPGA design from the Quartus II software through a circuit schematic in the Allegro Design Entry HDL software (Figure 9–1) or the Allegro Design Entry CIS software (Figure 9–2).

Figure 9–1. Design Flow with the Allegro Design Entry HDL Software

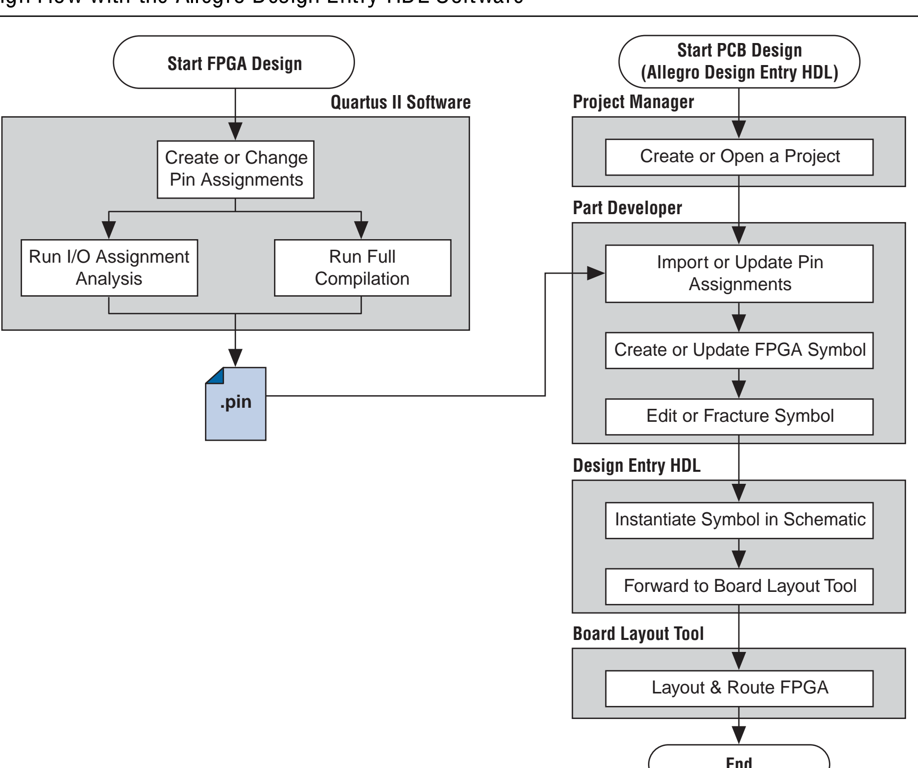
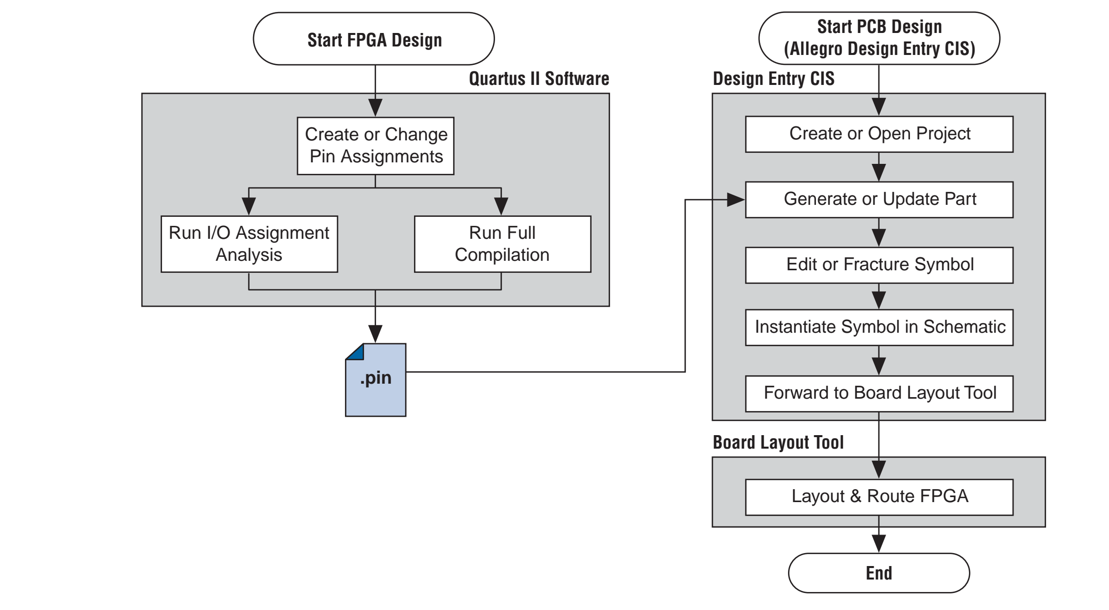


Figure 9–2. Design Flow with the Allegro Design Entry CIS Software



The basic steps in a complete design flow to integrate an Altera FPGA design starting in the Quartus II software through to a circuit schematic in Design Entry HDL or Design Entry CIS are as follows:

1. Start the Quartus II software.

2. In the Quartus II software, compile your design to generate a Pin-Out (**.pin**) file to transfer assignments to the Cadence tool.
3. If you are using the Cadence Allegro Design Entry HDL software for your schematic design:
   a. Open an existing project or create a new project in the Allegro Project Manager.
   b. Construct a new symbol or update an existing symbol using the Allegro PCB Librarian Part Developer.
   c. With the Part Developer, edit your symbol or fracture it into smaller parts, if desired.
   d. Instantiate the symbol in your Design Entry HDL software schematic and transfer the design to your board layout tool.
4. If you are using the Cadence Allegro Design Entry CIS software for your schematic design, perform the following steps:
   a. Generate a new part within an existing or new Allegro Design Entry CIS project, referencing the **.pin** file output from the Quartus II software. You can update an existing symbol with a new **.pin** file.
   b. Split the symbol into smaller parts as desired.
   c. Instantiate the symbol in your Design Entry CIS schematic and transfer the design to your board layout tool.

Figure 9–1 and Figure 9–2 show the possible design flows, depending on your tool choice. The Cadence PCB Librarian Expert license is required to use the PCB Librarian Part Developer to create FPGA symbols. You can update symbols with changes made to the FPGA design at any point using any of these tools.

Setting Up the Quartus II Software

You can transfer pin and signal assignments from the Quartus II software to the Cadence design tools by generating the Quartus II project **.pin** file. The **.pin** file is an output file generated by the Quartus II Fitter that contains pin assignment information. Use the Quartus II Pin Planner or Assignment Editor to set and change the assignments contained in the **.pin** file. This file cannot be used to import pin assignment changes into the Quartus II software. Use it only to transfer assignments for use with the Cadence design tools.

The **.pin** file lists all used and unused pins on your selected Altera device. It also provides the following basic information fields for each assigned pin on a device:

- Pin signal name and usage
- Pin number
- Signal direction
- I/O standard
- Voltage
- I/O bank
- User or Fitter-assigned

For information about using the Quartus II Pin Planner to create or change pin assignment details, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Generating .pin Files

The Quartus II software automatically generates the **.pin** file when your FPGA design is fully compiled or when you start I/O Assignment Analysis. To start I/O Assignment Analysis, on the Processing menu, point to **Start** and click **Start I/O Assignment Analysis**. The file is output by the Quartus II Fitter. The file is generated and placed in your Quartus II design directory with the name <*project name*>**.pin**. The Cadence design tools do not generate or change this file.

For more information about pin and signal assignment transfer and the files that the Quartus II software can import and export, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook.*

FPGA-to-Board Integration with the Cadence Allegro Design Entry HDL Software

The Cadence Allegro Design Entry HDL software is Cadence's high-end schematic capture tool (part of the Cadence 600 series design flow). Use this software to create flat circuit schematics for all types of PCB design. The Cadence Allegro Design Entry HDL software can also create hierarchical schematics to facilitate design reuse and team-based design. With the Cadence Allegro Design Entry HDL software, the design flow from FPGA-to-board is one-way, using only the **.pin** file generated by the Quartus II software. Signal and pin assignment changes can only be made in the Quartus II software and are reflected in updated symbols in a Design Entry HDL project.

Routing or pin assignment changes made in a board layout tool or a Design Entry HDL symbol cannot be back-annotated to the Quartus II software.

Figure 9–1 shows the design flow with the Cadence Allegro Design Entry HDL software.

For more information about the Cadence Allegro Design Entry HDL software and the Part Developer, including licensing, support, usage, training, and product updates, refer to the Help in the software or to the Cadence website at www.cadence.com.

Symbol Creation

In addition to circuit simulation, circuit board schematic creation is one of the first tasks required in the design of a new PCB. Schematics are required to understand how the PCB works, and to generate a netlist that is passed on to a board layout tool for board design and routing. The Allegro PCB Librarian Part Developer provides the ability to create schematic symbols based on FPGA designs exported from the Quartus II software.

Create symbols for Design Entry HDL with the Allegro PCB Librarian Part Developer available in the Allegro Project Manager. The Part Developer is the recommended method for importing FPGA designs into the Cadence Allegro Design Entry HDL software.

You must have a PCB Librarian Expert license from Cadence to run the Part Developer. The Part Developer provides a graphical interface with many options for creating, editing, fracturing, and updating symbols. If you do not use the Part Developer, you must create and edit symbols manually in the Symbol Schematic View in the Cadence Allegro Design Entry HDL software.

If you do not have a PCB Librarian Expert license, you can still automatically create FPGA symbols using the programmable IC (PIC) design flow found in the Allegro Project Manager. For more information about using the PIC design flow, refer to the Help in the Cadence design tools, or go to the Cadence website at www.cadence.com.

Before you create a symbol from an FPGA design, you must open or create a Design Entry HDL design project. You can do this with the Allegro Project Manager, the main interface to all of the Cadence tools.

To open an existing design in the Allegro Project Manager, on the File menu, click **Open** and select the main design file for your project (found in your Allegro Design Entry HDL project directory and called <*project directory*>**.cpm**).

To create a new project, on the File menu, point to **New** and click **New Design**. The New Project wizard appears. Use the wizard to name your new project, set the file location, and define associated part libraries.

Allegro PCB Librarian Part Developer

Create, fracture, and edit schematic symbols for your FPGA designs in Altera devices using the Part Developer. Most FPGA devices are physically large with hundreds of pins, requiring large schematic symbols that might not fit on a single schematic page. Symbols designed in the Part Developer can be split or fractured into a number of functional blocks called slots, allowing multiple smaller part fractures to exist on the same schematic page or across multiple pages. Figure 9–3 highlights how the Part Developer fits into the design flow.

Figure 9–3. Part Developer in the Design Flow

(1) **Part Developer**

.pin

Import or Update Pin Assignments

Create or Update FPGA Symbol

Edit or Fracture Symbol

(2) *Design Entry HDL*

Instantiate Symbol in Schematic

Forward to Board Layout Tool

Board Layout Tool

Layout & Route FPGA

End

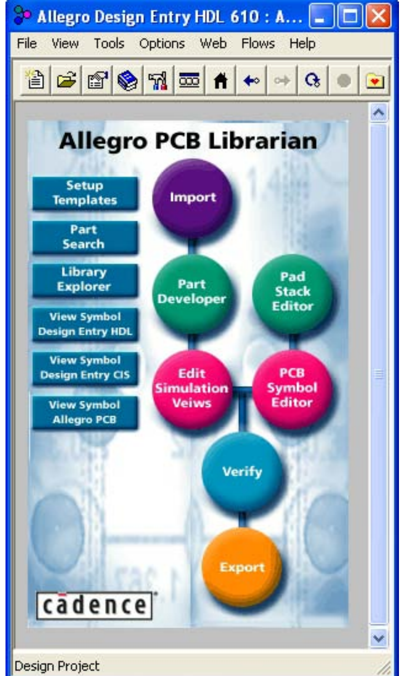
Notes to Figure 9–3:

(1) Refer to Figure 9–1 for the full design flow flowchart details.

(2) Grayed out steps are not part of the FPGA Symbol creation or update process.

Run the Part Developer from the Project Manager (Figure 9–4). To start the Part Developer in the Project Manager, on the Flows menu, click **Library Management**. Click **Part Developer** to start the tool.

Figure 9–4. Invoking the Part Developer from the Project Manager

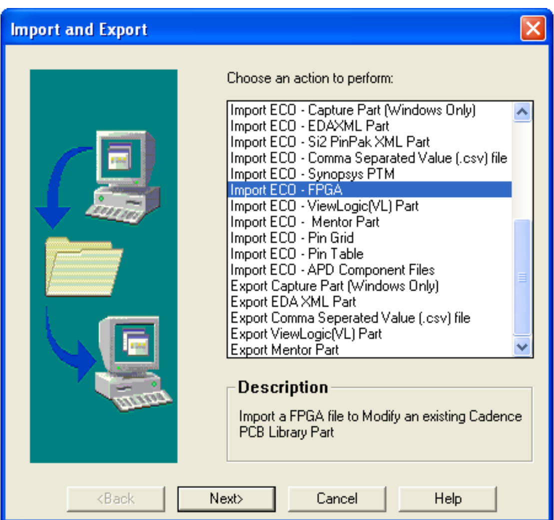


Import and Export Wizard

After you are in the Part Developer, you can use the Import and Export wizard to import your pin assignments from the Quartus II software. To access the wizard, perform the following steps:

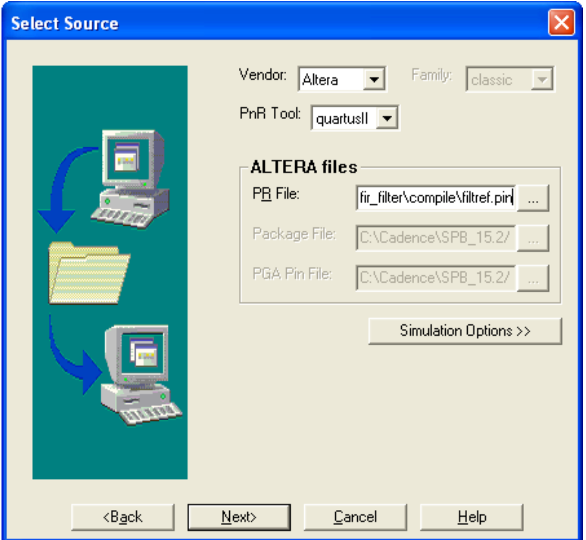
1. On the File menu, click **Import and Export**. The Import and Export wizard appears (Figure 9–5).

Figure 9–5. Import and Export Wizard



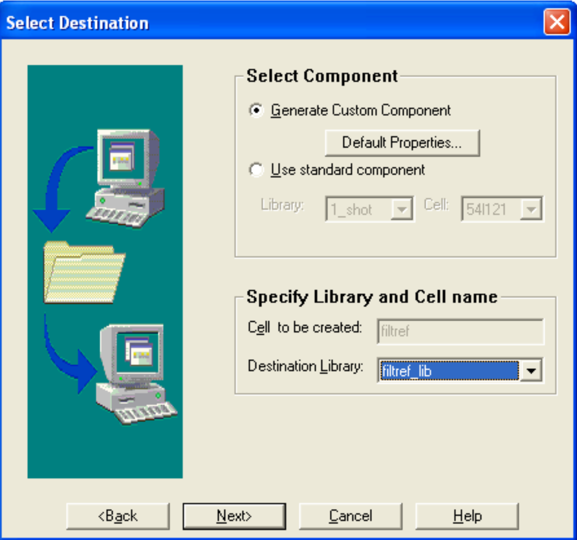
2. Select **Import ECO-FPGA**. Click **Next**. The **Select Source** dialog box appears (Figure 9–6).

Figure 9–6. Select Source Dialog Box



3. In the **Vendor** list, select **Altera.** In the **PnR Tool** list, select **quartusII**. To specify the **.pin** file in the **PR File** field, select the **.pin** file in your Quartus II project directory. Click **Simulation Options** if you want to select simulation input files. Click **Next**. The **Select Destination** dialog box appears (Figure 9–7).

Figure 9–7. Select Destination Dialog Box



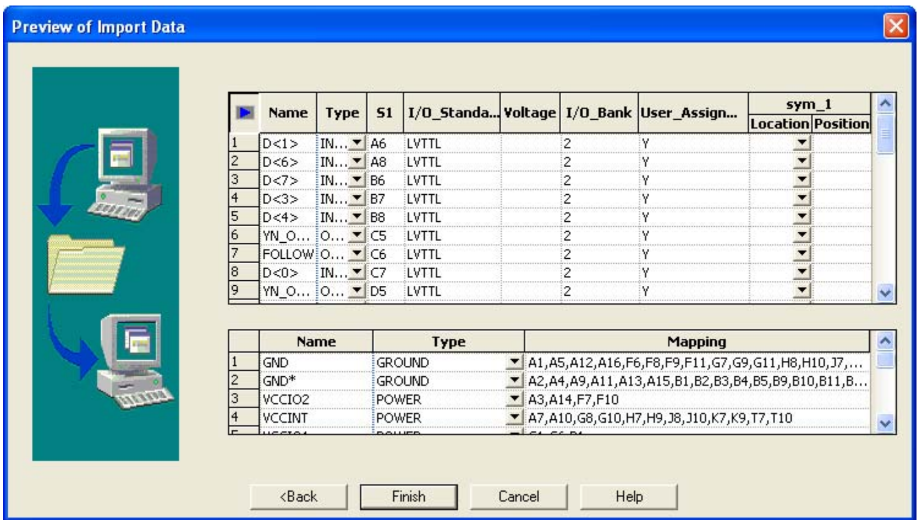
4. To create a new component in a library, click **Generate Custom Component**. To base your symbol on an existing component, click **Use standard component**.

You might want to do this if you previously created generic symbols for an FPGA device. You can place your pin and signal assignments from the Quartus II software on this symbol and reuse the symbol as a base any time you have a new FPGA design.

   In the **Library** list, select an existing library. You can now select from the cells contained in the selected library. Each cell represents all of the symbol versions and part fractures for that particular part. In the **Cell** list, select the existing cell to use as a base for your part.

5. In the **Destination Library** list, select a destination library for the component. Click **Next**. A preview of your import data appears (Figure 9–8).

Figure 9–8. Preview of Import Data Window



6. Review the assignments you are importing into the Part Developer based on the data in the **.pin** file. The location of each pin is not included in the information in this window, but inputs are placed on the left side of the created symbol, outputs on the right, power pins on the top, and ground pins on the bottom. Make any desired changes. When you have completed your changes, click **Finish** to create the symbol. The Part Developer main screen appears.

If the Part Developer is not set up to point to your PCB Librarian Expert license file, an error message displays in red at the bottom of the message text window of the Part Developer when you select the **Import and Export** command. To point to your PCB Librarian Expert license, on the File menu, click **Change Product** and select the correct product license.

For more information about licensing and obtaining licensing support, contact Cadence or refer to their website at www.cadence.com.

Edit and Fracture Symbol

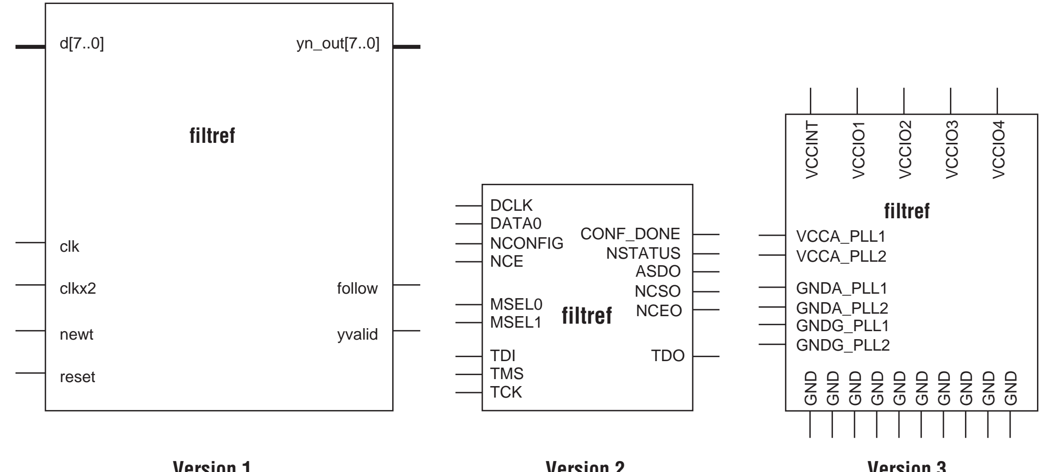
After you save your new symbol in the Part Developer software, you can edit the symbol graphics, fracture the symbol into multiple slots, and add or change package or symbol properties. These actions are available from the Part Developer main window.

The Part Developer Symbol Editor contains many graphical tools to edit the graphics of a particular symbol. Select the symbol in the cell hierarchy to edit the symbol graphics. The **Symbol Pins** tab appears. Edit the preview graphic of the symbol in the **Symbol Pins** tab.

Fracturing a Part Developer package into separate symbol slots is especially useful for FPGA designs. A single symbol for most FPGA packages might be too large for a single schematic page. Splitting the part into separate slots allows you to organize parts of the symbol by function, creating cleaner circuit schematics. For example, you could create one slot for an I/O symbol, a second slot for a JTAG symbol, and a third slot for a power/ground symbol.

Figure 9–9 shows a part fractured into separate slots.

Figure 9–9. Splitting a Symbol into Multiple Slots *(Note 1)*, *(2)*, *(3)*



Notes to Figure 9–9:

(1) Figure 9–9 represents a Cyclone device with JTAG or passive serial (PS) mode configuration option settings. Symbols created for other devices or other configuration modes might have different sets of configuration pins, but can be fractured in a similar manner.

(2) Symbol fractures are referred to in different ways in each of the tools described in this chapter. Refer to Table 9–2 for the specific tool naming conventions.

(3) The power/ground slot shows only a representation of power and ground pins. In actuality, the device contains a high number of power and ground pins.

While the Part Developer software refers to symbol fractures as slots, the other tools described in this chapter use different names to refer to symbol fractures. Table 9–2 lists the symbol fracture naming conventions for each of the tools addressed in this chapter.

Table 9–2. Symbol Fracture Naming

	Allegro PCB Librarian Part Developer Software	Allegro Design Entry HDL Software	Allegro Design Entry CIS Software
During symbol generation	Slots	—	Sections
During symbol schematic instantiation	—	Versions	Parts

To fracture a part into separate slots, or modify the slot locations of pins on parts that are already fractured in the Part Developer, perform the following steps:

1. Start the Cadence Allegro Design Project Manager.
2. On the Flows menu, click **Library Management**. The Library Management design flow appears. Click **Part Developer**. The Part Developer launches.
3. Click on the name of the package you want to change in the cell hierarchy. The **Package Pin** tab appears.

4. Click **Functions/Slots**. If you are not creating new slots but want to change the slot location of some pins, proceed to step 5. If you are creating new slots, click **Add**. A dialog box appears, allowing you to add extra symbol slots. Set the number of extra slots you want to add to the existing symbol, not the total number of desired slots for the part. Click **OK**.
5. Click **Distribute Pins**. Set the slot where each pin should reside. Use the checkboxes in each column to move pins from one slot to another. You can use the standard cut, copy, and paste keyboard commands on selected groups of checkboxes to move multiple pins from one slot to another. Click **OK**.
6. After distributing the pins, click the **Package Pin** tab and click **Generate Symbol(s)**. The **Generate Symbols** dialog box appears.
7. Select whether to create a new symbol or modify an existing symbol in each slot. Click **OK**.

The newly generated or modified slot symbols display as separate symbols in the cell hierarchy. Each of these symbols can be edited individually.

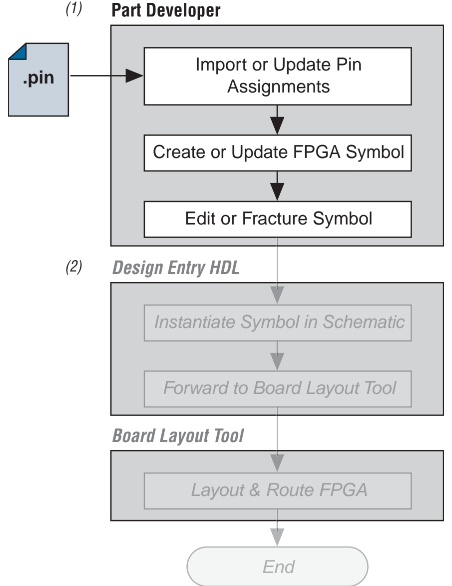
The Part Developer lets you remap pin assignments in the **Package Pin** tab of the main Part Developer window. If signals are remapped to different pins in the Part Developer, the changes are reflected only in regenerated symbols for use in your schematics. You cannot transfer pin assignment changes to the Quartus II software from the Part Developer, which creates a potential mismatch of the schematic symbols and assignments in the FPGA design. If pin assignment changes are necessary, make the changes in the Quartus II Pin Planner instead of the Part Developer, and update the symbol as described in the following sections.

For more information about creating, editing, and organizing component symbols with the Allegro PCB Librarian Part Developer, refer to the Part Developer Help.

Update FPGA Symbol

As the design process continues, you might have to make changes to the logic design in the Quartus II software, placing signals on different pins after the design is recompiled, or use the Quartus II Pin Planner to make changes manually. The board designer can request such changes to improve the board routing and layout. These types of changes must be carried forward to the circuit schematic and board layout tools to ensure signals are connected to the correct pins on the FPGA. Updating the **.pin** file in the Quartus II software facilitates this flow. Figure 9–10 shows this part of the design flow.

Figure 9–10. Updating the FPGA Symbol in the Design Flow



Notes to Figure 9–10:

(1) Refer to Figure 9–1 for the full design flow flowchart details.

(2) Grayed out steps are not part of the FPGA Symbol update process.

After the **.pin** file has been updated, perform the following steps to update the symbol using the Allegro PCB Librarian Part Developer:

1. On the File menu, click **Import and Export**. The Import and Export wizard appears.
2. In the list of actions to perform, select **Import ECO - FPGA**. Click **Next**. The **Select Source** dialog box appears.
3. Select the updated source of the FPGA assignment information. In the **Vendor** list, select **Altera**. In the **PnR Tool** list, select **quartusII**. In the **PR File** field, click **browse** to specify the updated **.pin** file in your Quartus II project directory. Click **Next**. The Select Destination window appears.
4. Select the source component and a destination cell for the updated symbol. To create a new component based on the updated pin assignment data, select **Generate Custom Component**. This replaces the cell listed under the **Specify Library and Cell** name header with a new, non-fractured cell. Any symbol edits or fractures are lost. You can preserve these edits by selecting **Use standard component and select the existing library and cell**. Select the destination library for the component and click **Next**. The **Preview of Import Data** dialog box appears.
5. Make any additional changes to your symbol. Click **Next**. A list of ECO messages displays summarizing what changes will be made to the cell. To accept the changes and update the cell, click **Finish**.
6. The main Part Developer window appears. You can edit, fracture, and generate the updated symbols as usual from this window.

If the Part Developer is not set up to point to your PCB Librarian Expert license file, an error message displays in red at the bottom of the message text window of the Part Developer when you select the **Import and Export** command. To point to your PCB Librarian Expert license, on the File menu, click **Change Product**, and select the correct product license. For more information about licensing and obtaining licensing support, contact Cadence or refer to their website at www.cadence.com.

Instantiating the Symbol in the Cadence Allegro Design Entry HDL Software

After the new symbol is saved in the Part Developer, perform the following steps to instantiate the symbol in your Design Entry HDL schematic:

1. In the Allegro Project Manager, switch to the board design flow.
2. On the Flows menu, click **Board Design**.
3. Click **Design Entry** to start the Design Entry HDL software.
4. To add the newly created symbol to your schematic, right-click in the main schematic window and choose **Add Component**, or on the Component menu, click **Add**. The **Add Component** dialog box appears.
5. Select the new symbol library location, and select the name of the cell you created from the list of cells.

The symbol is now “attached” to your cursor for placement in the schematic. If you fractured the symbol into slots, right-click the symbol and choose **Version** to select one of the slots for placement in the schematic.

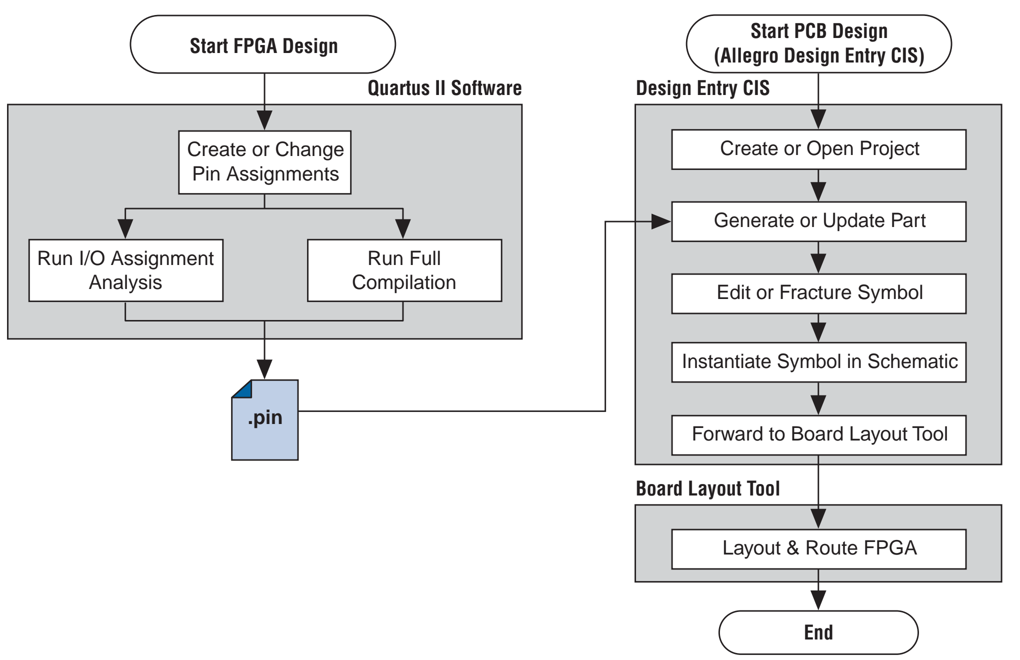
For more information about the Cadence Allegro Design Entry HDL software, including licensing, support, usage, training, and product updates, refer to the Help in the software or go to the Cadence website at www.cadence.com.

FPGA-to-Board Integration with Allegro Design Entry CIS

The Cadence Allegro Design Entry CIS software is Cadence’s mid-level schematic capture tool (part of the Cadence 200 series design flow based on OrCAD Capture CIS). Use this software to create flat circuit schematics for all types of PCB design. You can also create hierarchical schematics to facilitate design reuse and team-based design using this software. With the Cadence Allegro Design Entry CIS software, the design flow from FPGA-to-board is unidirectional using only the **.pin** file generated by the Quartus II software. Signal and pin assignment changes can only be made in the Quartus II software and are reflected in updated symbols in a Design Entry CIS schematic project.

Routing or pin assignment changes made in a board layout tool or a Design Entry CIS symbol cannot be back-annotated to the Quartus II software. Figure 9–11 shows the design flow with the Cadence Allegro Design Entry CIS software.

Figure 9–11. Design Flow with the Cadence Allegro Design Entry CIS Software



For more information about the Cadence Allegro Design Entry CIS software, including licensing, support, usage, training, and product updates, refer to the Help in the software, go to the Cadence website at www.cadence.com, or go to the EMA Design Automation website at www.ema-eda.com.

Allegro Design Entry CIS Project Creation

The Cadence Allegro Design Entry CIS software has built-in support for creating schematic symbols using pin assignment information imported from the Quartus II software.

If you have not already created a new project in the Cadence Allegro Design Entry CIS software, perform the following steps to create a new project:

1. On the File menu, point to **New** and click **Project.** The New Project wizard starts.

   When you create a new project, you can select the PC Board wizard, the Programmable Logic wizard, or a blank schematic.

2. Select the PC Board wizard to create a project where you can select which part libraries to use, or select a blank schematic.

The Programmable Logic wizard is used only to build an FPGA logic design in the Cadence Allegro Design Entry CIS software, which is unnecessary when using the Quartus II software.

No other special configuration for your project is required. Your new project is created in the specified location and initially consists of two files: the OrCAD Capture Project (**.opj**) file and the Schematic Design (**.dsn**) file.

Generate Part

After you create a new project or open an existing project in the Allegro Design Entry CIS software, you can generate a new schematic symbol based on your Quartus II FPGA design. You can also update an existing symbol if your **.pin** file has been updated in the Quartus II software. The Cadence Allegro Design Entry CIS software stores component symbols in OrCAD Library (**.olb**) files. When a symbol is placed in a library attached to a project, it is immediately available for instantiation in the project schematic.

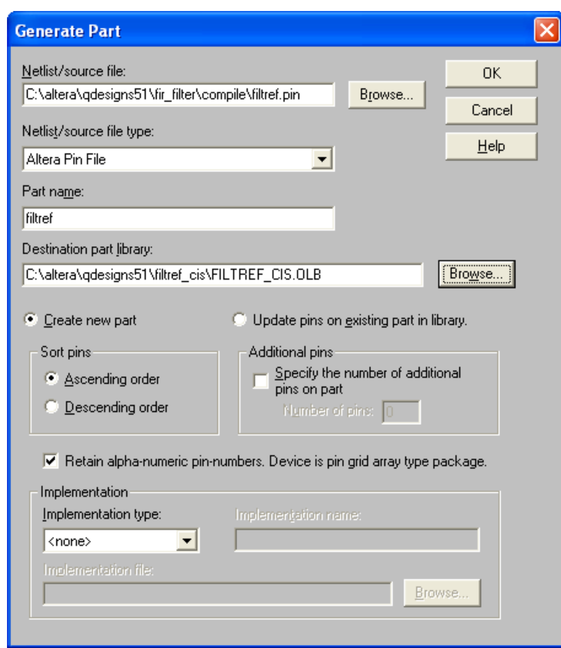
You can add symbols to an existing library or you can create a new library specifically for the symbols generated from your FPGA designs. To create a new library, perform the following steps:

1. On the File menu, point to **New** and click **Library** in the Cadence Allegro Design Entry CIS software to create a default library named **library1.olb**. This library appears in the **Library** folder in the Project Manager window of the Cadence Allegro Design Entry CIS software.
2. Right-click the new library and select **Save As** to specify a desired name and location for the library. The library file is not created until you save the new library.

You can now create a new symbol to represent your FPGA design in your schematic. To generate a schematic symbol, perform the following steps:

1. Start the Cadence Allegro Design Entry CIS software.
2. On the Tools menu, click **Generate Part**. The **Generate Part** dialog box appears (Figure 9–12).

Figure 9–12. Generate Part Dialog Box

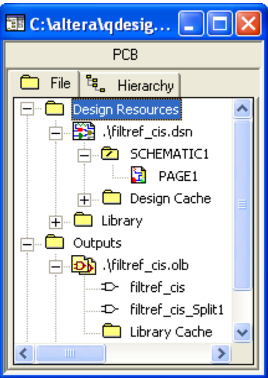


3. In the **Netlist/source file type** field, click **Browse** to specify the **.pin** file from your Quartus II design.

4. In the **Netlist/source file type** list, select **Altera Pin File**.
5. Enter the new part name.
6. Specify the **Destination part library** for the symbol. If you do not select an existing library for the part, a new library is created with a default name that matches the name of your Design Entry CIS project.
7. Select **Create new part** if you are creating a brand new symbol for this design. Select **Update pins on existing part in library** if you updated your **.pin** file in the Quartus II software and want to transfer any assignment changes to an existing symbol.
8. Select any other desired options and set **Implementation type** to **<none>**. The symbol is for a primitive library part based only on the **.pin** file and does not require a special implementation. Click **OK**.
9. Review the Undo warning and click **Yes** to complete the symbol generation.

The symbol is generated and placed in the selected library or in a new library found in the **Outputs** folder of the design in the Project Manager window (Figure 9–13). Double-click the name of the new symbol to see its graphical representation and edit it manually using the tools available in the Cadence Allegro Design Entry CIS software.

Figure 9–13. Project Manager Window



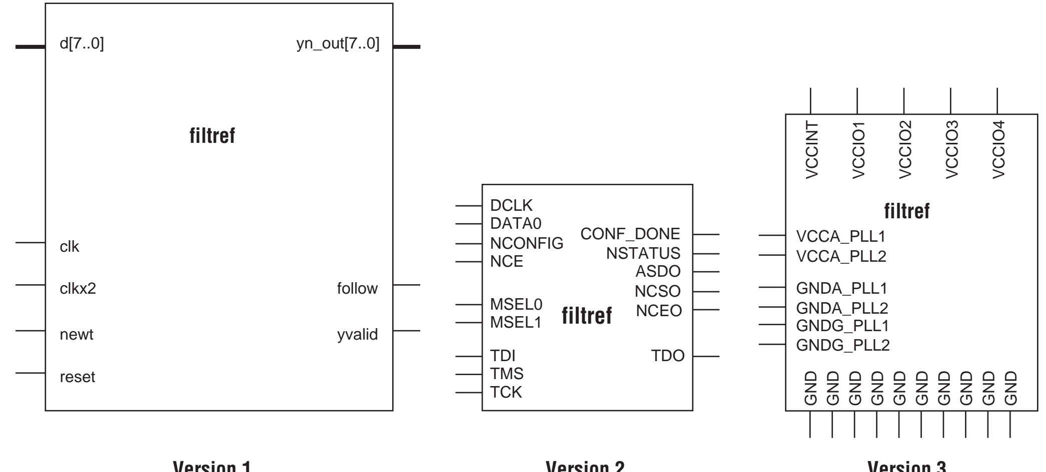
For more information about creating and editing symbols in the Allegro Design Entry CIS software, refer to the Help in the software.

Split Part

After a new symbol is saved in a project's library, you can fracture the symbol into multiple parts called sections. Fracturing a part into separate sections is especially useful for FPGA designs. A single symbol for most FPGA packages might be too large for a single schematic page. Splitting the part into separate sections allows you to organize parts of the symbol by function, creating cleaner circuit schematics. For example, you could create one slot for an I/O symbol, a second slot for a JTAG symbol, and a third slot for a power/ground symbol.

Figure 9–14 shows a part fractured into separate sections.

Figure 9–14. Splitting a Symbol into Multiple Sections *(Note 1)*, *(2)*, *(3)*



Notes to Figure 9–14:

(1) Figure 9–14 represents a Cyclone device with JTAG or passive serial (PS) mode configuration option settings. Symbols created for other devices or other configuration modes might have different sets of configuration pins, but can be fractured in a similar manner.

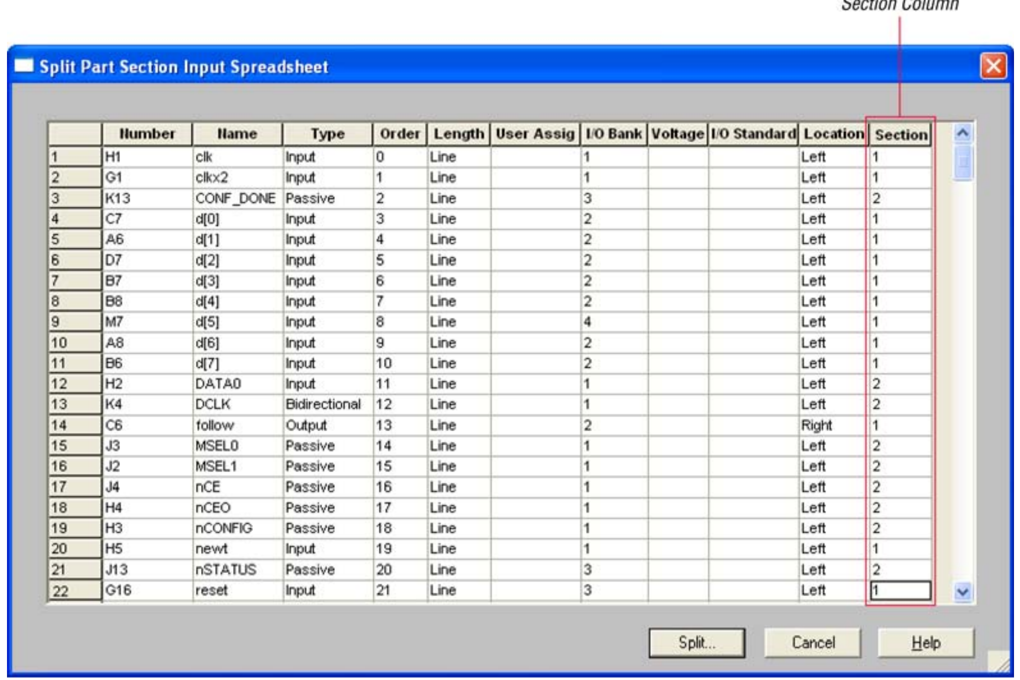
(2) Symbol fractures are referred to in different ways in each of the tools described in this chapter. Refer to Table 9–2 for the specific tool naming conventions.

(3) The power/ground section shows only a representation of power and ground pins. In actuality, the device contains a high number of power and ground pins.

While symbol generation in the Design Entry CIS software refers to symbol fractures as sections, the other tools described in this chapter use different names to refer to symbol fractures. Refer to Table 9–2 on page 9–11 for the symbol fracture naming conventions for each of the tools addressed in this chapter.

To split a part into sections, select the part in its library in the Project Manager window of Design Entry CIS. On the Tools menu, click **Split Part** or right-click the part and choose **Split Part**. The **Split Part Section Input Spreadsheet** appears (Figure 9–15).

Figure 9–15. Split Part Section Input Spreadsheet



	Number	Name	Type	Order	Length	User Assig	I/O Bank	Voltage	I/O Standard	Location	Section
1	H1	clk	Input	0	Line		1			Left	1
2	G1	clkx2	Input	1	Line		1			Left	1
3	K13	CONF_DONE	Passive	2	Line		3			Left	2
4	C7	d[0]	Input	3	Line		2			Left	1
5	A6	d[1]	Input	4	Line		2			Left	1
6	D7	d[2]	Input	5	Line		2			Left	1
7	B7	d[3]	Input	6	Line		2			Left	1
8	B8	d[4]	Input	7	Line		2			Left	1
9	M7	d[5]	Input	8	Line		4			Left	1
10	A8	d[6]	Input	9	Line		2			Left	1
11	B6	d[7]	Input	10	Line		2			Left	1
12	H2	DATA0	Input	11	Line		1			Left	2
13	K4	DCLK	Bidirectional	12	Line		1			Left	2
14	C6	follow	Output	13	Line		2			Right	1
15	J3	MSEL0	Passive	14	Line		1			Left	2
16	J2	MSEL1	Passive	15	Line		1			Left	2
17	J4	nCE	Passive	16	Line		1			Left	2
18	H4	nCEO	Passive	17	Line		1			Left	2
19	H3	nCONFIG	Passive	18	Line		1			Left	2
20	H5	newt	Input	19	Line		1			Left	1
21	J13	nSTATUS	Passive	20	Line		3			Left	2
22	G16	reset	Input	21	Line		3			Left	1

Each row in the spreadsheet represents a pin in the symbol. The spreadsheet column labeled **Section** indicates the section of the symbol to which each pin is assigned. By default, all pins in a new symbol are located in section 1. Change the values in this column to assign pins to different, new sections of the symbol. You can also specify the side of a section on which the pin will reside by changing the values in the **Location** column. When you are finished, click **Split**. A new symbol appears in the same library as the original with the name <*original part name*>**_Split1.**

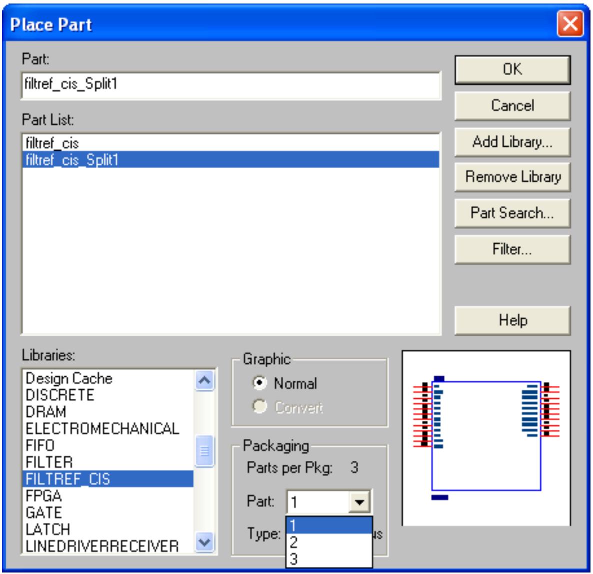
View and edit each section individually. To view the new sections of the part, double-click the part. The Part Symbol Editor window appears. The first section of the part is displayed for editing. On the View menu, click **Package** to view thumbnails of all the part sections. Double-click a thumbnail to edit that section of the symbol.

For more information about splitting parts into sections and editing symbol sections in the Cadence Allegro Design Entry CIS software, refer to the Help in the software.

Instantiate Symbol in Design Entry CIS Schematic

After a new symbol is saved in a library in your Design Entry CIS project, you can instantiate it on a page in your schematic. Open a schematic page in the Project Manager window of the Cadence Allegro Design Entry CIS software. On the schematic page, to add the newly created symbol to your schematic, on the Place menu, click **Part**. The **Place Part** dialog box appears (Figure 9–16).

Figure 9–16. Place Part Dialog Box



Select the new symbol library location and the newly created part name. If you select a part that is split into sections, you can select the section to place from the **Part** pop-up menu. Click **OK**. The symbol is now attached to your cursor for placement in the schematic. Click on the schematic page to place the symbol.

For more information about using the Cadence Allegro Design Entry CIS software, refer to the Help in the software.

Altera Libraries for Design Entry CIS

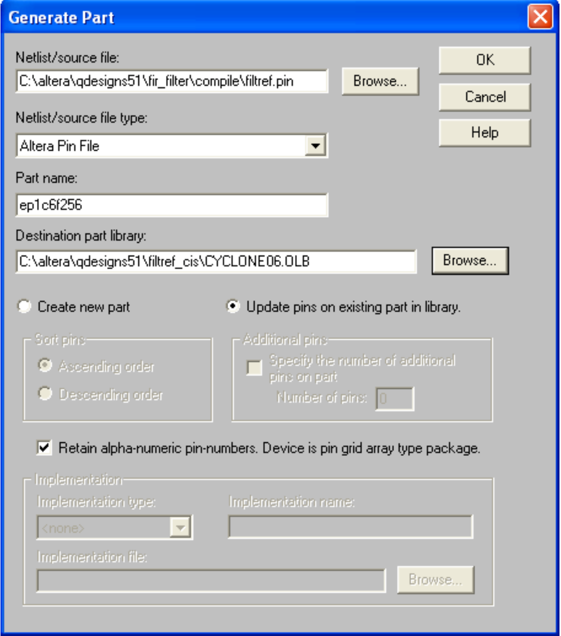
Altera provides downloadable **.olb** files for many of its device packages. You can add these libraries to your Design Entry CIS project and update the symbols with the pin assignments contained in the **.pin** file generated by the Quartus II software. This allows you to use the downloaded library symbols as a base for creating custom schematic symbols with your pin assignments that you can edit or fracture as desired. This can increase productivity by reducing the amount of time it takes to create and edit a new symbol.

To use the Altera-provided libraries with your Design Entry CIS project, perform the following steps:

1. Download the library of your target device from the Download Center page found through the Support page on the Altera website at www.altera.com.
2. Make a copy of the appropriate **.olb** file so that the original symbols are not altered. Place the copy in a convenient location, such as your Design Entry CIS project directory.
3. In the Project Manager window of the Cadence Allegro Design Entry CIS software, click once on the **Library** folder to select it. On the Edit menu, click **Project** or right-click the **Library** folder and choose **Add File** to select the copy of the downloaded **.olb** file and add it to your project. The new library is added to the list of part libraries for your project.

4. On the Tools menu, click **Generate Part.** The **Generate Part** dialog box appears (Figure 9–17).

Figure 9–17. Generate Part Dialog Box



5. In the **Netlist/source file** field, click **Browse** to specify the **.pin** file in your Quartus II design.
6. From the **Netlist/source file type** list, select **Altera Pin File**.
7. For **Part name**, enter the name of the target device the same as it appears in the downloaded library file. For example, if you are using a device from the **CYCLONE06.OLB** library, enter the part name to match one of the devices in this library such as `ep1c6f256`. You can rename the symbol later in the Project Manager window after the part is updated.
8. Set the **Destination part library** to the copy of the downloaded library you added to the project.
9. Select **Update pins on existing part in library**. Click **OK,** then click **Yes**.

The symbol is updated with your pin assignments. Double-click the symbol in the Project Manager window to view and edit the symbol. On the View menu, click **Package** if you want to view and edit other sections of the symbol. If the symbol in the downloaded library is already fractured into sections, as some of the larger packages are, you can edit each section but you cannot further fracture the part. Generate a new part without using the downloaded part library if you require additional sections.

For more information about creating, editing, and fracturing symbols in the Cadence Allegro Design Entry CIS software, refer to the Help in the software.

Conclusion

Transferring a complex, high-pin-count FPGA design to a PCB for prototyping or manufacturing is a daunting process that can lead to errors in the PCB netlist or design, especially when different engineers are working on different parts of the project. The design workflow available when the Quartus II software is used with tools from Cadence assists the FPGA designer and the board designer in preventing such errors and focusing all attention on the design.

Referenced Document

This chapter references the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Document Revision History

Table 9–3 shows the revision history for this chapter.

Table 9–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Chapter 9 was previously Chapter 7 in the 8.1 software release. ■ No change to content.	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	Updated references.	Updated for the Quartus II software version 8.0.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

Section III. Area, Timing and Power Optimization

Introduction

Physical implementation can be an intimidating and challenging phase of the design process. This section introduces features in Altera's Quartus® II software that you can use to achieve the highest design performance when you design for programmable logic devices (PLDs), especially high density FPGAs. The Quartus II software provides a comprehensive environment for FPGA designs, delivering unmatched performance, efficiency, and ease-of-use.

In a typical design flow, you must synthesize your design with Quartus II integrated synthesis or a third-party tool, place and route your design with the Fitter, and use the TimeQuest static timing analyzer to ensure your design meets the timing requirements. With the PowerPlay Power Analyzer, you ensure the design's power consumption is within limits. If your design does not meet all of your constraints, reiterate this process either partially or completely (based on the specific situation).

Refer to "Further Reading" for more information on any particular feature.

Physical Implementation

Most optimization issues are about preserving previous results, reducing area, reducing critical path delay, reducing power consumption, and reducing runtime. The Quartus II software includes advisors to address each of these issues and helps you optimize your design. Run these advisors during physical implementation for advice about your specific design situation.

You can reduce the time spent on design iterations by following the recommended design practices for designing with Altera® devices. Design planning is critical for successful design timing implementation and closure.

Trade Offs and Limitations

Many optimization goals can conflict with one another, so you might be required to make trade offs between different goals. For example, one major trade-off during physical implementation is between resource usage and critical path timing, because certain techniques (such as logic duplication) can improve timing performance at the cost of increased area. Similarly, a change in power requirements can result in area and timing trade offs. For example, if you reduce the number of high-speed tiles available, or if you attempt to shorten high-power nets at the expense of critical path nets.

In addition, system cost and time-to-market considerations can affect the choice of the device. For example, a device with a higher speed grade or more clock networks can facilitate timing closure at the expense of higher power consumption and system cost.

Finally, not all designs can be realized in a hardware circuit with limited resources and given constraints. If you encounter resource limitation, timing constraints, or power constraints that cannot be resolved by the Fitter, you might have to consider rewriting parts of the HDL code.

Preserving Results and Enabling Teamwork

Some of the Quartus II Fitter algorithms are pseudo-random in nature, which means that small changes to the design can have a large impact on the final result. For example, a critical path delay can change by 10% or more because of seemingly insignificant changes. If you are close to meeting your timing objectives, you can use the Fitter algorithm to your advantage by changing the fitter seed, which changes the pseudo-random result of the Fitter.

Conversely, if you have trouble meeting timing on a portion of your design, you can partition the troublesome portion and prevent it from recompiling if an unrelated part of the design is changed. This feature, known as incremental compilation, can reduce the Fitter runtimes by up to 70% if the design is partitioned, such that only small portions require recompilation at any one time.

When you use incremental compilation, you can apply design optimization options to individual design partitions and preserve performance in other partitions by leaving them untouched. Many of the optimization techniques often result in longer compilation times, but by applying them only on specific partitions, you can reduce this impact and complete more iterations per day.

In addition, by physically floorplanning your partitions with LogicLock, you can enable team-based flows and allow multiple people to work on different portions of the design.

Reducing Area

By default, the Quartus II Fitter might spread out a design to meet the set timing constraints. If you prefer to optimize your design to use the smallest area, you can change this behavior. If you require more area savings, you can enable certain physical synthesis options to modify your netlist to create more area-efficient implementation, but at the cost of increased runtime and decreased performance.

Reducing Critical Path Delay

To meet complex timing requirements involving multiple clocks, routing resources, and area constraints, the Quartus II software offers a close interaction between synthesis, timing analysis, floorplan editing, and place-and-route processes.

By default, the Quartus II Fitter tries to meet specified timing requirements and stops trying once the requirements are met. Therefore, using realistic constraints is important to successfully close timing. If you under-constrain your design, you are likely to get sub-optimal results. By contrast, if you over-constrain your design, the Fitter might over-optimize non-critical paths at the expense of true critical paths. In addition, you might incur an increased area penalty. Compilation time might increase because of excessively tight constraints.

If your resource use is very high, the Quartus II Fitter might have trouble finding a legal placement. In such circumstances, the Fitter automatically modifies some of its settings to try to trade off performance for area.

The Quartus II Fitter offers a number of advanced options that can help in improving the performance of your design when you properly set constraints. Use the Timing Optimization Advisor to determine which options are best suited for your design.

If you use incremental compilation, you can help resolve inter-partition timing requirements by locking down the results for each partition at a time or by guiding the placement of the partitions with LogicLock regions. You might be able to improve the timing on such paths by placing the partitions optimally to reduce the length of critical paths. Once your inter-partition timing requirements are met, use incremental compilation to preserve the results and work on partitions that have not met timing requirements.

In high-density FPGAs, routing accounts for a major part of critical path timing. Because of this, duplicating or retiming logic can allow the Fitter to shorten critical paths. The Quartus II software offers push-button netlist optimizations and physical synthesis options that can improve design performance at the expense of considerable increases of compilation time and area. Turn on only those options that help you keep reasonable compilation times. Alternately, you can modify your HDL to manually duplicate or retime logic.

Reduce Power Consumption

The Quartus II software has features that help reduce design power dissipation. The PowerPlay power optimization options control the power-driven compilation settings for Synthesis and Fitter.

Reducing Runtime

Many Fitter settings influence compilation time. Most of the default settings in the Quartus II software are set for reduced compilation time. You can modify these settings based on your project requirements.

The Quartus II software supports parallel compilation in computers with multiple processors. This can reduce compilation times by up to 15% while giving the identical result as serial compilation.

You can also reduce compilation time with your iterations by using incremental compilation. Use incremental compilation when you want to change parts of your design, while keeping most of the remaining logic unchanged.

Using Quartus II Tools

Design Analysis

The Quartus II software provides tools that help with a visual representation of your design. You can use the RTL Viewer to see a schematic representation of your design before behavioral simulation, synthesis, and place-and-route. The Technology Map Viewer provides a schematic representation of the design implementation in the selected device architecture after synthesis and place-and-route. It can also include timing information.

With incremental compilation, the Design Partition Planner and the Chip Planner allow you to partition and layout your design at a higher level. In addition, you can perform many different tasks with the Chip Planner, including: making floorplan assignments, implementing engineering change orders (ECOs), and performing power analysis. Also, you can analyze your design and achieve a faster timing closure with the Chip Planner. The Chip Planner provides physical timing estimates, critical path display, and routing congestion view to help guide placement for optimal performance.

Advisors

The Quartus II software includes several advisors to help you optimize your design. You can save time by following the recommendations in the timing optimization advisor, the area optimization advisor, and the power optimization advisor. These advisors give recommendations based on your project settings and your design constraints.

Design Space Explorer

Use the Design Space Explorer (DSE) to find optimum settings in the Quartus II software. DSE automatically tries different combinations of netlist optimizations and advanced Quartus II software compiler settings, and reports the best settings for your design. You can try different seeds with the DSE if you are fairly close to meeting timing requirements. Finally, the DSE can run the different compilations on multiple computers at once, which shortens the timing closure process.

Further Reading

This section includes the following chapters:

- Chapter 10, Area and Timing Optimization
- Chapter 11, Power Optimization
- Chapter 12, Analyzing and Optimizing the Design Floorplan
- Chapter 13, Netlist Optimizations and Physical Synthesis
- Chapter 14, Design Space Explorer

Other supporting documents in volume 1 of the Quartus II Handbook are:

- *Design Planning with the Quartus II Software*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Designs*
- *Design Recommendations for Altera Devices and the Quartus II Design Assistant*
- *Recommended HDL Coding Styles*
- *Section IV. Engineering Change Management*

10. Area and Timing Optimization

QII52005-9.0.0

Introduction

Good optimization techniques are essential for achieving the highest possible quality of results when designing for programmable logic devices (PLDs). The optimization features available in the Quartus® II software are designed to allow you to meet design requirements by applying these techniques at multiple points in the design process.

This chapter explains techniques to reduce resource usage, improve timing performance, and reduce compilation times when designing for Altera® devices. It also explains how and when to use some of the features described in other chapters of the *Quartus II Handbook*. This introduction describes the various stages in a design optimization process, and points you to the appropriate sections in the chapter for area, timing, or compilation time optimization.

Topics in this chapter include:

- "Initial Compilation: Required Settings" on page 10–3
- "Initial Compilation: Optional Settings" on page 10–6
- "Design Analysis" on page 10–11
- "Resource Utilization Optimization Techniques (LUT-Based Devices)" on page 10–19
- "Timing Optimization Techniques (LUT-Based Devices)" on page 10–32
- "Resource Utilization Optimization Techniques (Macrocell-Based CPLDs)" on page 10–55
- "Timing Optimization Techniques (Macrocell-Based CPLDs)" on page 10–60
- "Compilation-Time Optimization Techniques" on page 10–64
- "Other Optimization Resources" on page 10–70
- "Scripting Support" on page 10–71

The application of these techniques varies from design to design. Applying each technique does not always improve design results. Settings and options in the Quartus II software have default values that generally provide the best trade-off between compilation time, resource utilization, and timing performance. You can adjust these settings to determine whether other settings provide better results for your design.

When using advanced optimization settings and tools, it is important to benchmark their effect on your quality of results and to use them only if they improve results for your design.

Use the optimization flow described in this chapter to explore various compiler settings and determine the techniques that provide the best results.

Optimizing Your Design

The first stage in the optimization process is to perform an initial compilation to view the quality of results for your design. "Initial Compilation: Required Settings" on page 10–3 provides guidelines on some of the settings and assignments that are recommended for your initial compilation. "Initial Compilation: Optional Settings" on page 10–6 describes settings that you might have to turn on based on your design requirements. "Design Analysis" on page 10–11 explains how to analyze the compilation results.

You can use incremental compilation as part of the optimization process. Incremental compilation can be used as a tool for timing preservation that aids in timing closure, as well as compilation time reduction; however, it can cause a slight increase in resource utilization.

For more details about incremental compilation with the Quartus II software, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

After you have analyzed the compilation results, perform the optimization stages in the recommended order, as described in this chapter.

For LUT-based devices (FPGAs, MAX® II series of devices), perform optimizations in the following order:

1. If your design does not fit, refer to "Resource Utilization Optimization Techniques (LUT-Based Devices)" on page 10–19 before trying to optimize I/O timing or register-to-register timing.
2. If your design does not meet the required I/O timing performance, refer to "I/O Timing Optimization Techniques (LUT-Based Devices)" on page 10–73 before trying to optimize register-to-register timing.
3. If your design does not meet the required slack on any of the clock domains in the design, refer to "Register-to-Register Timing Optimization Techniques (LUT-Based Devices)" on page 10–40.

For macrocell-based devices (MAX 7000 and MAX 3000 CPLDs), perform optimizations in the following order:

1. If your design does not fit, refer to "Resource Utilization Optimization Techniques (Macrocell-Based CPLDs)" on page 10–55 before trying to optimize I/O timing or register-to-register timing.
2. If your timing performance requirements are not met, refer to "Timing Optimization Techniques (Macrocell-Based CPLDs)" on page 10–60.
3. For device-independent techniques to reduce compilation time, refer to "Compilation-Time Optimization Techniques" on page 10–64.

You can use all these techniques in the GUI or with Tcl commands. For more information about scripting techniques, refer to "Scripting Support" on page 10–71.

Initial Compilation: Required Settings

This section describes the basic assignments and settings for your initial compilation. Ensure that you check all the following suggested compilation assignments before compiling the design in the Quartus II software. Significantly different compilation results can occur depending on the assignments you have set.

The following settings are required:

- "Device Settings"
- "I/O Assignments"
- "Timing Requirement Settings" on page 10–4
- "Device Migration Settings" on page 10–5
- "Partitions and Floorplan Assignments for Incremental Compilation" on page 10–6

Device Settings

Assigning a specific device determines the timing model that the Quartus II software uses during compilation. Choose the correct speed grade to obtain accurate results and the best optimization. The device size and the package determine the device pin-out and how many resources are available in the device.

To choose the target device, on the Assignments menu, click **Device**.

In a Tcl script, use the following command to set the device:

```
set_global_assignment -name DEVICE <device> ↵
```

I/O Assignments

The I/O standards and drive strengths specified for a design affect I/O timing. Specify I/O assignments so that the Quartus II software uses accurate I/O timing delays in timing analysis and Fitter optimizations.

The Quartus II software can choose pin locations automatically for best quality of results. If your pin locations are not fixed due to PCB layout requirements, leave pin locations unconstrained to achieve the best results. If your pin locations are already fixed, make pin assignments to constrain the compilation appropriately. "Resource Utilization Optimization Techniques (Macrocell-Based CPLDs)" on page 10–55 includes recommendations for making pin assignments that can have a larger effect on your quality of results in smaller macrocell-based architectures.

Use the Assignment Editor and Pin Planner to assign I/O standards and pin locations.

For more information about I/O standards and pin constraints, refer to the appropriate device handbook. For information about planning and checking I/O assignments, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*. For information about using the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

Timing Requirement Settings

Using comprehensive timing requirement settings is an important step for achieving the optimal quality of results, for the following reasons:

- Correct timing assignments allow the software to work hardest to optimize the performance of the timing-critical parts of the design and make trade-offs for performance. This optimization can also save area or power utilization in non-critical parts of the design.
- If enabled, the Quartus II software performs physical synthesis optimizations based on timing requirements (refer to "Physical Synthesis Optimizations" on page 10–41 for more information).
- Depending on the **Fitter Effort** setting, the Quartus II Fitter can reduce runtime considerably if your timing requirements are being met. For a full description of the different effort levels, refer to "Fitter Effort Setting" on page 10–10.

As a general rule, do not over-constrain the software by applying timing requirements that are higher than your design requirements. Use your real design requirements to get the best results. Power utilization might also be larger in an over-constrained design when the software balances power and performance during compilation.

Both the Classic and TimeQuest Timing Analyzers check your design against the timing assignments. The compilation report and timing analysis reporting commands show whether timing requirements are met and provide detailed timing information about paths that violate timing requirements.

To make clock and timing assignments for the Quartus II TimeQuest Timing Analyzer, create a Synopsys Design Constraint (**.sdc**) file that contains all of your constraints. You can also create constraints in the TimeQuest GUI. Use the `write_sdc` command, or, in the TimeQuest Timing Analyzer, on the Constraints menu, click **Write SDC File** to write your constraints to an **.sdc** file. You can add an **.sdc** file to your project on the **TimeQuest Timing Analyzer** page under **Timing Analysis Settings**.

To make clock assignments for the Quartus II Classic Timing Analyzer, on the Assignments menu, click the **Timing Analysis Settings**. Select the **Classic Timing Analyzer Settings** page. Use the **Delay requirements**, **Minimum delay requirements**, and **Clock Settings** boxes to make global settings, or to apply settings to individual clocks, click **Individual Clocks** (recommended for multiple-clock designs). Create the clock setting and apply it to the appropriate clock node in the design. The Timing Wizard can also step you through the process of making individual clock constraints for the Quartus II Classic Timing Analyzer. To run the Timing Wizard, on the Assignments menu, click **Timing Wizard**.

The Classic Timing Analyzer is not supported for designs using some of the newer Altera device families.

© March 2009 Altera Corporation

Ensure that every clock signal has an accurate clock setting assignment. If clocks come from a common oscillator, they can be considered related. Ensure that all related or derived clocks are set up correctly in the assignments. All I/O pins that require I/O timing optimization must have settings. You should also specify minimum timing constraints as applicable. If there is more than one clock or there are different I/O requirements for different pins, make multiple clock settings and individual I/O assignments instead of using the global settings.

Make any complex timing assignments required in the design, including any cut-timing and multicycle path assignments. Common situations for these types of assignments include reset or static control signals, cases in which it is not important how long it takes a signal to reach a destination, and paths that can operate in more than one clock cycle. These assignments allow the Quartus II software to make appropriate trade-offs between timing paths and can enable the Compiler to improve timing performance in other parts of the design.

For more information about timing assignments and timing analysis, refer to the *Quartus II TimeQuest Timing Analyzer* and *Quartus II Classic Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*. For more information on how to specify multicycle exceptions in the TimeQuest Timing Analyzer, refer to *AN 481: Applying Multicycle Exceptions in the TimeQuest Timing Analyzer*.

Timing Constraint Check—Report Unconstrained Paths

To ensure that all constraints or assignments have been applied to design nodes, you can report all unconstrained paths in your design.

While using the Quartus II TimeQuest Timing Analyzer, you can report all the unconstrained paths in your design with the Report Unconstrained Paths command (`report_ucp`) in the Task pane.

If you are using the Quartus II Classic Timing Analyzer, perform the following steps:

1. On the Assignments menu, click **Timing Analysis Settings**. The **Settings** dialog box appears.
2. Under **Timing Analysis Settings**, select **Classic Timing Analyzer Settings**.
3. Click **More Settings**. The **More Timing Settings** dialog box appears.
4. From the **Existing option settings** list, select **Report Unconstrained Paths**. From the **Setting** list, select **On**.

Device Migration Settings

If you anticipate a change to the target device later in the design cycle, either because of changes in the design or other considerations, plan for it at the beginning of your design cycle. Whenever you select a target device in **Settings** dialog box, you can also list any other compatible devices you can migrate to by clicking on the **Migration Devices** button on the **Device** page. If you plan to move your design to a HardCopy® device, make sure to select the device from the list under the **Companion device** tab on the **Device** page.

Selecting the migration device and companion device early in the design cycle helps you to minimize changes to the design at a later stage.

Partitions and Floorplan Assignments for Incremental Compilation

The Quartus II incremental compilation feature enables hierarchical and team-based design flows in which you compile parts of your design while other parts of the design remain unchanged, or import parts of your design from separate Quartus II projects.

Using the incremental compilation methodology with a good partitioning of the design can often help to achieve timing closure. Creating LogicLock™ regions and using incremental compilation can help you achieve timing closure block by block, and preserve the timing performance between iterations, which helps achieve timing closure for the entire design.

Using incremental compilation also helps reduce compilation times. For information about using the incremental compilation feature to reduce your compilation time, refer to "Incremental Compilation" on page 10–64.

If you want to take advantage of this feature for a team-based design flow to reduce your compilation times, or to improve the timing performance of your design during iterative compilation runs, make meaningful design partitions as well as create a floorplan for your design partitions. Assignments can negatively affect a design's quality of results if you do not follow Altera's recommendations. Good assignments can improve your quality of results.

If you plan to use incremental compilation, you must create a floorplan for your design. If you are not using incremental compilation, this step is optional.

For guidelines about how to create partition and floorplan assignments for your design, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Initial Compilation: Optional Settings

This section describes the settings that are optional, but that can help to compile your design. You can selectively set all the optional settings that help to improve performance (if required) and reduce compilation time. These settings vary between designs and there is no standard set that applies to all designs. Significantly different compilation results can occur depending on the assignments you have set.

The following settings are optional:

- "Design Assistant" on page 10–7
- "Smart Compilation Setting" on page 10–7
- "Early Timing Estimation" on page 10–7
- "Optimize Hold Timing" on page 10–8
- "Asynchronous Control Signal Recovery/Removal Analysis" on page 10–9
- "Limit to One Fitting Attempt" on page 10–9

Design Assistant

You can run the Design Assistant to analyze the post-fitting results of your design during a full compilation. The Design Assistant checks rules related to areas such as gated clocks, reset signals, asynchronous design practices, and signal race conditions. This is especially useful during the early stages of your design, so that you can work on any areas of concern in your design before proceeding with design optimization.

On the Assignments menu, click **Settings**. In the **Category** list, select **Design Assistant** and turn on **Run Design Assistant during compilation**.

You can also specify which rules you want the Design Assistant to apply when analyzing and generating messages for a design.

For more information about the rules in the Design Assistant, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

Smart Compilation Setting

Smart compilation can reduce compilation time by skipping compiler stages that are not required to recompile the design. This is especially useful when you perform multiple compilation iterations during the optimization phase of the design process. However, smart compilation uses more disk space. To turn on smart compilation, on the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Process Settings** and turn on **Use smart compilation**.

This feature skips entire compiler stages (such as Analysis and Synthesis) when they are not required. This feature is different from incremental compilation, which you can use to compile parts of your design while preserving results for unchanged parts. For information about using the incremental compilation feature to reduce your compilation time, refer to "Incremental Compilation" on page 10–64.

Early Timing Estimation

The Quartus II software provides an Early Timing Estimation feature that estimates your design's timing results before the software performs full placement and routing. On the Processing menu, point to **Start**, and click **Start Early Timing Estimate** to generate initial compilation results after you have run analysis and synthesis. When you want a quick estimate of a design's performance before proceeding with further design or synthesis tasks, this command can save significant compilation time. Using this feature provides a timing estimate up to 45× faster than running a full compilation, though the fit is not fully optimized or routed. Therefore, the timing report is only an estimate. On average, the estimated delays are within 11% of those achieved by a full compilation compared to the final timing results.

You can specify what type of delay estimates to use with this feature. On the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Process Settings**, and select **Early Timing Estimate**. On the **Early Timing Estimate** page, the following options are available:

- The **Realistic** option, which is the default, generates delay estimates that are likely to be close to the results of a full compilation.

- The **Optimistic** option uses delay estimates that are lower than those likely to be achieved by a full compilation, which results in an optimistic performance estimate.
- The **Pessimistic** option uses delay estimates that are higher than those likely to be achieved by a full compilation, which results in a pessimistic performance estimate.

All three options offer the same reduction in compilation time.

You can use the Chip Planner or the Timing Closure Floorplan (for older devices) to view the placement estimate created by this feature to identify critical paths in the design. Then, if necessary, you can add or modify floorplan constraints such as LogicLock regions, or make other changes to the design. You can then rerun the Early Timing Estimator to quickly assess the impact of any floorplan assignments or logic changes, enabling you to try different design variations and find the best solution.

Optimize Hold Timing

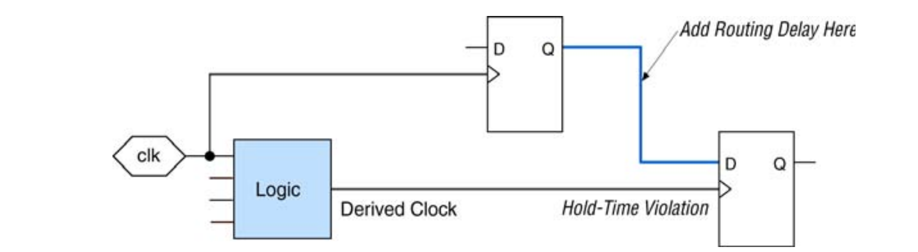
The **Optimize Hold Timing** option directs the Quartus II software to optimize minimum delay timing constraints. This option is available only for the Arria® GX, Stratix®, Cyclone®, MAX II series of devices, and HardCopy III and HardCopy II ASICs. By default, the Quartus II software optimizes hold timings for all paths for designs using newer devices such as Arria II GX, Arria GX, Stratix III, Stratix IV, and Cyclone III devices. By default, the Quartus II software optimizes hold timings only for I/O paths and minimum TPD paths for older devices.

When you turn on **Optimize Hold Timing**, the Quartus II software adds delay to paths to guarantee that the minimum delay requirements are satisfied. In the **Fitter Settings** panel, if you choose **I/O Paths and Minimum TPD Paths** (the default choice for older devices such as Cyclone II and Stratix II family of devices if you turn on **Optimize Hold Timing**), the Fitter works to meet the following criteria:

- Hold times (t_H) from device input pins to registers
- Minimum delays from I/O pins to I/O registers or from I/O registers to I/O pins
- Minimum clock-to-out time (t_{CO}) from registers to output pins

If you select **All Paths**, the Fitter also works to meet hold requirements from registers to registers, as in Figure 10–1, where a derived clock generated with logic causes a hold time problem on another register. However, if your design has internal hold time violations between registers, Altera recommends that you correct the problems by making changes to your design, such as using a clock enable signal instead of a derived or gated clock.

Figure 10–1. Optimize Hold Timing Option Fixing an Internal Hold Time Violation



For design practices that can help eliminate internal hold time violations, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

Asynchronous Control Signal Recovery/Removal Analysis

This option determines whether you want the software to analyze the results of recovery, and removal checks for paths that end at an asynchronous clear, preset, or load signal of a register. Recovery time is the minimum length of time an asynchronous control signal, for example, clear and preset, must be stable before the active clock edge. Removal time is the minimum time an asynchronous control signal must be stable after the active clock edge. Recovery and removal requirements are similar to setup and hold time requirements, respectively.

When using TimeQuest for timing analysis, Recovery/Removal analysis and optimization are always performed during placement and routing. You can use the `create_timing_summary` Tcl command to report the recovery and removal analysis. The slack for Removal/Recovery is determined in a similar way to Setup and Hold checks.

When using the Quartus II Classic Timing Analyzer for timing analysis, Recovery/Removal analysis is turned off by default. Turning on the option adds additional constraints during placement and routing, which can increase compilation time and reduce performance. If this analysis is required, on the Assignments menu, click **Settings**. In the **Category** list, select **Timing Requirements & Options**, then click **More Settings**. Turn on **Enable Recovery/Removal analysis**. By running the Recovery/Removal analysis, you can make sure that no timing failures are related to the asynchronous control signals in your design.

For more details about Recovery/Removal analysis with the TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

For designs containing FIFOs, Altera recommends turning on Recovery/Removal analysis. Recovery/Removal analysis helps to analyze corner-case conditions to achieve better functional coverage.

Limit to One Fitting Attempt

A design might fail to fit for several reasons, such as logic overuse or illegal assignments. For most of these failures, the Quartus II software informs you of the problem. However, if the design uses too much routing, the Quartus II software makes up to two additional attempts to fit your design, each time trying harder than the last to reduce routing resource utilization. Each of these fit attempts takes significantly longer than the previous attempt.

However, for large designs, you might not wish to wait for all three attempts, and would rather receive an error message earlier. This would allow you to take corrective action sooner. To get an early error indication, turn on **Limit to one fitting attempt** from the **Fitter Settings** page. When you choose this setting for your design, the Quartus II software quits with an error message after the first failed fitting attempt.

Refer to “Routing” on page 10–29 for instructions about how to lower the design’s routing utilization, so your design can be made to fit into the target device if it fails to fit due to the lack of routing resources.

Optimize Multi-Corner Timing

Historically, FPGA timing analysis has been performed using only worst-case delays, which are described in the slow corner timing model. However, due to process variation and changes in the operating conditions, delays on some paths can be significantly smaller than those in the slow corner timing model. This can result in hold time violations on those paths, and in rare cases, additional setup time violations.

By default, the Fitter optimizes constraints using only the slow corner timing model. You can turn on the **Optimize multi-corner timing** option to instruct the Fitter to also optimize constraints using the fast corner timing model and operating conditions, as well as the slow corner timing model and operating conditions to avoid the possible timing violations described above. By optimizing for both fast and slow process corners, you can create a design implementation that is more robust across process, temperature, and voltage variations. This option is available only for Arria GX, Stratix, Cyclone, and MAX II series of devices.

To turn on the **Optimize fast-corner timing** option, on the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings** and turn on **Optimize fast-corner timing**. Using the two different timing models can be important to account for process, voltage, and temperature variations for each device. Turning this option on increases compilation time by approximately 10%.

For designs with external memory interfaces such as DDR and QDR, Altera recommends that you turn on the **Optimize fast-corner timing** setting.

Fitter Effort Setting

To set the Fitter effort, on the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**. The **Fitter effort** settings are **Auto Fit**, **Standard Fit**, and **Fast Fit**. The default setting depends on the device family specified.

Auto Fit

The **Auto Fit** option (available for Arria GX, Stratix, Cyclone, and MAX II series of devices) focuses the full Fitter effort on those aspects of the design that require further optimization. **Auto Fit** can significantly reduce compilation time relative to **Standard Fit** if your design has some easy-to-meet timing requirements, low routing resource utilization, or both. However, those designs that require full optimization generally receive the same effort as is achieved by selecting **Standard Fit**. **Auto Fit** is the default Fitter effort setting for all devices for which this option is available.

If you want the Fitter to attempt to exceed the timing requirements by a certain margin instead of simply meeting them, specify a minimum slack in the **Desired worst case slack** box.

Specifying a minimum slack does not guarantee that the Fitter achieves the slack requirement; it only guarantees that the Fitter applies full optimization unless the target slack is exceeded.

In some designs with multiple clocks, it might be possible to improve the timing performance on one clock domain while reducing the performance on other clock domains by over-constraining the most important clock. If you use this technique, ensure that any performance improvements that you see are real gains by performing a sweep over multiple seeds. For more information, refer to "Fitter Seed" on page 10–47.

Over-constraining the clock for which you require maximum slack, while using the **Auto Fit** option, increases the chances that the Fitter is able to meet this requirement.

The **Auto Fit** option also causes the Quartus II Fitter to optimize for shorter compilation times instead of maximum performance if the design includes no timing assignments. For designs with no timing assignments, the resulting f_{MAX} is, on average, 10% lower than that achieved using the **Standard Fit** option. If your design has aggressive timing requirements or is hard to route, the placement does not stop early and the compilation time is the same as using the **Standard Fit** option. For designs with no timing requirements or easily achieved timing requirements, you can achieve an average compilation time reduction of 40% by using the **Auto Fit** option.

The **Auto Fit** option might increase the number of routing wires used. This can lead to an increase in the dynamic power when compared to using the **Standard Fit** option, unless the **Extra effort** option in the **PowerPlay power optimization** list is also enabled. When you turn on **Extra effort**, **Auto Fit** continues to optimize for reduction of wire usage even after meeting the register-to-register requirement. There is no adverse effect on the dynamic power consumption. If dynamic power consumption is a concern, select **Extra effort** in both the **Analysis & Synthesis Settings** and the **Fitter Settings** pages.

For more details, refer to the "Power Driven Compilation" section in the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Standard Fit

Use the **Standard Fit** option to exceed specified timing requirements and achieve the best possible timing results and lowest routing resource utilization for your design. However, this setting usually increases compilation time relative to **Auto Fit**, because it applies full optimization, regardless of the design requirement.

Fast Fit

The **Fast Fit** option reduces the amount of optimization effort for each algorithm employed during fitting. This option reduces the compilation time by about 50%, resulting in a fit that has, on average, 10% lower f_{MAX} than that achieved using the **Standard Fit** setting.

Design Analysis

The initial compilation establishes whether the design achieves a successful fit and meets the specified performance. This section describes how to analyze your design results in the Quartus II software. After design analysis, proceed to optimization as described in "Optimizing Your Design" on page 10–2.

Error and Warning Messages

After your initial compilation, it is important to evaluate all error and warning messages to see if any design or setting changes are required. If required, make these changes and recompile the design before proceeding with design optimization.

To suppress messages that you have evaluated and that can be ignored, right-click on the message in the Messages window and click **Suppress**.

For more information about message suppression, refer to the "Message Suppression" section in the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

Ignored Timing Assignments

You can use the **Ignored Timings Assignments** page in the Compilation Report to view any assignments that were ignored by the Quartus II Classic Timing Analyzer during the previous compilation. The Quartus II Classic Timing Analyzer ignores assignments that are invalid, conflict with other assignments, or become obsolete through the use of other assignments. If any assignments are ignored, analyze why they were ignored. If necessary, correct the assignments and recompile the design before proceeding with design optimization.

If you are using TimeQuest for timing analysis, you can list any ignored constraints by clicking **Report Ignored Constraints** in the Reports menu in the TimeQuest GUI or by using the following command to generate a listing of ignored timing constraints:

```
report_sdc -ignored -panel_name "Ignored Constraints" ↵
```

For more information about the `report_sdc` command and its options, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Resource Utilization

Determining device utilization is important regardless of whether a successful fit is achieved. If your compilation results in a no-fit error, resource utilization information is important for analyzing the fitting problems in your design. If your fitting is successful, review the resource utilization information to determine whether the future addition of extra logic or other design changes can introduce fitting difficulties.

To determine resource usage, refer to the **Flow Summary** section of the Compilation Report. This section reports how many resources are used, including pins, memory bits, digital signal processing (DSP) block 9-bit elements (for Arria GX, Stratix, and Stratix II devices) or 18-bit elements (for Arria II GX, Stratix IV and Stratix III devices), and phase-locked loops (PLLs). The **Flow Summary** indicates whether the design exceeds the available device resources. More detailed information is available by viewing the reports under **Resource Section** in the **Fitter** section of the Compilation Report.

For Arria II GX, Arria GX, Stratix IV, Stratix III, and Stratix II devices, a device with low utilization does not have the lowest adaptive logic module (ALM) utilization possible. For these devices, the Fitter uses adaptive look-up tables (ALUTs) in different ALMs—even when the logic can be placed within one ALM—to achieve the best timing and routability results. In achieving these results, the Fitter can spread

logic throughout the device. As the device fills up, the Fitter automatically searches for logic functions with common inputs to place in one ALM. The number of partnered ALUTs and packed registers also increases. Therefore, a design that is reported as close to 100% full might still have space for extra logic if logic and registers can be packed together more aggressively.

If resource usage is reported as less than 100% and a successful fit cannot be achieved, either there are not enough routing resources or some assignments are illegal. In either case, a message appears in the **Processing** tab of the Messages window describing the problem.

If the Fitter finishes faster than the Fitter runs on similar designs, a resource might be over-utilized or there might be an illegal assignment. If the Quartus II software seems to run for an excessively long time compared to runs on similar designs, a legal placement or route probably cannot be found. Look for errors and warnings that indicate these types of problems.

Refer to "Limit to One Fitting Attempt" on page 10–9 for more information about how to get a quick error message on hard-to-fit designs.

You can use the Chip Planner or the Timing Closure Floorplan (for older devices) to find areas of the device that have routing congestion.

For details about using the Chip Planner and the Timing Closure Floorplan tools, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

I/O Timing (Including t_{PD})

The Quartus II TimeQuest Timing Analyzer supports the Synopsys Design Constraints (SDC) format for constraining your design. When using TimeQuest for timing analysis, use the `set_input_delay` constraint to specify the data arrival time at an input port with respect to a given clock. For output ports, use the `set_output_delay` command with respect to a given clock. You can use the `report_timing` Tcl command to generate the required I/O timing reports.

If you are using the Quartus II Classic Timing Analyzer, from the Compilation Report, select **Timing Analyzer** to determine whether or not I/O timing has been met. The t_{SU}, t_H, and t_{CO} reports list the I/O paths, together with the required timing number if you have specified a timing requirement, the actual timing number for the timing as reported by the Quartus II software, and the slack (the difference between your requirement and the actual number). If you have any point-to-point propagation delay (t_{PD}) assignments, the t_{PD} report lists the corresponding paths.

The I/O paths that do not meet the required timing performance are reported as having negative slack and are displayed in red. In cases when you do not make an explicit I/O timing assignment to an I/O pin, the Quartus II timing analysis software still reports the **Actual** number, which is the timing number that must be met for that timing parameter when the device runs in your system.

To analyze the reasons why your timing requirements were not met, right-click an entry in the report and click **List Paths**. A message listing the paths appears in the **System** tab of the Messages window. To expand a selection, click the "+" icon at the beginning of the line (Figure 10–2). This is a good method to determine where the greatest delay is located along the path.

The List Paths report lists the slack time and how that slack time was calculated. By expanding the various entries, you can see the incremental delay through each node in the path as well as the total delay. The incremental delay is the sum of the interconnect delay (`IC`) and the cell delay (`CELL`) through the logic.

Figure 10–2. I/O Slack Report

```
Info: Slack time is -184 ps for clock "clk" between source pin "newt" and destination register "taps:inst|xn[3]~reg0"
  Info: + tsu requirement for source pin and destination register is 0.615 ns
  Info: - tsu from clock to input pin is 0.799 ns
    Info: + Longest pin to register delay is 3.853 ns
      Info: 1: + IC(0.000 ns) + CELL(1.490 ns) = 1.490 ns; Loc. = PIN_M8; Fanout = 34; PIN Node = 'newt'
      Info: 2: + IC(1.961 ns) + CELL(0.402 ns) = 3.853 ns; Loc. = IOC_X0_Y20_N1; Fanout = 1; REG Node = 'taps:inst|xn[3]~reg0'
      Info: Total cell delay = 1.892 ns ( 49.10 % )
      Info: Total interconnect delay = 1.961 ns ( 50.90 % )
    Info: + Micro setup delay of destination is 0.096 ns
    Info: - Shortest clock path from clock "clk" to destination register is 3.150 ns
System  Processing
```

To analyze I/O timing, right-click on an I/O entry in the report, point to **Locate**, and click **Locate in Chip Planner** or **Timing Closure Floorplan** (for older devices) to highlight the I/O path on the floorplan. Negative slack indicates paths that failed to meet their timing requirements. Other options allow you to see all the intermediate nodes (combinational logic cells) on a path and the delay for each level of logic, or to see the fan-in and fan-out of a selected node.

For more information about how timing numbers are calculated, refer to the *Quartus II Classic Timing Analyzer* chapter or the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Register-to-Register Timing

This section contains the following sections:

- "Timing Analysis with the TimeQuest Timing Analyzer"
- "Tips for Analyzing Failing Paths"
- "Tips for Analyzing Failing Clock Paths that Cross Clock Domains"

Timing Analysis with the TimeQuest Timing Analyzer

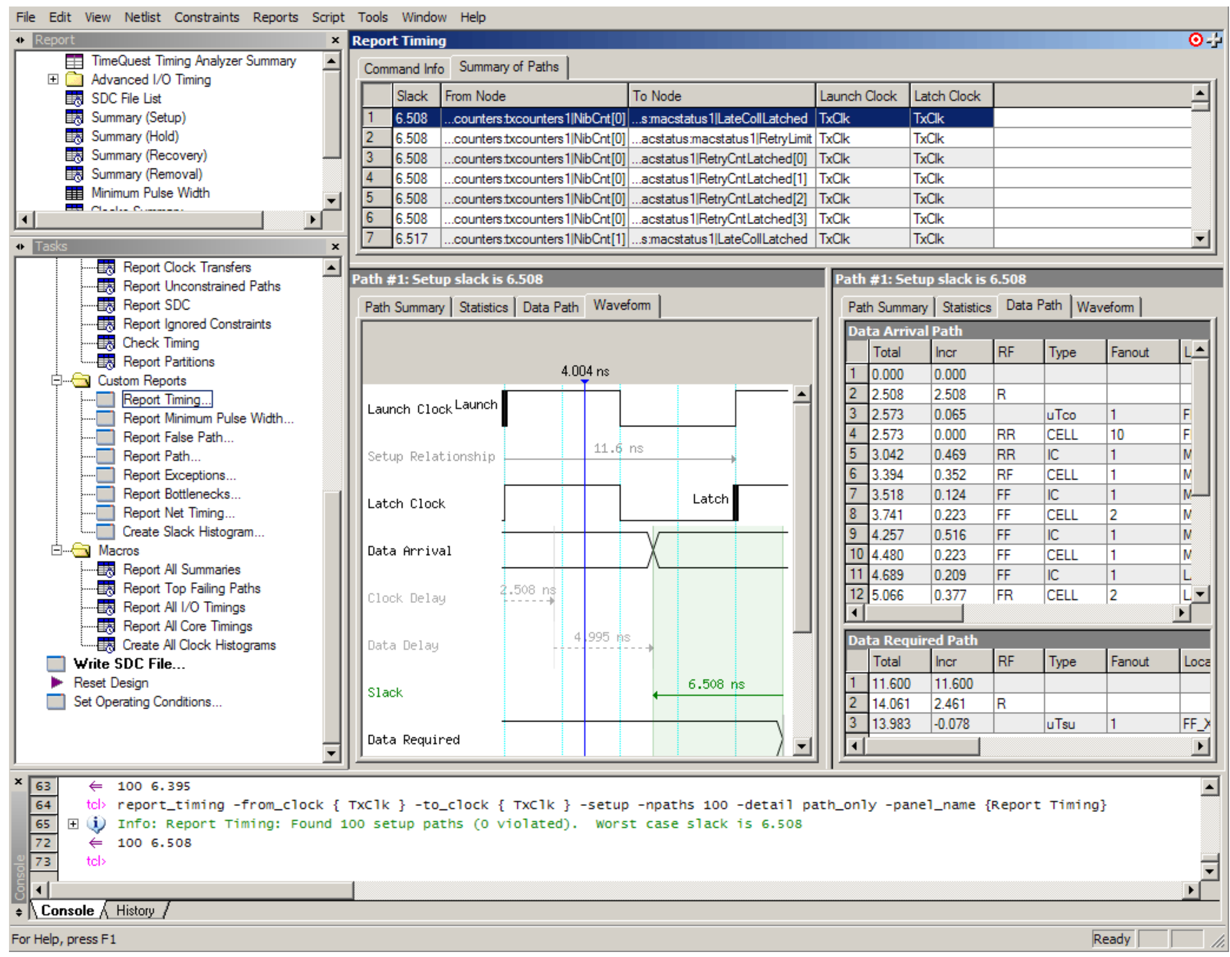
If you are using the TimeQuest Timing Analyzer, you should analyze all valid register-to-register paths by using appropriate constraints. Use the `report_timing` command to generate the required timing reports for any register-to-register path. Your design meets timing requirements when you do not have negative slack on any register-to-register path on any of the clock domains.

All paths that do not meet the timing requirement are shown with a negative slack and appear in red color in the TimeQuest Timing Analyzer GUI.

When you select a path listed in the TimeQuest report timing panel, the corresponding Path Summary that indicates the source and destination registers, the complete data path that indicates various nodes in the path, and the waveforms of relevant signals are displayed (Figure 10–3). You can locate a selected path in the Chip Planner or the Technology Map Viewer by using the right-click menu. You can use this to analyze your failing paths or timing critical paths. Similarly, if you know that a path is not a valid path, you can set it to be a false path using the right-click menu.

To see the path details of any selected path, click on the **Data Path** tab in the path details pane that appears. This displays the details of the Data Arrival Path, as well as the Data Required Path. For a graphical view of the information, click on the **Waveform** tab.

Figure 10–3. TimeQuest Timing Analyzer GUI



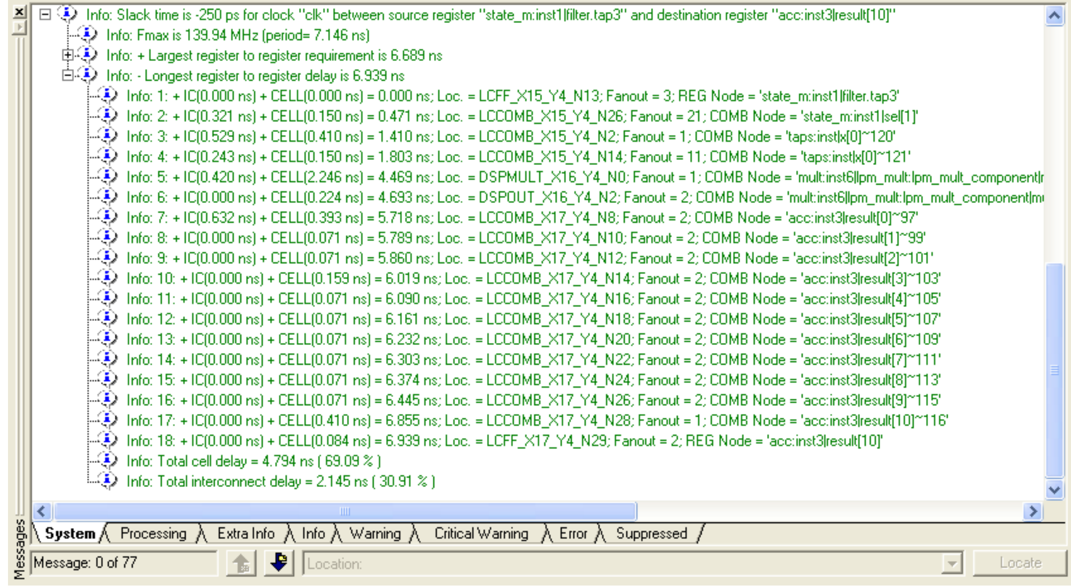
Timing Analysis with the Classic Timing Analyzer

If you are using the Quartus II Classic Timing Analyzer, in the Compilation Report window, use the **Timing Analyzer** section to determine whether register-to-register timing requirements are met. The **Clock Setup** folder displays figures for the actual register-to-register for each clock and the slack delays as reported by the Quartus II software. The paths that do not meet timing requirements are shown with a negative slack and appear in red.

To determine why your timing requirements were not met, right-click on an entry in the report and click **List Paths**. A message listing the paths appears in the **System** tab of the Messages window. The expanded report for the path appears (Figure 10–4). Click the "+" icon at the beginning of the line. Use this method to determine where the greatest delay is located along the path.

The List Paths report shows the slack time and how that slack time was calculated. By expanding the various entries, you can see the incremental delay through each node in the path as well as the total delay. The incremental delay is the sum of the interconnect delay (`IC`) and the cell delay (`CELL`) through the logic.

Figure 10–4. f_{MAX} Slack Report



To visually analyze register-to-register timing paths, right-click on a path, point to **Locate**, and click **Locate in Chip Planner**. For older devices, such as APEX™ series devices, ACEX®, FLEX® 10K, and MAX 7000 devices, click **Locate in Timing Closure Floorplan** to perform this analysis. The Chip Planner or Timing Closure Floorplan appears with the path highlighted. Use the **Critical Path Settings** to select which failing paths to show. To turn critical paths on or off, use the **Show Critical Paths** command.

For more information about how timing analysis results are calculated, refer to the *Quartus II Classic Timing Analyzer* and the *Quartus II TimeQuest Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*.

You also can see the logic in a particular path by cross-probing to the RTL Viewer or Technology Map Viewer. These viewers allow you to see a gate-level or technology-mapped representation of your design netlist. To locate a timing path in one of the viewers, right-click on a path in the report, point to **Locate**, and click **Locate in RTL Viewer** or **Locate in Technology Map Viewer**. When you locate a timing path in the Technology Map Viewer, the annotated schematic displays the same delay information that is shown when you use the **List Paths** command.

For more information about the netlist viewers, refer to the *Analyzing Designs with Quartus II Netlist Viewers* chapter in volume 1 of the *Quartus II Handbook*.

Tips for Analyzing Failing Paths

When you are analyzing clock path failures, focus on improving the paths that show the worst slack. The Fitter works hardest on paths with the least slack. If you fix these paths, the Fitter might be able to improve the other failing timing paths in the design.

Check for particular nodes that appear in many failing paths. Look for paths that have common source registers, destination registers, or common intermediate combinational nodes. In some cases, the registers might not be identical, but are part of the same bus. In the timing analysis report panels, clicking on the **From** or **To** column headings can be helpful to sort the paths by the source or destination registers. Clicking first on **From**, then on **To**, uses the registers in the **To** column as the primary sort and **From** as the secondary sort. If you see common nodes, these nodes indicate areas of your design that might be improved through source code changes or Quartus II optimization settings. Constraining the placement for just one of the paths might decrease the timing performance for other paths by moving the common node further away in the device.

Tips for Analyzing Failing Clock Paths that Cross Clock Domains

When analyzing clock path failures, check whether these paths cross between two clock domains. This is the case if the **From Clock** and **To Clock** in the timing analysis report are different. There can also be paths that involve a different clock in the middle of the path, even if the source and destination register clock are the same. To analyze these paths in more detail, right-click on the entry in the report and click **List Paths**.

Expand the **List Paths** entry in the Messages window and analyze the largest register-to-register requirement. Evaluate the setup relationship between the source and destination (launch edge and latch edge) to determine if that is reducing the available setup time. For example, the path can start at a rising edge and end at a falling edge, which reduces the setup relationship by one half clock cycle.

Check if the PLL phase shift is reducing the setup requirement. You might be able to adjust this using PLL parameters and settings.

Check if the PLL compensation delay is reducing the setup relationship. If you are using the Quartus II Classic Timing Analyzer, you can direct the software to analyze this delay as clock skew by enabling Clock Latency analysis. On the Assignments menu, click **Timing Analysis Settings**. In the **Category** list, select **Classic Timing Analyzer Settings** and click **More Settings**. In the **Name** list, select **Enable Clock Latency**. In the **Setting** list, select **On**. Typically, you must enable this option if your design results in timing violations for paths that pass between PLL clock domains. The Quartus II TimeQuest Timing Analyzer performs this analysis by default.

Paths that cross clock domains are generally protected with synchronization logic (for example, FIFOs or double-data synchronization registers) to allow asynchronous interaction between the two clock domains. In such cases, you can ignore the timing paths between registers in the two clock domains while running timing analysis, even if the clocks are related.

The Fitter attempts to optimize all failing timing paths. If there are paths that can be ignored for optimization and timing analysis, but the paths do not have constraints that instruct the Fitter to ignore them, the Fitter tries to optimize those paths as well. In some cases, optimizing unnecessary paths can prevent the Fitter from meeting the timing requirements on timing paths that are critical to the design. It is beneficial to specify all paths that can be ignored, so that the Fitter can put more effort into the paths that must meet their timing requirements instead of optimizing paths that can be ignored.

For more details about how to ignore timing paths that cross clock domains, refer to the *Quartus II Classic Timing Analyzer* chapter or the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the Quartus II handbook.

Evaluate the clock skew between the source clock and the destination clock to determine if that is reducing the available setup time. You can check the shortest and longest clock path reports to see what is causing the clock skew. Avoid using combinational logic in clock paths because it contributes to clock skew. Differences in the logic or in its routing between the source and destination can cause clock skew problems and result in warnings during compilation.

Global Routing Resources

Global routing resources are designed to distribute high-fan-out, low-skew signals (such as clocks) without consuming regular routing resources. Depending on the device, these resources can span the entire chip, or some smaller portion, such as a quadrant. The Quartus II software attempts to assign signals to global routing resources automatically, but you might be able to make more suitable assignments manually.

Refer to the relevant device handbook for details about the number and types of global routing resources available.

Check the global signal utilization in your design to ensure that appropriate signals have been placed on global routing resources. In the Compilation Report, open the Fitter report and click the **Resource Section**. Analyze the Global & Other Fast Signals and Non-Global High Fan-out Signals reports to determine whether any changes are required.

You might be able to reduce clock skew for high fan-out signals by placing them on global routing resources. Conversely, you can reduce the insertion delay of low fan-out signals by removing them from global routing resources. Doing so can improve clock enable timing and control signal recovery/removal timing, but increases clock skew. You can also use the **Global Signal** setting in the Assignment Editor to control global routing resources.

Compilation Time

In long compilations, most of the time is spent in the Analysis & Synthesis and Fitter modules. Analysis & Synthesis includes synthesis netlist optimizations, if you have turned on that option. The Fitter includes two steps, placement and routing, and also includes physical synthesis if you turned on that option. The **Flow Elapsed Time** section of the Compilation Report shows how much time is spent running the Analysis & Synthesis and Fitter modules. The Fitter Messages report in the **Fitter** section of the Compilation Report shows the specific time that was spent in placement and how much time was spent in routing.

Placement is the process of finding optimum locations for the logic in your design. Routing is the process of connecting the nets between the logic in your design. There are many possible placements for the logic in a design, and finding better placements typically takes more compilation time. Good logic placement allows you to more easily meet your timing requirements and makes the design easier to route.

The applicable messages are indicated as shown in the following example, with each time component in two-digit format:

```
Info: Fitter placement operations ending: elapsed time = <days:hours:mins:secs>
Info: Fitter routing operations ending: elapsed time = <days:hours:mins:secs>
```

Days are not shown if the time is less than one day.

While the Fitter is running (including Placement and Routing), hourly info messages similar to the following message are displayed every hour to indicate Fitter operations are progressing normally.

```
Info: Placement optimizations have been running for x hour(s)
```

In this case, x indicates the number of hours the process has run.

Resource Utilization Optimization Techniques (LUT-Based Devices)

After design analysis, the next stage of design optimization is to improve resource utilization. Complete this stage before proceeding to I/O timing optimization or register-to-register timing optimization. Ensure that you have already set the basic constraints described in "Initial Compilation: Required Settings" on page 10–3 before proceeding with the resource utilization optimizations discussed in this section. If a design does not fit into a specified device, use the techniques in this section to achieve a successful fit. After you optimize resource utilization and your design fits in the desired target device, optimize I/O timing as described in "I/O Timing Optimization Techniques (LUT-Based Devices)" on page 10–73. These tips are valid for all FPGA families and the MAX II family of CPLDs.

Using the Resource Optimization Advisor

The Resource Optimization Advisor provides guidance in determining settings that optimize the resource usage. To run the Resource Optimization Advisor, on the Tools menu, point to **Advisors**, and click **Resource Optimization Advisor**.

The Resource Optimization Advisor provides step-by-step advice about how to optimize the resource usage (logic element, memory block, DSP block, I/O, and routing) of your design. Some of the recommendations in these categories might contradict each other. Altera recommends evaluating the options, and choosing the settings that best suit your requirements.

Resolving Resource Utilization Issues Summary

Resource utilization issues can be divided into the following three categories:

- Issues relating to I/O pin utilization or placement, including dedicated I/O blocks such as PLLs or LVDS transceivers (refer to "I/O Pin Utilization or Placement").
- Issues relating to logic utilization or placement, including logic cells containing registers and look-up tables as well as dedicated logic, such as memory blocks and DSP blocks (refer to "Logic Utilization or Placement" on page 10–21).
- Issues relating to routing (refer to "Routing" on page 10–29).

I/O Pin Utilization or Placement

Use the suggestions in the following sections to help you resolve I/O resource problems.

Use I/O Assignment Analysis

On the Processing menu, point to **Start** and click **Start I/O Assignment Analysis** to help with pin placement. The **Start I/O Assignment Analysis** command allows you to check your I/O assignments early in the design process. You can use this command to check the legality of pin assignments before, during, or after compilation of your design. If design files are available, you can use this command to accomplish more thorough legality checks on your design's I/O pins and surrounding logic. These checks include proper reference voltage pin usage, valid pin location assignments, and acceptable mixed I/O standards.

Common issues with I/O placement relate to the fact that differential standards have specific pin pairings, and certain I/O standards might be supported only on certain I/O banks.

If your compilation or I/O assignment analysis results in specific errors relating to I/O pins, follow the recommendations in the error message. Right-click on the message in the Messages window and click **Help** to open the Quartus II Help topic for this message.

Modify Pin Assignments or Choose a Larger Package

If a design that has pin assignments fails to fit, compile the design without the pin assignments to determine whether a fit is possible for the design in the specified device and package. You can use this approach if a Quartus II error message indicates fitting problems due to pin assignments.

If the design fits when all pin assignments are ignored or when several pin assignments are ignored or moved, you might have to modify the pin assignments for the design or choose a larger package.

If the design fails to fit because insufficient I/Os are available, a successful fit can often be obtained by using a larger device package (which can be the same device density) that has more available user I/O pins.

For more information about I/O assignment analysis, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Logic Utilization or Placement

Use the suggestions in the following subsections to help you resolve logic resource problems, including logic cells containing registers and lookup tables (LUTs), as well as dedicated logic such as memory blocks and DSP blocks.

Optimize Synthesis for Area, Not Speed

If your design fails to fit because it uses too much logic, resynthesize the design to improve the area utilization. First, ensure that you have set your device and timing constraints correctly in your synthesis tool. Particularly when the area utilization of the design is a concern, ensure that you do not over-constrain the timing requirements for the design. Synthesis tools generally try to meet the specified requirements, which can result in higher device resource usage if the constraints are too aggressive.

If resource utilization is an important concern, some synthesis tools offer an easy way to optimize for area instead of speed. If you are using Quartus II integrated synthesis, choose **Balanced** or **Area** for the **Optimization Technique**. You can also specify this logic option for specific modules in your design with the Assignment Editor in cases where you want to reduce area using the **Area** setting (potentially at the expense of register-to-register timing performance) while leaving the default **Optimization Technique** setting at **Balanced** (for the best trade-off between area and speed for certain device families) or **Speed**. You can also use the **Speed Optimization Technique for Clock Domains** logic option to specify that all combinational logic in or between the specified clock domain(s) is optimized for speed.

In some synthesis tools, not specifying an f_{MAX} requirement can result in less resource utilization.

In the Quartus II software, the **Balanced** setting typically produces utilization results that are very similar to those produced by the **Area** setting, with better performance results. The **Area** setting can give better results in some unusual cases.

For information about setting timing requirements and synthesis options in Quartus II integrated synthesis and other synthesis tools, refer to the appropriate chapter in *Section III. Synthesis* in volume 1 of the *Quartus II Handbook*, or your synthesis software’s documentation.

The Quartus II software provides additional attributes and options that can help improve the quality of your synthesis results.

Restructure Multiplexers

Multiplexers form a large portion of the logic utilization in many FPGA designs. By optimizing your multiplexed logic, you can achieve a more efficient implementation in your Altera device.

The Quartus II software provides the **Restructure Multiplexers** logic option, which can extract and optimize buses of multiplexers during synthesis. This option is available on the **Analysis & Synthesis Settings** page of the **Settings** dialog box and is useful if your design contains buses of fragmented multiplexers. This option restructures multiplexers more efficiently for area, allowing the design to implement multiplexers with a reduced number of logic elements (LEs) or ALMs. Using the **Restructure Multiplexers** logic option can reduce your design's register-to-register timing performance. This option is turned on automatically when you set the Quartus II Analysis & Synthesis **Optimization Technique** option to **Area** or **Balanced**. To change the default setting, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and click the appropriate option from the **Restructure Multiplexers** list to set the option globally.

For design guidelines to achieve optimal resource utilization for multiplexer designs, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. For more information about the **Restructure Multiplexers** option in the Quartus II software, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Perform WYSIWYG Resynthesis with Balanced or Area Setting

If you use another EDA synthesis tool and want to determine if the Quartus II software can remap the circuit to use fewer LEs or ALMs, follow these steps:

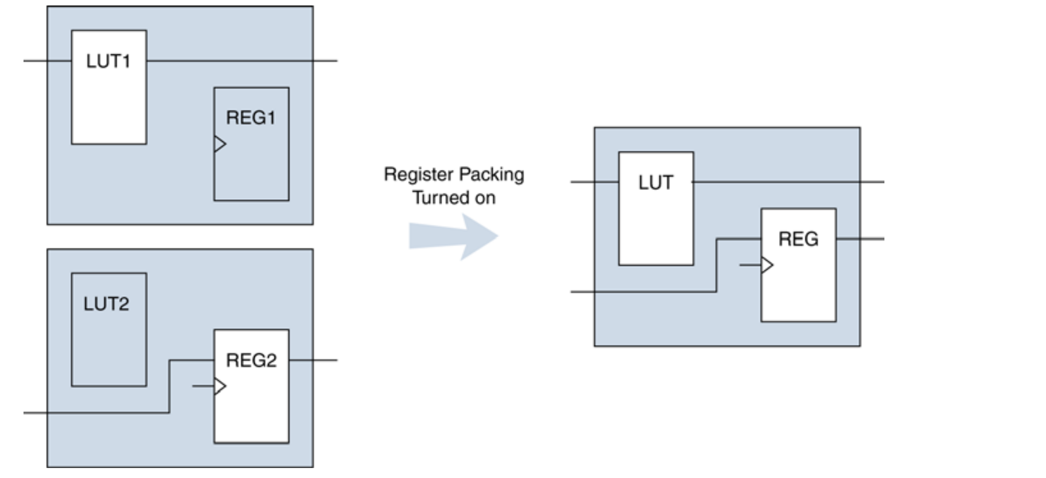
1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Turn on **Perform WYSIWYG primitive resynthesis (using optimization techniques specified in Analysis & Synthesis settings)**. Or, on the Assignments menu, click **Assignment Editor**, and set the **Perform WYSIWYG Primitive Resynthesis** logic option for a specific module in your design.
4. On the same page, choose **Balanced** or **Area** under **Optimization Technique**. Or, on the Assignments menu, click **Assignment Editor**. Set the **Optimization Technique** to **Balanced** or **Area** for a specific module in your design.
5. Recompile the design.

The **Balanced** setting typically produces utilization results that are very similar to the **Area** setting with better performance results. The **Area** setting can give better results in some unusual cases. Performing WYSIWYG resynthesis for area in this way typically reduces register-to-register timing performance.

Use Register Packing

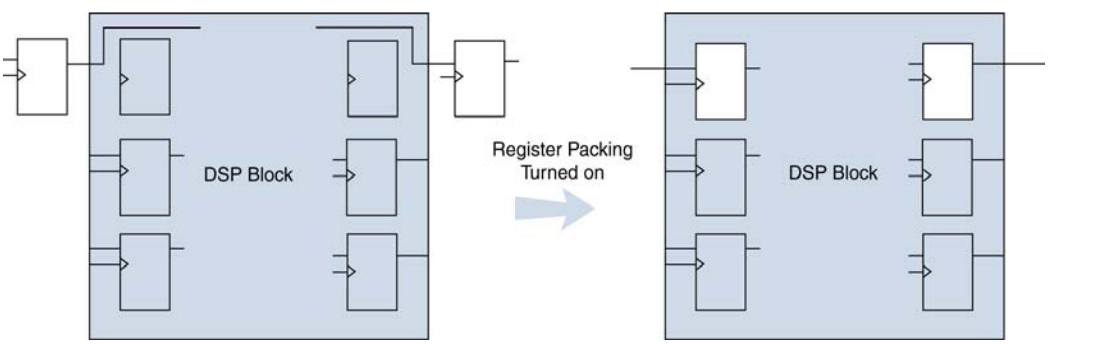
The **Auto Packed Registers** option implements the functions of two cells into one logic cell by combining the register of one cell in which only the register is used with the LUT of another cell in which only the LUT is used. Figure 10–5 shows register packing and the gain of one logic cell in the design.

Figure 10–5. Register Packing



Registers can also be packed into DSP blocks (Figure 10–6).

Figure 10–6. Register Packing in DSP Blocks



The following list shows the most common cases in which register packing helps to optimize a design:

- A LUT can be implemented in the same cell as an unrelated register with a single data input
- A LUT can be implemented in the same cell as the register that is fed by the LUT
- A LUT can be implemented in the same cell as the register that feeds the LUT
- A register can be packed into a RAM block
- A register can be packed into a DSP block
- A register can be packed into an I/O Element (IOE)

The following options are available for register packing (for certain device families):

- **Off**—Does not pack registers
- **Normal**—Packs registers when this is not expected to hurt timing results
- **Minimize Area**—Aggressively packs registers to reduce area

- **Minimize Area with Chains**—Aggressively packs registers to reduce area. This option packs registers with carry chains. It also converts registers into register cascade chains and packs them with other logic to reduce area. This option is available only for Arria GX, Stratix, Cyclone, and MAX II series of devices.
- **Auto**—This is the default setting for register packing. This setting tells the Fitter to attempt to achieve the best performance while maintaining a fit for the design in the specified device. The Fitter combines all combinational (LUT) and sequential (register) functions that benefit circuit speed. In addition, more aggressive combinations of unrelated combinational and sequential functions are performed to the extent required to reduce the area of the design to achieve a fit in the specified device. This option is available only for Arria GX, Stratix, and Cyclone series of devices.
- **Sparse**—In this mode, the combinational (LUT) and sequential (register) functions are combined such that the combined logic has either a combinational output or a sequential output, but not both. This mode is available only for Arria II GX, Arria GX, Stratix III, Stratix II, Cyclone III, and Cyclone II devices. This option results in a higher logic array block (LAB) usage, but might give you better timing performance because of reduced routing congestion.
- **Sparse Auto**—In this mode, the Quartus II Fitter starts with sparse mode packing, and then attempts to achieve best performance while maintaining a fit for the specified device. Later optimizations are carried out in a way similar to the **Auto** mode. This mode is available only for Arria II GX, Arria GX, Stratix IV, Stratix III, Stratix II, Cyclone III, and Cyclone II devices.

Turning on register packing decreases the number of LEs or ALMs in the design, but could also decrease performance in some cases. On the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**, and then click **More Settings. Turn on Auto Packed Registers** to turn on register packing.

The area reduction and performance results with register packing can vary greatly depending on the design.

The **Auto** setting performs more aggressive register packing as required, so the typical results vary depending on the device resource utilization.

Remove Fitter Constraints

A design with conflicting constraints or constraints that are difficult to meet might not fit in the targeted device. This can occur when the location or LogicLock assignments are too strict and not enough routing resources are available on the device.

In this case, use the **Routing Congestion** view in the Chip Planner or Timing Closure Floorplan (for older devices) to locate routing problems in the floorplan, then remove any location or LogicLock region assignments in that area. If your design still does not fit, the design is over-constrained. To correct the problem, remove all location and LogicLock assignments and run successive compilations, incrementally constraining the design before each compilation. You can delete specific location assignments in the Assignment Editor, the Chip Planner, or the Timing Closure Floorplan. To remove LogicLock assignments in the Chip Planner (or in the Timing Closure Floorplan), in the LogicLock Regions Window, or on the Assignments menu, click **Remove Assignments.** Turn on the assignment categories you want to remove from the design in the **Available assignment categories** list.

For more information about the **Routing Congestion** view in the Chip Planner and the Timing Closure Floorplan, refer to *Analyzing and Optimizing the Design Floorplan* in volume 2 of the *Quartus II Handbook*. Also refer to the Quartus II Help.

Change State Machine Encoding

State machines can be encoded using various techniques. Using binary or gray code encoding typically results in fewer state registers than one-hot encoding, which requires one register for every state bit. If your design contains state machines, changing the state machine encoding to one that uses the minimal number of registers might reduce resource utilization. The effect of state machine encoding varies depending on the way your design is structured.

If your design does not manually encode the state bits, you can specify the state machine encoding in your synthesis tool. When using Quartus II integrated synthesis, on the Assignments menu, click **Settings**. In the **Category** list, select A**nalysis & Synthesis Settings** and turn on **Minimal Bits for State Machine Processing**. You also can specify this logic option for specific modules or state machines in your design with the Assignment Editor.

You can also use the following Tcl command in scripts to modify the state machine encoding.

```
set_global_assignment -name state_machine_processing <value>
```

In this case, <*value*> can be `AUTO`, `ONE-HOT`, `MINIMAL BITS`, or `USER-ENCODE`.

Flatten the Hierarchy During Synthesis

Synthesis tools typically provide the option of preserving hierarchical boundaries, which can be useful for verification or other purposes. However, optimizing across hierarchical boundaries allows the synthesis tool to perform the most logic minimization, which can reduce area. Therefore, to achieve the best results, flatten your design hierarchy whenever possible. If you are using Quartus II integrated synthesis, ensure that the **Preserve Hierarchical Boundary** logic option is turned off, that is, make sure that you have not turned on the option in the Assignment Editor or with Tcl assignments. If you are using Quartus II incremental compilation, you cannot flatten your design across design partitions. Incremental compilation always preserves the hierarchical boundaries between design partitions. Follow Altera's recommendations for design partitioning, such as registering partition boundaries to reduce the effect of cross-boundary optimizations.

For more information about using incremental compilation and recommendations for design partitioning, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Retarget Memory Blocks

If the design fails to fit because it runs out of device memory resources, your design might require a certain type of memory the device does not have. For example, a design that requires two M-RAM blocks can be targeted to a Stratix EP1S10 device, which has only one M-RAM block. You might be able to obtain a fit by building one of the memories with a different size memory block, such as an M4K memory block.

If the memory block was created with the MegaWizard™ Plug-In Manager, open the MegaWizard Plug-In Manager and edit the RAM block type so it targets a new memory block size.

ROM and RAM memory blocks can also be inferred from your HDL code, and your synthesis software can place large shift registers into memory blocks by inferring the ALTSHIFT_TAPS megafunction. This inference can be turned off in your synthesis tool to cause the memory to be placed in logic instead of in memory blocks. To disable inference when using Quartus II integrated synthesis, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis**. The **Analysis & Synthesis** page appears.
3. Turn off the **Auto RAM Replacement**, **Auto ROM Replacement**, or **Auto Shift Register Replacement** logic option as appropriate for your project. Or, disable the option for a specific entity in the Assignment Editor.

Depending on your synthesis tool, you can also set the RAM block type for inferred memory blocks. In Quartus II integrated synthesis, set the **ramstyle** attribute to the desired memory type for the inferred RAM blocks, or set the option to **logic** to implement the memory block in standard logic instead of a memory block.

Consider the resource utilization by hierarchy in the report file, and determine whether there is an unusually high register count in any of the modules. Some coding styles can prevent the Quartus II software from inferring RAM blocks from the source code because of their architectural implementation and forces the software to implement the logic in flipflops. As an example, a function such as an asynchronous reset on a register bank might make it incompatible with the RAM blocks in the device architecture, so that the register bank is implemented in flipflops. It is often possible to move a large register bank into RAM by slight modification of associated logic.

For more information about memory inference control in other synthesis tools, refer to the appropriate chapter in *Section III. Synthesis* in volume 1 of the *Quartus II Handbook*, or your synthesis software's documentation. For more information about coding styles and HDL examples that ensure memory inference, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Use Physical Synthesis Options to Reduce Area

The physical synthesis options for fitting can help you decrease the resource usage; additional optimizations for fitting are available. When you enable these settings for physical synthesis for fitting, the Quartus II software makes placement-specific changes to the netlist that reduce resource utilization for a specific Altera device.

The compilation time might increase considerably when you use physical synthesis options.

With the Quartus II software, you can apply physical synthesis options to specific instances, which can reduce the impact on compilation time. Physical synthesis instance assignments allow you to enable physical synthesis algorithms for specific portions of their design.

If you want the performance gain from physical synthesis, but do not want a specific hierarchy of the design to be modified, you can selectively disable physical synthesis for that hierarchy. Likewise, if you do not want to run physical synthesis for most parts of the design, but require the algorithms for a specific module in the design, you can enable physical synthesis for a single module.

The following physical synthesis optimizations for fitting are available:

- Physical synthesis for combinational logic
- Map logic into memory

On the Assignments menu, click **Settings**. In the **Category** list, expand **Compilation Process Settings** and select **Physical Synthesis Optimization**. The **Physical Synthesis Optimization** page appears. Under **Optimize for fitting**, turn on the options to enable physical synthesis optimizations during fitting. You can also specify the physical synthesis effort, which sets the level of physical synthesis optimization that you want the Quartus II software to perform.

The **Perform physical synthesis for combinational logic** option allows the Quartus II Fitter to resynthesize the combinational logic in a design to reduce the resource utilization to help achieve a fit.

The **Perform logic to memory mapping** option allows the Quartus II Fitter to automatically map logic into unused memory blocks during fitting, reducing the number of logic elements required to implement the design.

To apply physical synthesis assignments for fitting on a per instance basis, use the Quartus II Assignment Editor. The following assignments are available as instance assignments for fitting:

- **Perform physical synthesis for combinational logic**
- **Perform logic to memory mapping**

In the Assignment Editor, indicate the module instance you want to apply the setting in the **To** tab. Select the required physical synthesis assignment in the Assignment **Name** tab. In the **Value** tab, select **ON**. In the **Enabled** tab, select **Yes**.

Retarget or Balance DSP Blocks

A design might not fit because it requires too many DSP blocks. All DSP block functions can be implemented with logic cells, so you can retarget some of the DSP blocks to logic to obtain a fit.

If the DSP function was created with the MegaWizard Plug-In Manager, open the MegaWizard Plug-In Manager and edit the function so it targets logic cells instead of DSP blocks. The Quartus II software uses the `DEDICATED_MULTIPLIER_CIRCUITRY` megafunction parameter to control the implementation.

DSP blocks also can be inferred from your HDL code for multipliers, multiply-adders, and multiply-accumulators. This inference can be turned off in your synthesis tool. When you are using Quartus II integrated synthesis, you can disable inference by turning off the **Auto DSP Block Replacement** logic option for your whole project. On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and turn off **Auto DSP Block Replacement**. Alternatively, you can disable the option for a specific block with the Assignment Editor.

For more information about disabling DSP block inference in other synthesis tools, refer to the appropriate chapter in *Section III. Synthesis* in volume 1 of the *Quartus II Handbook*, or your synthesis software's documentation.

The Quartus II software also offers the **DSP Block Balancing** logic option, which implements DSP block elements in logic cells or in different DSP block modes. The default **Auto** setting allows DSP block balancing to convert the DSP block slices automatically as appropriate to minimize the area and maximize the speed of the design. You can use other settings for a specific node or entity, or on a project-wide basis, to control how the Quartus II software converts DSP functions into logic cells and DSP blocks. Using any value other than **Auto** or **Off** overrides the `DEDICATED_MULTIPLIER_CIRCUITRY` parameter used in megafunction variations.

For more details about the Quartus II logic options described in this section, refer to the Quartus II Help.

Optimize Source Code

If your design does not fit because of logic utilization, and the methods described in the preceding sections do not sufficiently improve the resource utilization of the design, modify the design at the source to achieve the desired results. You can often improve logic significantly by making design-specific changes to your source code. This is typically the most effective technique for improving the quality of your results.

If your design does not fit into available LEs or ALMs, but you have unused memory or DSP blocks, check to see if you have code blocks in your design that describe memory or DSP functions that are not being inferred and placed in dedicated logic. You might be able to modify your source code to allow these functions to be placed into dedicated memory or DSP resources in the target device.

Ensure that your state machines are recognized as state machine logic and optimized appropriately in your synthesis tool. State machines that are recognized are generally optimized better than if the synthesis tool treats them as generic logic. In the Quartus II software, you can check for the State Machine report under **Analysis & Synthesis** in the Compilation Report. This report provides details, including the state encoding for each state machine that was recognized during compilation. If your state machine is not being recognized, you might have to change your source code to enable it to be recognized.

For coding style guidelines, including examples of HDL code for inferring memory and DSP functions, refer to the "Instantiating Altera Megafunctions" and the "Inferring Altera Megafunctions" sections of the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. For guidelines and sample HDL code for state machines, refer to the "General Coding Guidelines" section of the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Use a Larger Device

If a successful fit cannot be achieved because of a shortage of LEs or ALMs, memory, or DSP blocks, you might require a larger device.

Routing

Use the suggestions in the following subsections to help you resolve routing resource problems.

Set Auto Register Packing to Sparse or Sparse Auto

This option is useful for reducing LE or ALM count in a design. This option is available for Arria GX, Cyclone, and Stratix series of devices. On the Assignments menu, click **Settings**. The **Settings** dialog box appears. In the **Category** list, select **Fitter Settings**. Click **More Settings**. Under **Option**, in the **Name** list, select **Auto Packed Registers**. In the **Settings** list, select the **Sparse** or **Sparse Auto** from the list.

When you select **Sparse**, the Fitter combines functions to improve the performance of many designs. When you select **Sparse Auto**, the Fitter attempts to achieve the highest performance with the possibility of increasing the area, but without exceeding the logic capacity of the device. These options might help improve the routing because they do not aggressively pack registers.

Selecting the default **Auto** setting can help routing in many designs. However, for some dense designs, the Fitter attempts to combine additional logic to reduce the area of the design to achieve the best performance. It does this by fitting the design within the best area of the selected device. Therefore, the Fitter can turn on the more aggressive minimize area with the chains option, making it more difficult to route the design.

As an alternative, select **Normal**, and then increase the aggressiveness of register packing to reduce LE/ALM count if the design does not fit.

When you choose a register packing setting to perform more aggressive register packing than the **Auto** setting, the extra register packing can affect the routability of the design as an unintended result. The **Minimize the area with chains** setting restricts placement and reduces routability significantly more than using the **Minimize Area** setting. For more information about register packing, refer to "Use Register Packing" on page 10–22.

Set Fitter Aggressive Routability Optimizations to Always

If routing resources are resulting in no-fit errors, use this option to reduce routing wire utilization. On the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**. Click **More Settings**. In the **More Fitter Settings** dialog box, set **Fitter Aggressive Routability Optimizations** to **Always** and click **OK**.

If there is a significant imbalance between placement and routing time (during the first fitting attempt), it might be because of high wire utilization. By turning on this option, you might be able to reduce your compilation time.

On average, in Arria GX and Stratix II devices, this option saves approximately 3% wire utilization but can reduce performance by approximately 1%. In Stratix III devices, this option saves approximately 6% wire utilization, at the same time reducing the performance by approximately 3%. In Cyclone III devices, using this option saves approximately 4.5% wire utilization while reducing the performance by about 4%.

These optimizations are used automatically when the Fitter performs more than one fitting attempt, but turning the option on increases the optimization effort on the first fitting attempt. This option also ensures that the Quartus II software uses maximum optimization to reduce routability, even if the Fitter Effort is set to **Auto Fit**.

Increase Placement Effort Multiplier

Increasing the placement effort can improve the routability of the design, allowing the software to route a design that otherwise requires too many routing resources. On the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**. Click **More Settings**. In the **More Fitter Settings** dialog box, increase the value of the **Placement Effort Multiplier** to increase placement effort. The default value is 1.0. Legal values must be greater than 0 and can be non-integer values. Numbers less than 1 reduce the placement effort and might affect placement quality. Higher numbers increase compilation time but can improve placement quality. For example, a value of 4 increases fitting time by approximately 2 to 4 times but can increase the quality of results. Increasing the placement effort multiplier does not tend to improve timing optimization unless the design also has very high routing resource usage.

Increased effort is used automatically when the Fitter performs more than one fitting attempt. Setting a multiplier higher than one (before compilation) increases the optimization effort on the first fitting attempt. The second and third fitting loops increase the Placement Effort Multiplier to 4 and then to 16. These loops result in increased compilation times, with possible improvement in the quality of placement.

You can modify the Placement Effort Multiplier using the following Tcl command:

```
set_global_assignment -name PLACEMENT_EFFORT_MULTIPLIER <value> ↵
```

<value> can be any positive, non-zero number.

Increase Router Effort Multiplier

The Router Effort Multiplier controls how quickly the router tries to find a valid solution. The default value is 1.0 and legal values must be greater than 0. Numbers higher than 1 (as high as 3 is generally reasonable) can improve routing quality at the expense of run-time on difficult-to-route circuits. Numbers closer to 0 (for example, 0.1) can reduce router runtime, but usually reduce routing quality slightly. Experimental evidence shows that a multiplier of 3.0 reduces overall wire usage by about 2%. There is usually no gain in performance beyond a multiplier value of 3.

You can set the Router Effort Multiplier to a value higher than the default value for difficult-to-route designs. To set the Router Effort Multiplier, on the Assignments menu, click **Settings**, and then click **Fitter Settings**. Click the **More Settings** button. From the options available, select **Router Effort Multiplier** and edit the value in the dialog box that appears.

You can modify the Router Effort Multiplier by the following Tcl command. *<value>* can be any positive, non-zero number.

```
set_global_assignment -name ROUTER_EFFORT_MULTIPLIER  <value> ↵
```

Remove Fitter Constraints

A design with conflicting constraints or constraints that are difficult to meet may not fit the targeted device. This can occur when location or LogicLock assignments are too strict and there are not enough routing resources.

In this case, use the **Routing Congestion** view in the Chip Planner or the Timing Closure Floorplan (for older devices) to locate routing problems in the floorplan, then remove all location and LogicLock region assignments from that area. If your design still does not fit, the design is over-constrained. To correct the problem, remove all location and LogicLock assignments and run successive compilations, incrementally constraining the design before each compilation. You can delete specific location assignments in the Assignment Editor, the Chip Planner, or Timing Closure Floorplan (for supported devices). Remove LogicLock assignments in the Chip Planner (or in the Timing Closure Floorplan), in the LogicLock Regions Window, or on the Assignments menu, click **Remove Assignments**. Turn on the assignment categories you want to remove from the design in the **Available assignment categories** list.

For more information about the **Routing Congestion** view in the Chip Planner or the Timing Closure Floorplan, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*. You can also refer to the Quartus II Help.

Optimize Synthesis for Area, Not Speed

In some cases, resynthesizing the design to improve the area utilization can also improve the routability of the design. First, ensure that you have set your device and timing constraints correctly in your synthesis tool. Ensure that you do not over-constrain the timing requirements for the design, particularly when the area utilization of the design is a concern. Synthesis tools generally try to meet the specified requirements, which can result in higher device resource usage if the constraints are too aggressive.

If resource utilization is important to improving the routing results in your design, some synthesis tools offer an easy way to optimize for area instead of speed. If you are using Quartus II integrated synthesis, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and choose **Balanced** or **Area** under **Optimization Technique**.

You can also specify this logic option for specific modules in your design with the Assignment Editor in cases where you want to reduce area using the **Area** setting (potentially at the expense of register-to-register timing performance). You can apply the setting to specific modules while leaving the default **Optimization Technique** setting at **Balanced** (for the best trade-off between area and speed for certain device families) or **Speed**. You can also use the **Speed Optimization Technique for Clock Domains** logic option to specify that all combinational logic in or between the specified clock domain(s) is optimized for speed.

In the Quartus II software, the **Balanced** setting typically produces utilization results that are very similar to those obtained with the **Area** setting, with better performance results. The **Area** setting can yield better results in some unusual cases.

In some synthesis tools, not specifying an f_{MAX} requirement can result in less resource utilization, which can improve routability.

For information about setting timing requirements and synthesis options in Quartus II integrated synthesis and other synthesis tools, refer to the appropriate chapter in *Section III. Synthesis* in volume 1 of the *Quartus II Handbook*, or your synthesis software's documentation.

Optimize Source Code

If your design does not fit because of routing problems and the methods described in the preceding sections do not sufficiently improve the routability of the design, modify the design at the source to achieve the desired results. You can often improve results significantly by making design-specific changes to your source code, such as duplicating logic or changing the connections between blocks that require significant routing resources.

Use a Larger Device

If a successful fit cannot be achieved because of a shortage of routing resources, you might require a larger device.

Timing Optimization Techniques (LUT-Based Devices)

This section contains guidelines if your design does not meet its timing requirements.

Timing Optimization Advisor

The Timing Optimization Advisor guides you in making settings that optimize your design to meet your timing requirements. To run the Timing Optimization Advisor, on the Tools menu, point to **Advisors**, and click on **Timing Optimization Advisor**. This advisor describes many of the suggestions made in this section.

When you open the Timing Optimization Advisor after compilation, you find recommendations to improve the timing performance of your design. Some of the recommendations in these advisors can contradict each other. Altera recommends evaluating these options and choosing the settings that best suit the given requirements.

Metastability Analysis and Optimization Techniques

Metastability problems can occur when a signal is transferred between circuitry in unrelated or asynchronous clock domains, because the designer cannot guarantee that the signal will meet its setup and hold time requirements. The mean time between failure (MTBF) is an estimate of the average time between instances when metastability could cause a design failure.

For more information about metastability and MTBF, refer to the *Understanding Metastability in FPGAs* white paper.

You can use the Quartus II software to analyze the average MTBF due to metastability when a design synchronizes asynchronous signals, and optimize the design to improve the MTBF. These metastability features are supported only for designs constrained with the TimeQuest Timing Analyzer, and for select device families.

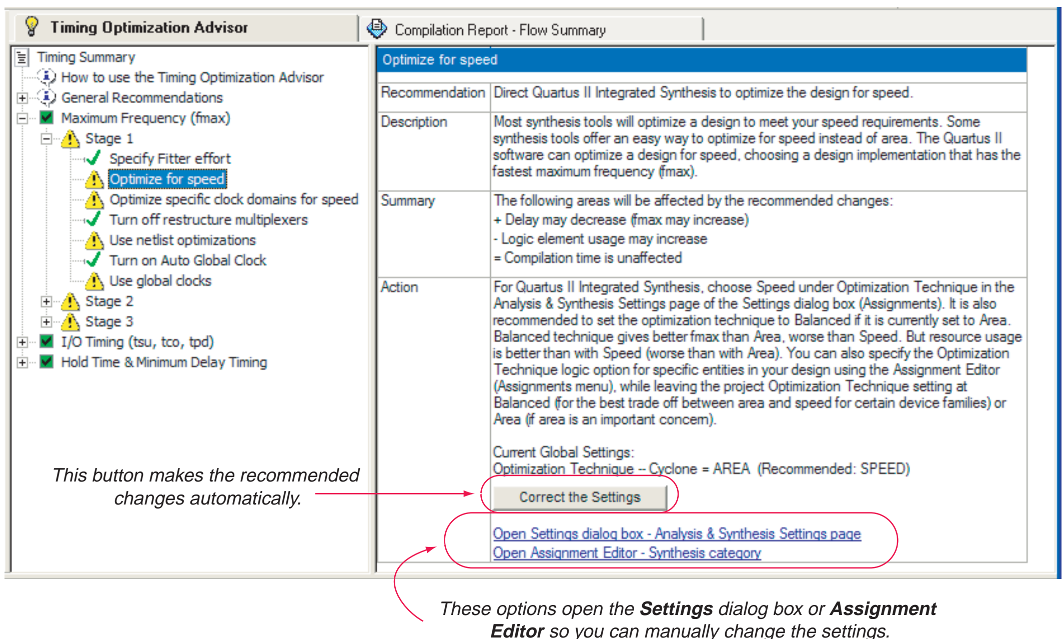
If the MTBF of your design is low, refer to the Metastability Optimization section in the Timing Optimization Advisor, which suggests various settings that can help optimize your design in terms of metastability.

For details about the metastability features in the Quartus II software, refer to the *Managing Metastability with the Quartus II Software* chapter in the *Quartus II Handbook*. This chapter describes how to enable metastability analysis and identify the register synchronization chains in your design, provides details about metastability reports, and provides additional guidelines for managing metastability.

I/O Timing Optimization

The example in Figure 10–7 shows the Timing Optimization Advisor after compiling a design that meets its frequency requirements, but requires setting changes to improve the timing.

Figure 10–7. Timing Optimization Advisor



When you expand one of the categories in the Advisor, such as **Maximum Frequency (fmax)** or **I/O Timing (tsu, tco, tpd)**, the recommendations are divided into stages. The stages show the order in which you should apply the recommended settings. The first stage contains the options that are easiest to change, make the least drastic changes to your design optimization, and have the least effect on compilation time. Icons indicate whether each recommended setting has been made in the current project. In Figure 10–7, the checkmark icons in the list of recommendations for Stage 1 indicate recommendations that are already implemented. The warning icons indicate recommendations that are not followed for this compilation. The information icons indicate general suggestions. For these entries, the advisor does not report whether these recommendations were followed, but instead explains how you can achieve better performance. Refer to the “How to use” page in the Advisor for a legend that provides more information for each icon.

There is a link from each recommendation to the appropriate location in the Quartus II user interface where you can change the settings. For example, consider the **Synthesis Netlist Optimizations** page of the **Settings** dialog box or the **Global Signals category** in the Assignment Editor. This approach provides the most control over which settings are made and helps you learn about the settings in the software. In some cases, you can also use the **Correct the Settings** button to automatically make the suggested change to global settings.

For some entries in the advisor, a button appears that allows you to further analyze your design and gives you more information. The advisor provides a table with the clocks in the design and indicates whether they have been assigned a timing constraint.

The next stage of design optimization focuses on I/O timing. Ensure that you have made the appropriate assignments as described in "Initial Compilation: Required Settings" on page 10–3, and that the resource utilization is satisfactory before proceeding with I/O timing optimization. The suggestions given in this section are applicable to all Altera FPGA families and to the MAX II family of CPLDs.

Because changes to the I/O paths affect the internal register-to-register timing, complete this stage before proceeding to the register-to-register timing optimization stage as described in the "Register-to-Register Timing Optimization Techniques (LUT-Based Devices)" on page 10–40.

The options presented in this section address how to improve I/O timing, including the setup delay (t_{SU}), hold time (t_H), and clock-to-output (t_{CO}) parameters.

Improving Setup and Clock-to-Output Times Summary

Table 10–1 shows the recommended order in which to use techniques to reduce t_{SU} and t_{CO} times. Check marks indicate which timing parameters are affected by each technique. Reducing t_{SU} times increases hold (t_H) times.

Table 10–1. Improving Setup and Clock-to-Output Times *(Note 1)* (Part 1 of 2)

Technique	Affects t_{SU}	Affects t_{CO}
Ensure that the appropriate constraints are set for the failing I/Os (page 10–3)	✓	✓
Use timing-driven compilation for I/O (page 10–36)	✓	✓
Use fast input register (page 10–36)	✓	—
Use fast output register, fast output enable register, and fast OCT register (page 10–36)	—	✓
Decrease the value of **Input Delay from Pin to Input Register** or set **Decrease Input Delay to Input Register = ON** (page 10–37)	✓	—
Decrease the value of **Input Delay from Pin to Internal Cells**, or set **Decrease Input Delay to Internal Cells = ON** (page 10–37)	✓	—
Decrease the value of **Delay from Output Register to Output Pin**, or set **Increase Delay to Output Pin = OFF** (page 10–37)	—	✓
Increase the value of **Input Delay from Dual-Purpose Clock Pin to Fan-Out Destinations** (page 10–39)	✓	—
Use PLLs to shift clock edges (page 10–39)	✓	✓
Use the Fast Regional Clock (page 10–40)	—	✓
For MAX II series of devices, set **Guarantee I/O paths** to zero, **Hold Time at Fast Timing Corner** to **OFF**, or when t_{SU} and t_{PD} constraints permit (page 10–40)	✓	—

Table 10–1. Improving Setup and Clock-to-Output Times *(Note 1)* (Part 2 of 2)

Technique	Affects t_{SU}	Affects t_{CO}
Increase the value of **Delay to output enable pin** or set **Increase delay to output enable pin** (page 10–39)	—	✓

Note to Table 10–1:

(1) These options may not apply to all device families.

Timing-Driven Compilation

To perform IOC timing optimization using the **Optimize IOC Register Placement For Timing** option, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Fitter Settings** and click **More Settings**.
3. In the **More Fitter Settings** dialog box, under **Existing options settings**, select **Optimize IOC Register Placement for Timing.**

This option moves registers into I/O elements if required to meet t_{SU} or t_{CO} assignments, duplicating the register if necessary (as in the case in which a register fans out to multiple output locations). This option is turned on by default and is a global setting. The option does not apply to MAX II series of devices because they do not contain I/O registers.

For APEX 20KE and APEX 20KC devices, if the I/O register is not available, the Fitter tries to move the register into the LAB adjacent to the I/O element.

The **Optimize IOC Register Placement for Timing** option affects only pins that have a t_{SU} or t_{CO} requirement. Using the I/O register is possible only if the register directly feeds a pin or is fed directly by a pin. This setting does not affect registers with any of the following characteristics:

- Have combinational logic between the register and the pin
- Are part of a carry or cascade chain
- Have an overriding location assignment
- Use the synchronous load or asynchronous clear ports of APEX 20K and APEX II devices
- Are input registers that use the synchronous load port and the value is not 1 (in device families where the port is available, other than APEX 20K, APEX II, and FLEX 6000 devices)
- Use the asynchronous load port and the value is not 1 (in device families where the port is available)

Registers with the characteristics listed are optimized using the regular Quartus II Fitter optimizations.

Fast Input, Output, and Output Enable Registers

You can place individual registers in I/O cells manually by making fast I/O assignments with the Assignment Editor. For an input register, use the **Fast Input Register** option; for an output register, use the **Fast Output Register** option; and for an output enable register, use the **Fast Output Enable Register** option. Stratix II devices also support the **Fast OCT (on-chip termination) Register** option. In the MAX II series of devices, which have no I/O registers, these assignments lock the register into the LAB adjacent to the I/O pin if there is a pin location assignment for that I/O pin.

If the fast I/O setting is on, the register is always placed in the I/O element. If the fast I/O setting is off, the register is never placed in the I/O element. This is true even if the **Optimize IOC Register Placement for Timing** option is turned on. If there is no fast I/O assignment, the Quartus II software determines whether to place registers in I/O elements if the **Optimize IOC Register Placement for Timing** option is turned on.

The four fast I/O options (**Fast Input Register**, **Fast Output Register**, **Fast Output Enable Register**, and **Fast OCT Register**) also can be used to override the location of a register that is in a LogicLock region, and force it into an I/O cell. If this assignment is applied to a register that feeds multiple pins, the register is duplicated and placed in all relevant I/O elements. In MAX II series of devices, the register is duplicated and placed in each distinct LAB location that is next to an I/O pin with a pin location assignment.

Programmable Delays

Various programmable delay options can be used to minimize the t_{SU} and t_{CO} times. For Arria GX, Stratix, and Cyclone series devices, and MAX II series of devices, the Quartus II software automatically adjusts the applicable programmable delays to help meet timing requirements. For APEX series devices, the default values are set to avoid any hold time problems. Programmable delays are advanced options that you should use only after you compile a project, check the I/O timing, and determine that the timing is unsatisfactory. For detailed information about the effect of these options, refer to the device family handbook or data sheet.

After you have made a programmable delay assignment and compiled the design, you can view the value of every delay chain for every I/O pin in the **Delay Chain Summary** section of the Compilation Report.

You can assign programmable delay options to supported nodes with the Assignment Editor. You can also view and modify the delay chain setting for the target device with the Chip Planner and Resource Property Editor. When you use the Resource Property Editor to make changes after performing a full compilation, recompiling the entire design is not necessary; you can save changes directly to the netlist. Because these changes are made directly to the netlist, the changes are not made again automatically when you recompile the design. The change management features allow you to reapply the changes on subsequent compilations.

Though the programmable delays in newer devices like Arria II GX, Stratix IV, and Stratix III are user-controllable, Altera recommends their use for advanced users only. However, the Quartus II software might use the programmable delays internally during the Fitter phase.

For more details about Stratix III programmable delays, refer to the *Stratix III Device Handbook* and *AN 474: Implementing Stratix III Programmable I/O Delay Settings in the Quartus II Software*. For more information about using the Chip Planner and Resource Property Editor, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Table 10–2 summarizes the programmable delays available for Altera devices.

Table 10–2. Programmable Delays for Altera Devices (Part 1 of 3)

Programmable Delay	Description	I/O Timing Impact	Devices
Decrease input delay to input register	Decreases propagation delay from an input pin to the data input of the input register in the I/O cell associated with the pin. Applied to an input/bidirectional pin or register it feeds.	Decreases t_{SU} Increases t_H	■ Stratix ■ Stratix GX ■ Cyclone ■ MAX 7000B ■ APEX II ■ APEX 20KE ■ APEX 20KC
Input delay from pin to input register	Sets propagation delay from an input pin to the data input of the input register implemented in the I/O cell associated with the pin. Applied to an input/bidirectional pin.	Changes t_{SU} Changes t_H	■ Arria GX ■ Stratix II ■ Stratix II GX ■ Cyclone III ■ Cyclone II
Decrease input delay to internal cells	Decreases the propagation delay from an input or bidirectional pin to logic cells and embedded cells in the device. Applied to an input/bidirectional pin or register it feeds.	Decreases t_{SU} Increases t_H	■ Stratix ■ Stratix GX ■ Cyclone ■ APEX II ■ APEX 20KE ■ APEX 20KC ■ ACEX 1K ■ FLEX 10K ■ FLEX 6000
Input delay from pin to internal cells	Sets the propagation delay from an input or bidirectional pin to logic and embedded cells in the device. Applied to an input or bidirectional pin.	Changes t_{SU} Changes t_H	■ Stratix II ■ Stratix II GX ■ Cyclone III ■ Cyclone II ■ HardCopy II ■ MAX IIE and MAX IIZ

Table 10–2. Programmable Delays for Altera Devices (Part 2 of 3)

Programmable Delay	Description	I/O Timing Impact	Devices
Decrease input delay to output register	Decreases the propagation delay from the interior of the device to an output register in an I/O cell. Applied to an input/bidirectional pin or register it feeds.	Decreases t_{PD}	■ Arria GX ■ Stratix ■ Stratix GX ■ APEX II ■ APEX 20KE ■ APEX 20KC
Increase delay to output enable pin	Increases the propagation delay through the tri-state output to the pin. The signal can either come from internal logic or the output enable register in an I/O cell. Applied to an output/bidirectional pin or register feeding it.	Increases t_{CO}	■ Stratix ■ Stratix GX ■ APEX II
Delay to output enable pin	Sets the propagation delay to an output enable pin from internal logic or the output enable register implemented in an I/O cell.	Changes t_{CO}	■ Arria GX ■ Stratix II ■ Stratix II GX ■ Cyclone III
Increase delay to output pin	Increases the propagation delay to the output or bidirectional pin from internal logic or the output register in an I/O cell. Applied to output/bidirectional pin or register feeding it.	Increases t_{CO}	■ Stratix ■ Stratix GX ■ Cyclone ■ APEX II ■ APEX 20KE ■ APEX 20KC
Delay from output register to output pin	Sets the propagation delay to the output or bidirectional pin from the output register implemented in an I/O cell. This option is off by default.	Changes t_{CO}	■ Arria GX ■ Stratix II ■ Stratix II GX ■ Cyclone III ■ Cyclone II
Increase input clock enable delay	Increases the propagation delay from the interior of the device to the clock enable input of an I/O input register.	—	■ Stratix ■ Stratix GX ■ APEX II ■ APEX 20KE ■ APEX 20KC
Input delay from dual purpose clock pin to fan-out destinations	Sets the propagation delay from a dual-purpose clock pin to its fan-out destinations that are routed on the global clock network. Applied to an input or bidirectional dual-purpose clock pin.	—	■ Cyclone III ■ Cyclone II

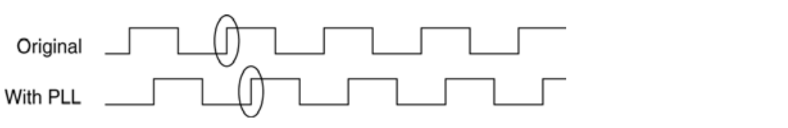
Table 10–2. Programmable Delays for Altera Devices (Part 3 of 3)

Programmable Delay	Description	I/O Timing Impact	Devices
Increase output clock enable delay	Increases the propagation delay from the interior of the device to the clock enable input of the I/O output register and output enable register.	—	■ Stratix ■ Stratix GX ■ APEX II ■ APEX 20KE ■ APEX 20KC
Increase output enable clock enable delay	Increases the propagation delay from the interior of the device to the clock enable input of an output enable register.	—	■ Stratix ■ Stratix GX
Increase t_{ZX} delay to output pin	Used for zero bus-turnaround (ZBT) by increasing the propagation delay of the falling edge of the output enable signal.	Increases t_{CO}	■ Stratix ■ Stratix GX ■ APEX II

Use PLLs to Shift Clock Edges

Using a PLL typically improves I/O timing automatically. If the timing requirements are still not met, most devices allow the PLL output to be phase shifted to change the I/O timing. Shifting the clock backwards gives a better t_{CO} at the expense of t_{SU}, while shifting it forward gives a better t_{SU} at the expense of t_{CO} and t_H. Refer to Figure 10–8. This technique can be used only in devices that offer PLLs with the phase shift option.

Figure 10–8. Shift Clock Edges Forward to Improve t_{SU} at the Expense of t_{CO}



You can achieve the same type of effect in certain devices using the programmable delay called **Input Delay from Dual Purpose Clock Pin to Fan-Out Destinations**, described in Table 10–2.

Use Fast Regional Clock Networks and Regional Clocks Networks

Altera devices have a variety of hierarchical clock structures available within them. These include dedicated global clock networks (GCLKs), regional clock networks (RCLKs), fast regional clock networks (FCLK) and periphery clock networks (PCLKs). The available resources differ between various Altera device families. Refer to the appropriate device handbook to get the number of various clocking resources available in your target device.

In general, fast regional clocks have less delay to I/O elements than regional and global clocks, and are used for high fan-out control signals. Regional clocks on the other hand provide the lowest clock delay and skew for logic contained in a single quadrant. Placing clocks on these low-skew and low-delay clock nets provides better t_{CO} performance.

Change How Hold Times are Optimized for MAX II Devices

For MAX II series of devices, you can use the **Guarantee I/O paths have zero hold time at Fast Timing Corner** option to control how hold time is optimized by the Quartus II software. On the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**. Click **More Settings.** In the **More Fitter Settings** dialog box, set the option globally. Or, on the Assignments menu, click **Assignment Editor** to set this option for specific I/Os.

The option controls whether the Fitter uses timing-driven compilation to optimize a design to achieve a zero hold time for I/Os that feed globally clocked registers at the fast (best-case) timing corner, even in the absence of any user timing assignments. When this option is set to **On** (default), the Fitter guarantees zero hold time (t_H) for I/Os feeding globally clocked registers at the fast timing corner, at the expense of possibly violating t_{SU} or t_{PD} timing constraints. When this option is set to **When tsu and tpd constraints permit**, the Fitter achieves zero hold time for I/Os feeding globally clocked registers at the fast timing corner only when t_{SU} or t_{PD} timing constraints are not violated. When this option is set to **Off**, designs are optimized to meet user timing assignments only.

By setting this option to **Off** or **When tsu and tpd constraints permit**, you improve t_{SU} at the expense of t_H.

Register-to-Register Timing Optimization Techniques (LUT-Based Devices)

The next stage of design optimization is to improve register-to-register (f_{MAX}) timing. There are a number of options available if the performance requirements are not achieved after compilation.

Note that the coding style affects the performance of your design to a greater extent than to other changes in settings. Always evaluate your code and make sure to use synchronous design practices. For more details, refer to the handbook chapter on design practices and coding guidelines.

When using the Quartus II TimeQuest Timing Analyzer, register-to-register timing optimization is the same as maximizing the slack on the clock domains in your domain. You can use the techniques described in this section to improve the slack on different timing paths in your design.

Before optimizing your design, you should understand the structure of your design as well as the type of logic affected by each optimization. An optimization can decrease performance if the optimization does not benefit your logic structure.

Improving Register-to-Register Timing Summary

The choice of options and settings to improve the timing margin (slack) or to improve register-to-register timing depends on the failing paths in the design. To achieve the results that best approximate your performance requirements, apply the following techniques and compile the design after each step:

1. Ensure that your timing assignments are complete. For details, refer to "Timing Requirement Settings" on page 10–4.

2. Ensure that you have reviewed all warning messages from your initial compilation and check for ignored timing assignments. Refer to "Design Analysis" on page 10–11 for details and fix any of these problems before proceeding with optimization.
3. Apply netlist synthesis optimization options and physical synthesis.
4. Try multiple different Fitter seeds (page 10–47). You can omit this step if a large number of critical paths are failing, or if the paths are failing badly.
5. Apply the following synthesis options to optimize for speed:
   - Optimize Synthesis for Speed, Not Area (page 10–44)
   - Flatten the Hierarchy During Synthesis (page 10–45)
   - Set the Synthesis Effort to High (page 10–45)
   - Change State Machine Encoding (page 10–45)
   - Duplicate Logic for Fan-Out Control (page 10–45)
   - Prevent Shift Register Inference (page 10–46)
   - Use Other Synthesis Options Available in Your Synthesis Tool (page 10–46)
6. Make LogicLock assignments (page 10–49) to control placement.
7. Make design source code modifications to fix areas of the design that are still failing timing requirements by significant amounts (page 10–48).
8. Make location assignments, or as a last resort, perform manual placement by back-annotating the design (page 10–51).

You can use the Design Space Explorer (DSE) to automate the process of running several different compilations with different settings. For more information, refer to the *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*.

If these techniques do not achieve performance requirements, additional design source code modifications might be required (page 10–48).

Physical Synthesis Optimizations

The Quartus II software offers physical synthesis optimizations that can help improve the performance of many designs regardless of the synthesis tool used. Physical synthesis optimizations can be applied both during synthesis and during fitting.

Physical synthesis optimizations that occur during the synthesis stage of the Quartus II compilation operate either on the output from another EDA synthesis tool or as an intermediate step in Quartus II integrated synthesis. These optimizations make changes to the synthesis netlist to improve either area or speed, depending on your selected optimization technique.

To view and modify the synthesis netlist optimization options, on the Assignments menu, click **Settings**. In the **Category** list, expand **Compilation Process Settings** and select **Physical Synthesis Optimizations**.

If you use a third-party EDA synthesis tool and want to determine if the Quartus II software can remap the circuit to improve performance, you can use the **Perform WYSIWYG Primitive Resynthesis** option. This option directs the Quartus II software to unmap the LEs in an atom netlist to logic gates and then map the gates back to Altera-specific primitives. Using Altera-specific primitives enables the Fitter to remap the circuits using architecture-specific techniques.

To turn on the **Perform WYSIWYG Primitive Resynthesis** option, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings** and check the box for **Perform WYSIWYG Primitive Resynthesis**.

The Quartus II technology mapper optimizes the design for **Speed**, **Area**, or **Balanced**, according to the setting of the **Optimization Technique** option. To change this setting, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and choose **Speed** or **Balanced** under **Optimization Technique**.

The physical synthesis optimizations occur during the Fitter stage of the Quartus II compilation. Physical synthesis optimizations make placement-specific changes to the netlist that improve speed performance results for a specific Altera device.

The following physical synthesis optimizations are available during the Fitter stage for improving performance:

- Physical synthesis for combinational logic
- Automatic asynchronous signal pipelining
- Physical synthesis for registers
  - Register duplication
  - Register retiming

You can apply physical synthesis options on specific instances if you want the performance gain from physical synthesis only on parts of your design.

To view and modify the **Physical Synthesis Optimizations**: On the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**, and specify the physical synthesis optimization options on the **Physical Synthesis Optimizations** page. You can also specify the **Physical synthesis effort**, which sets the level of physical synthesis optimization that you want the Quartus II software to perform.

The **Perform physical synthesis for combinational logic** option allows the Quartus II Fitter to resynthesize the combinational logic in a design to reduce delay along the critical path and improve design performance.

The **Perform automatic asynchronous signal pipelining** option allows the Quartus II Fitter to insert pipeline stages for asynchronous clear and asynchronous load signals automatically during fitting to increase circuit performance. You can use this option if asynchronous control signal recovery and removal times do not meet your requirements. The option improves performance for designs in which asynchronous signals in very fast clock domains cannot be distributed across the chip quickly enough (because of long global network delays).

To apply physical synthesis assignments for fitting on a per instance basis, use the Quartus II Assignment Editor. The following assignments are available as instance assignments:

- Perform physical synthesis for combinational logic
- Perform register duplication for performance
- Perform register retiming for performance
- Perform automatic asynchronous signal pipelining

In the Assignment Editor, indicate the module instance you want to apply to the specific physical synthesis setting in the **To** tab. Select the required physical synthesis assignment in the **Assignment Name** tab. In the **Value** tab, select **ON**. In the **Enabled** tab, select **Yes**.

The **Perform automatic asynchronous signal pipelining** option adds registers to nets driving the asynchronous clear or asynchronous load ports of registers. This adds register delays (and latency) to the reset, adding the same number of register delays for each destination using the reset. Therefore, the option should be used only when adding latency to reset signals does not violate any design requirements. This option also prevents the promotion of signals to use global routing resources.

The **Perform register duplication** physical synthesis option allows the Quartus II software to duplicate registers based on Fitter placement information to improve design performance. The Fitter can also duplicate combinational logic when this option is enabled.

The **Perform register retiming** physical synthesis option allows the Quartus II software to move registers across combinational logic to balance timing. This option applies to registers and combinational logic that have already been placed into logic cells.

You can perform physical synthesis during the fitter stage to improve the fitting results as well.

The Quartus II software performs the optimizations that help achieve a better fit when you turn on the **Perform physical synthesis for combinational logic** option.

The Fitter performs physical synthesis optimizations on logic and registers, allowing the mapping of logic and registers into unused memory blocks in the device to achieve a fit, when you turn on the **Perform logic to memory mapping** option.

For more information and detailed descriptions of these netlist optimization options, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

Because performance results are design-dependent, try the physical synthesis options in different combinations until you achieve the best results. Generally, turning on all the options gives the best results but significantly increases compilation time. The following information provides typical benchmark results on different designs with varying amounts of logic using synthesis netlists from leading third-party synthesis tools and compiled with Quartus II software. These results use the default **Balanced** setting for the Optimization Technique for WYSIWYG resynthesis. Changing the setting to **Speed** or **Area** can affect your results.

Using WYSIWIG primitive resynthesis can reduce area by about 8% in about 60% of designs, with an average improvement in f_{MAX} of about 3% and a maximum improvement in f_{MAX} of about 6%. By using other physical synthesis options for combinational logic and registers, you might be able to increase the f_{MAX} between 10% to 20% (depending on the physical synthesis effort level selected) in about 80% of the designs.

Compilation time might increase considerably when you use high physical synthesis effort levels, sometimes as much as 4×. The optimizations are design dependent, and some designs might not produce much improvement with physical synthesis.

Turn Off Extra-Effort Power Optimization Settings

If PowerPlay power optimization settings are set to **Extra Effort**, your design performance can be affected. If improving timing performance is more important than reducing power use, set the PowerPlay power optimization setting to **Normal**.

To change the PowerPlay power optimization level, on the Assignments menu, choose **Settings**. The **Setting** dialog box appears. From the **Category** list, select **Analysis & Synthesis Settings**. From the pull-down menu, select the appropriate level of PowerPlay power optimization level.

For more information about reducing power use, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Optimize Synthesis for Speed, Not Area

The manner in which the design is synthesized has a large impact on design performance. Design performance varies depending on the way the design is coded, the synthesis tool used, and the options specified when synthesizing. Change your synthesis options if a large number of paths are failing, or if specific paths are failing badly and have many levels of logic.

Set your device and timing constraints in your synthesis tool. Synthesis tools are timing-driven and optimized to meet specified timing requirements. If you do not specify target frequency, some synthesis tools optimize for area.

Some synthesis tools offer an easy way to instruct the tool to focus on speed instead of area.

For Quartus II integrated synthesis, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and specify **Speed** as the **Optimization Technique** option. You can also specify this logic option for specific modules in your design with the Assignment Editor while leaving the default **Optimization Technique** setting at **Balanced** (for the best trade-off between area and speed for certain device families) or **Area** (if area is an important concern). You can also use the **Speed Optimization Technique for Clock Domains** option to specify that all combinational logic in or between the specified clock domain(s) is optimized for speed.

To achieve best performance with push-button compilation, follow the recommendations in the following sections for other synthesis settings. You can use the DSE to experiment with different Quartus II synthesis options to optimize your design for the best performance.

For information about setting timing requirements and synthesis options in Quartus II integrated synthesis and third-party synthesis tools, refer to the appropriate chapter in *Section III. Synthesis* in volume 1 of the *Quartus II Handbook*, or refer to your synthesis software documentation.

Flatten the Hierarchy During Synthesis

Synthesis tools typically let you preserve hierarchical boundaries, which can be useful for verification or other purposes. However, the best optimization results generally occur when the synthesis tool optimizes across hierarchical boundaries, because doing so often allows the synthesis tool to perform the most logic minimization, which can improve performance. Whenever possible, flatten your design hierarchy to achieve the best results. If you are using Quartus II integrated synthesis, ensure that the **Preserve Hierarchical Boundary** option is turned off. If you are using Quartus II incremental compilation, you cannot flatten your design across design partitions. Incremental compilation always preserves the hierarchical boundaries between design partitions. Follow Altera's recommendations for design partitioning such as registering partition boundaries to reduce the effect of cross-boundary optimizations.

For more information about using incremental compilation and recommendations for design partitioning, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Set the Synthesis Effort to High

Some synthesis tools offer varying synthesis effort levels to trade off compilation time with synthesis results. Set the synthesis effort to high to achieve best results when applicable.

Change State Machine Encoding

State machines can be encoded using various techniques. One-hot encoding, which uses one register for every state bit, usually provides the best performance. If your design contains state machines, changing the state machine encoding to one-hot can improve performance at the cost of area.

If your design does not manually encode the state bits, you can select the state machine encoding chosen in your synthesis tool. In Quartus II integrated synthesis, on the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and for **State Machine Processing**, choose **One-Hot**. You also can specify this logic option for specific modules or state machines in your design with the Assignment Editor.

In some cases (especially in Stratix II and Stratix III devices), encoding styles other than the default offer better performance. Experiment with different encoding styles to see what effect the style has on your resource utilization and timing performance.

Duplicate Logic for Fan-Out Control

Duplicating logic or registers can help improve timing in cases where moving a register in a failing timing path to reduce routing delay creates other failing paths, or where there are timing problems due to the fan-out of the registers. Most often, timing failures occur not because of the high Fan-Out registers, but because of the location of those registers. Duplicating registers, where source and destination registers are physically close, can help improve slack on critical paths.

Many synthesis tools support options or attributes that specify the maximum fan-out of a register. When using Quartus II integrated synthesis, you can set the **Maximum Fan-Out** logic option in the Assignment Editor to control the number of destinations for a node so that the fan-out count does not exceed a specified value. You can also use the `maxfan` attribute in your HDL code. The software duplicates the node as required to achieve the specified maximum fan-out.

Logic duplication using **Maximum Fan-Out** assignments normally increases resource utilization and can potentially increase compilation time, depending on the placement and the total resource usage within the selected device. The improvement in timing performance that results because of **Maximum Fan-Out** assignments is very design-specific.

If you are using **Maximum Fan-Out** assignments, Altera recommends benchmarking your design with and without these assignments to evaluate whether they give the expected improvement in timing performance. Use the assignments only when you get improved results.

You can manually duplicate registers in the Quartus II software regardless of the synthesis tool used. To duplicate a register, apply the **Manual Logic Duplication** option to the register with the Assignment Editor.

The **Manual Logic Duplication** option also accepts wildcards. This is an easy and powerful duplication technique that you can use without editing your source code. For example, you can use this technique to make a duplicate of a large fan-out node for all of its destinations in a certain design hierarchy, such as `hierarchy_A`. To apply such an assignment in the Assignment Editor, make an entry such as the one shown in Table 10–3.

Table 10–3. Duplicating Logic in the Assignment Editor

From	To	Assignment Name	Value
`My_high_fanout_node`	`*hierarchy_A*`	Manual Logic Duplication	`high_fanout_to_A`

For more information about the manual logic duplication option, refer to the Quartus II Help.

Prevent Shift Register Inference

In some cases, turning off the inference of shift registers increases performance. Doing so forces the software to use logic cells to implement the shift register instead of implementing the registers in memory blocks using the ALTSHIFT_TAPS megafunction. If you implement shift registers in logic cells instead of memory, logic utilization is increased.

Use Other Synthesis Options Available in Your Synthesis Tool

With your synthesis tool, experiment with the following options if they are available:

- Turn on register balancing or retiming
- Turn on register pipelining
- Turn off resource sharing

These options can increase performance. They typically increase the resource utilization of your design.

Fitter Seed

The Fitter seed affects the initial placement configuration of the design. Changing the seed value changes the Fitter results because the fitting results change whenever there is a change in the initial conditions. Because each seed value results in a somewhat different fit, you can experiment with several different seeds to attempt to obtain better fitting results and timing performance.

When there are changes in your design, there is some random variation in performance between compilations. This variation is inherent in placement and routing algorithms—there are too many possibilities to try them all and get the absolute best result, so the initial conditions change the compilation result.

Any design change that directly or indirectly affects the Fitter has the same type of random effect as changing the seed value. This includes any change in source files, **Analysis & Synthesis Settings**, **Fitter Settings**, or **Timing Analyzer Settings**. The same effect can appear if you use a different computer processor type or different operating system because different systems can change the way floating point numbers are calculated in the Fitter.

If a change in optimization settings slightly affects the register-to-register timing or number of failing paths, you can't always be certain that your change caused the improvement or degradation, or whether it could be due to random effects in the Fitter. If your design is still changing, running a seed sweep (compiling your design with multiple seeds) determines whether the average result has improved after an optimization change and whether a setting that increases compilation time has benefits worth the increased time (such as setting the **Physical Synthesis Effort** to **Extra**). The sweep also shows the amount of random variation you should expect for your design.

If your design is finalized, you can compile your design with different seeds to obtain one optimal result. However, if you subsequently make any changes to your design, you will likely have to perform seed sweep again.

On the Assignments menu, select **Fitter Settings** to control the initial placement with the Seed. You can use the DSE to perform a seed sweep easily.

You can use the following Tcl command from a script to specify a Fitter seed:

```
set_global_assignment -name SEED <value> ↵
```

For more information about compiling with different seeds using the DSE script, refer to the *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*.

Set Maximum Router Timing Optimization Level

To improve routability in designs where the router did not pick up the optimal routing lines, set the **Router Timing Optimization Level** to **Maximum**. This setting determines how aggressively the router tries to meet timing requirements. Setting this option to **Maximum** can increase design speed slightly at the cost of increased compilation time. Setting this option to **Minimum** can reduce compilation time at the cost of slightly reduced design speed. The default value is **Normal**.

To modify the **Router Timing Optimization Level**, on the Assignments menu, click **Settings**. The **Settings** dialog box appears. In the **Category** list, click **Fitter Settings**. Click on the **More Settings** tab. From the available settings, select **Router Timing Optimization Level** and choose the required setting from the list.

Enable Beneficial Skew Optimization

The Quartus II Fitter intentionally inserts some small delays on global clock networks to improve performance on designs that target Arria II GX, Stratix IV, Stratix III, and Cyclone III devices. This is called beneficial skew optimization and is enabled by default for devices that support this feature.The value of skew introduced depends on the device family and the speed grade of the chosen device. For example, when this option is turned on for a Stratix III device (-2 speed grade), a skew value of approximately 150 ps is introduced if the inclusion improves the timing performance of your design. If you are targeting a Cyclone III device (-6 speed grade), the delay value introduced is approximately 350 ps. For Arria II GX and Stratix IV devices, an approximate skew of 100 ps could be introduced. To enable the **Beneficial Skew Optimization** option, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Fitter Settings**. The **Fitter Settings** page appears.
3. Click **More Settings**. The **More Fitter Settings** dialog box appears.
4. Under **Options**, in the **Name** list, select **Enable Beneficial Skew Optimization**. In the **Setting** list, select **On**.
5. Click **OK**.
6. In the **Settings** dialog box, click **OK**.

When you turn on **Enable Beneficial Skew Optimization** globally, you can disable skew insertion on a particular clock or destination by using an instance level `ENABLE_BENEFICIAL_SKEW_OPTIMIZATION` assignment.

When you want to use **Enable Beneficial Skew Optimization**, you must also set the **Optimize hold timing** option to **All paths** (in the **Fitter Settings** page of the **Settings** dialog box). If you turn on **Enable Beneficial Skew Optimization**, the fitter overrides the setting of **Optimize hold timing** if it is not set to **All paths**, and displays an info message describing the change.

Optimize Source Code

If the methods described in the preceding sections do not sufficiently improve timing of the design, modify your design files to achieve the desired results. Try restructuring the design to use pipelining or more efficient coding techniques. In many cases, optimizing the design’s source code can have a very significant effect on your design performance. In fact, optimizing your source code is typically the most effective technique for improving the quality of your results, and is often a better choice than using LogicLock or location assignments.

If the critical path in your design involves memory or DSP functions, check whether you have code blocks in your design that describe memory or functions that are not being inferred and placed in dedicated logic. You might be able to modify your source code to cause these functions to be placed into high-performance dedicated memory or resources in the target device.

Ensure that your state machines are recognized as state machine logic and optimized appropriately in your synthesis tool. State machines that are recognized are generally optimized better than if the synthesis tool treats them as generic logic. In the Quartus II software, you can check for the State Machine report under **Analysis & Synthesis** in the Compilation Report. This report provides details, including the state encoding for each state machine that was recognized during compilation. If your state machine is not being recognized, you might have to change your source code to enable it to be recognized.

For coding style guidelines including examples of HDL code for inferring memory, functions, guidelines, and sample HDL code for state machines, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

LogicLock Assignments

Using LogicLock assignments to improve timing performance is only recommended for older Altera devices, such as the APEX 20K family. For designs using these devices, you can make LogicLock assignments for based nodes optimization, design hierarchy, or critical paths. This method can be used if a large number of paths are failing, and recoding the design does not seem to be necessary. LogicLock assignments can help if routing delays form a large portion of your critical path delay, and placing logic closer together in the device improves the routing delay.

Improving fitting results with LogicLock assignments, especially for larger devices, such as the Stratix and Arria GX series of devices can be difficult. The LogicLock feature is intended to be used for performance preservation and to floorplan your design. Therefore, LogicLock assignments do not always improve the performance of the design. In many cases, you cannot improve upon results from the Fitter by making location assignments.

If there are existing LogicLock assignments in your design, remove the assignments if your design methodology permits it. Recompile the design to see if the assignments are making the performance worse.

When making LogicLock assignments, it is important to consider how much flexibility to give the Fitter. LogicLock assignments provide more flexibility than hard location assignments. Assignments that are more flexible require higher Fitter effort, but reduce the chance of design over-constraint. The following types of LogicLock assignments are available, listed in the order of decreasing flexibility:

- Auto size, floating location regions
- Fixed size, floating location regions
- Fixed size, locked location regions

For more information about using LogicLock regions, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

To determine what to put into a LogicLock region, refer to the timing analysis results and analyze the critical paths in the Chip Planner (Timing Closure Floorplan for older devices). The register-to-register timing paths in the Timing Analyzer section of the Compilation Report helps you recognize patterns.

The following sections describe cases in which LogicLock regions can help to optimize a design.

Hierarchy Assignments

For a design with the hierarchy shown in Figure 10–9, which has failing paths in the timing analysis results similar to those shown in Table 10–4, mod_A is probably a problem module. In this case, a good strategy to fix the failing paths is to place the mod_A hierarchy block in a LogicLock region so that all the nodes are closer together in the floorplan.

Figure 10–9. Design Hierarchy

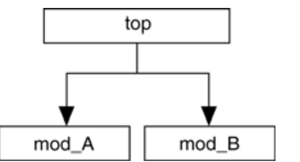


Table 10–4 shows the failing paths connecting two regions together within mod_A listed in the timing analysis report.

Table 10–4. Failing Paths in a Module Listed in Timing Analysis

From	To
|mod_A|reg1	|mod_A|reg9
|mod_A|reg3	|mod_A|reg5
|mod_A|reg4	|mod_A|reg6
|mod_A|reg7	|mod_A|reg10
|mod_A|reg0	|mod_A|reg2

Hierarchical LogicLock regions are also important if you are using an incremental compilation flow. Each design partition for incremental compilation should be placed in a separate LogicLock region to reduce conflicts and ensure good quality of results as the design develops. You can use auto size and floating location regions to find a good design floorplan, but you should fix the size and placement to achieve the best results in future compilations.

For more information about using incremental compilation and recommendations for creating a design floorplan using LogicLock regions, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* and *Best Practices for Incremental Compilation and Floorplan Assignments* chapters in volume 1 of the *Quartus II Handbook*, and *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Path Assignments

If you see a pattern such as the one shown in Figure 10–10 and Table 10–5, it often indicates paths with a common problem. In this case, a path-based assignment can be made from all d_reg registers to all memaddr registers. You can make a path-based assignment to place all source registers, destination registers, and the nodes between them in a LogicLock region with the wildcard characters “*” and “?.”

You also can explicitly place the nodes of a critical path in a LogicLock region. However, using this method instead of path assignments can result in alternate paths between the source and destination registers becoming critical paths.

Figure 10–10. Failing Paths in Timing Analysis

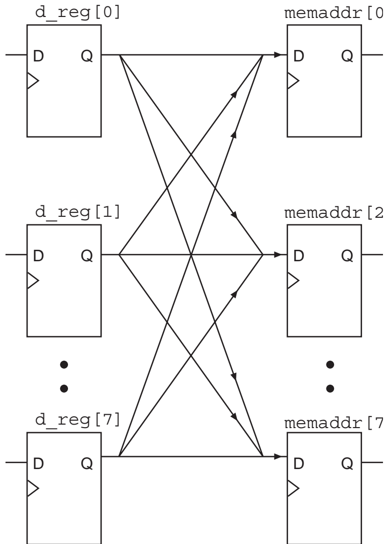


Table 10–5 shows the failing paths listed in the timing analysis report.

Table 10–5. Failing Paths in Timing Analysis

From	To
|d_reg[1]	|memaddr[5]
|d_reg[1]	|memaddr[6]
|d_reg[1]	|memaddr[7]
|d_reg[2]	|memaddr[0]
|d_reg[2]	|memaddr[1]

For more information about path-based LogicLock assignments, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Location Assignments and Back-Annotation

If a small number of paths are failing to meet their timing requirements, you can use hard location assignments to optimize placement. Location assignments are less flexible for the Quartus II Fitter than LogicLock assignments. In some cases, when you are very familiar with your design, you can enter location constraints in a way that produces better results.

Improving fitting results, especially for larger devices, such as the Stratix and Arria GX series of devices, can be difficult. Location assignments do not always improve the performance of the design. In many cases, you cannot improve upon the results from the Fitter by making location assignments.

The following commonly used location assignments are listed in the order of decreasing flexibility:

- Custom regions
- Back-annotated LAB location assignments
- Back-annotated LE or ALM location assignments

Custom Regions

A custom region is a rectangular region containing user-assigned nodes, which are constrained in the region's boundaries. If any portion of a block in the device floorplan overlaps a custom region, such as an M-RAM block, it is considered to be entirely in that region.

Custom regions are hard location assignments that cannot be overridden and are very similar to fixed-size, locked-location, LogicLock regions. Custom regions are commonly used when logic must be constrained to a specific portion of the device.

Back-Annotation and Manual Placement

Assigning the location of nodes in a design to the locations to which they were assigned during the last compilation is called "back-annotation." When nodes are locked to their assigned locations in a back-annotated design, you can manually move specific nodes without affecting other back-annotated nodes. The process of manually moving and reassigning specific nodes is called manual placement.

Back-annotation is very restrictive to the compiler, so you should back-annotate only when the design has been finalized and no further changes are expected. Assignments can become invalid if the design is changed. Combinational nodes often change names when a design is resynthesized, even if they are unrelated to the logic that was changed.

Moving nodes manually can be very difficult for large devices. In many cases, you cannot improve upon the Fitter's results. Illegal or unroutable location constraints can cause "no fit" errors. Before making location assignments, determine whether to back-annotate to lock down the assigned locations of all nodes in the design. When you are using a hierarchical design flow, you can lock down node locations in one LogicLock region only, while other node locations are left floating in a fixed LogicLock region. By implementing a hierarchical approach, you can use the LogicLock design methodology to reduce the dependence of logic blocks on other logic blocks in the device.

Consistent node names are required to perform back-annotation. If you use Quartus II integrated synthesis or any Quartus II optimizations, such as the WYSIWYG primitive resynthesis netlist optimization or any physical synthesis optimizations, you must create an atom netlist before you back-annotate to lock down the placement of any nodes. This creates consistent node names.

Physical synthesis optimizations are placement-specific as well as design-specific. Unless you back-annotate the design before recompilation, the physical synthesis results can differ. This happens because the atom netlist creates different placement results. By back-annotating the design, the design source and the atom netlist use the same placement when the design is recompiled. When you use an atom netlist and you want to maintain the same placement results as a previous compilation, use LogicLock regions and back-annotate the placement of all nodes in the design. Not back-annotating the design can result in the design source and the atom netlist having different placement results and therefore different synthesis results.

For more information about creating atom netlists for your design, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

When you back-annotate a design, you can choose whether to assign the nodes either to LABs (this is preferred because of increased flexibility) or LEs/ALMs. You also can choose to back-annotate routing to further restrict the Fitter and force a specific routing within the device.

Using back-annotated routing with physical synthesis optimizations can result in a routing failure.

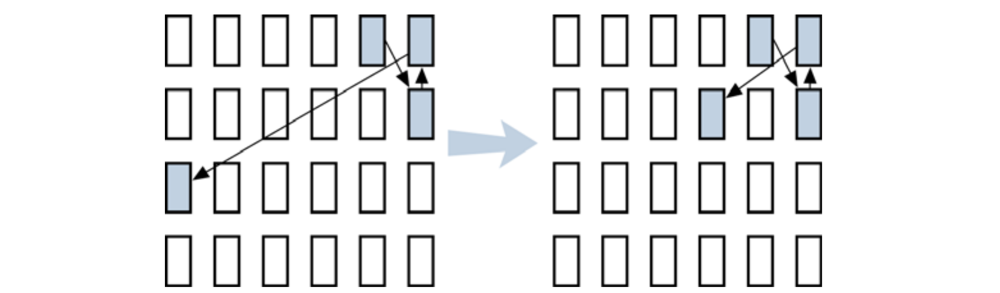
For more information about back-annotating routing, refer to the Quartus II Help.

When performing manual placement at a detailed level, Altera recommends that you move LABs, not logic cells (LEs or ALMs). The Quartus II software places nodes that share the same control signals in appropriate LABs. Successful placement and routing is more difficult when you move individual logic cells. This is because LEs with different control signals that are put into the same LAB might not have any unused control signals available, and the design might not fit.

In general, when you are performing manual placement and routing, fix all I/O paths first, because often fewer options are available to meet I/O timing. After I/O timing is met, focus on manually placing register-to-register timing paths. This strategy is consistent with the methodology outlined in this chapter.

The best way to meet performance is to move nodes closer together. For a critical path such as the one shown in Figure 10–11, moving the destination node closer to the other nodes reduces the delay and helps meet your timing requirements.

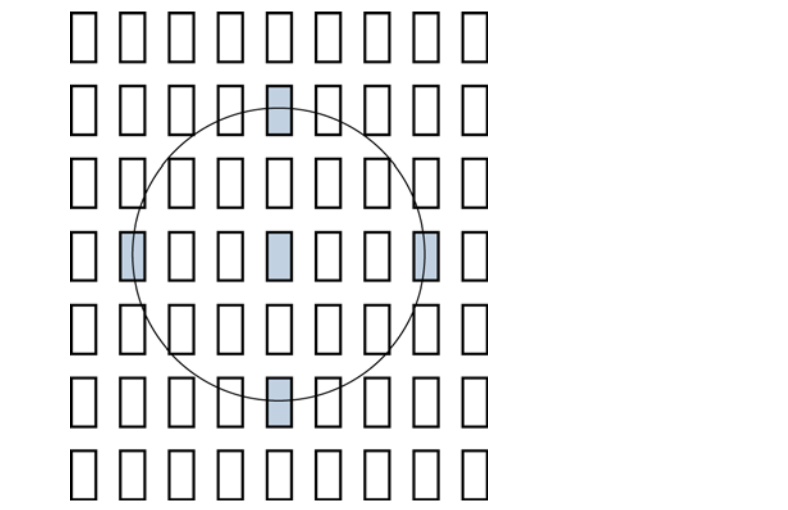
Figure 10–11. Reducing Delay of Critical Path



Optimizing Placement for Stratix, Stratix II, Arria GX, and Cyclone II Devices

In the Arria GX, Stratix, and Cyclone series of devices, the row interconnect delay is slightly faster than the column interconnect delay. Therefore, when placing nodes, optimal placement is typically an ellipse around the source or destination node. In Figure 10–12, if the source is located in the center, any of the shaded LABs should give approximately the same delay.

Figure 10–12. Possible Optimal Placement Ellipse



In addition, you should avoid crossing any M-RAM memory blocks for node-to-node routing, because routing paths across M-RAM blocks requires using R24 or C16 routing lines.

The Quartus II software calculates the interconnect delay based on different electrical characteristics of each individual wire, such as the length, fan-out, distribution of the parasitic loading on the wire, and so forth.

To determine the actual delays to and from a resource, use the **Show Physical Timing Estimate** feature in the Chip Planner.

For more information about using the Chip Planner or the Timing Closure Floorplan, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Optimizing Placement for Cyclone Devices

In Cyclone devices, the row and column interconnect delays are similar; therefore, when placing nodes, optimal placement is typically a circle around the source or destination node.

Try to avoid long routes across the device. Long routes require more than one routing line to cross the Cyclone device.

Optimizing Placement for APEX II and APEX 20KE/C Devices

For the APEX II and APEX 20KE/C device families, the delay for paths is reduced by placing the source and destination nodes in the same geographical resource location. The following device resources are listed in order from fastest to slowest:

- LAB
- MegaLAB structure
- MegaLAB column
- Row

For example, if the nodes cannot be placed in the same MegaLAB structure to reduce the delay, place them in the same MegaLAB column. For the actual delays to and from resources, use the **Show Physical Timing Estimate** feature in the Timing Closure Floorplan.

Resource Utilization Optimization Techniques (Macrocell-Based CPLDs)

The following recommendations help you take advantage of the macrocell-based architecture in the MAX 7000 and MAX 3000 device families to yield maximum speed, reliability, and device resource utilization while minimizing fitting difficulties.

After design analysis, the first stage of design optimization is to improve resource utilization. Complete this stage before proceeding to timing optimization. First, ensure that you have set the basic constraints described in "Initial Compilation: Required Settings" on page 10–3. If your design is not fitting into a specified device, use the techniques in this section to achieve a successful fit.

Use Dedicated Inputs for Global Control Signals

MAX 7000 and MAX 3000 devices have four dedicated inputs that can be used for global register control. Because the global register control signals can bypass the logic cell array and directly feed registers, product terms can be preserved for primary logic. Also, because each signal has a dedicated path into the LAB, global signals also can bypass logic and data path interconnect resources.

Because the dedicated input pins are designed for high fan-out control signals and provide low skew, you should always assign global signals (such as clock, clear, and output enable) to the dedicated input pins.

You can use logic-generated control signals for global control signals instead of dedicated inputs. However, the following list shows the disadvantages of using logic-generated control signals:

- More resources are required (logic cells, interconnect).
- More data skew is introduced.
- If the logic-generated control signals have high fan-out, the design can be more difficult to fit.

By default, the Quartus II software uses dedicated inputs for global control signals automatically. You can assign control signals to dedicated input pins in one of the following ways:

- In the Assignment Editor, choose one of the two following methods:
  - Assign pins to dedicated pin locations.
  - Assign a **Global Signal** setting to the pins.

- On the Assignments menu, click **Settings**. On the **Analysis & Synthesis Settings** page, in the **Auto Global Options** section, in the **Category** list, select **Register Control Signals**.
- Insert a `GLOBAL` primitive after the pins.
- If you have already assigned pins for the design in the MAX+PLUS® II software, on the Assignments menu, click **Import Assignments**.

Reserve Device Resources

Because pin and logic option assignments can be necessary for board layout and performance requirements, and because full utilization of the device resources can increase the difficulty of fitting the design, Altera recommends that you leave 10% of the device's logic cells and 5% of the I/O pins unused to accommodate future design modifications. Following the Altera-recommended device resource reservation guidelines for macrocell-based CPLDs increases the chance that the Quartus II software can fit the design during recompilation after changes or assignments have been made.

Pin Assignment Guidelines and Procedures

Sometimes user-specified pin assignments are necessary for board layout. This section discusses pin assignment guidelines and procedures.

To minimize fitting issues with pin assignments, follow these guidelines:

- Assign speed-critical control signals to dedicated inputs.
- Assign output enables to appropriate locations.
- Estimate fan-in to assign output pins to the appropriate LAB.
- Assign output pins that require parallel expanders to macrocells numbered 4 to 16.

Altera recommends that you allow the Quartus II software to choose pin assignments automatically when possible. You can use the Quartus II Pin Advisor feature (accessible from the Tools menu) for pin connection guidelines. For more information about the Pin Advisor, refer to Quartus II Help.

Control Signal Pin Assignments

Assign speed-critical control signals to dedicated input pins. Every MAX 7000 and MAX 3000 device has four dedicated input pins (`GCLK1`, `OE2/GCLK2`, `OE1`, and `GCLRn`). You can assign clocks to global clock dedicated inputs (`GCLK1` and `OE2/GCLK2`), clear to the global clear dedicated input (`GCLRn`), and speed-critical output enable to global OE dedicated inputs (`OE1` and `OE2/GCLK2`).

Output Enable Pin Assignments

Occasionally, because the total number of required output enable pins is more than the dedicated input pins, output enable signals must be assigned to I/O pins.

To minimize possible fitting errors when assigning the output enable pins for MAX 7000 and MAX 3000 devices, refer to **Pin-Out Files for Altera Devices** on the Altera website (www.altera.com).

Estimate Fan-In When Assigning Output Pins

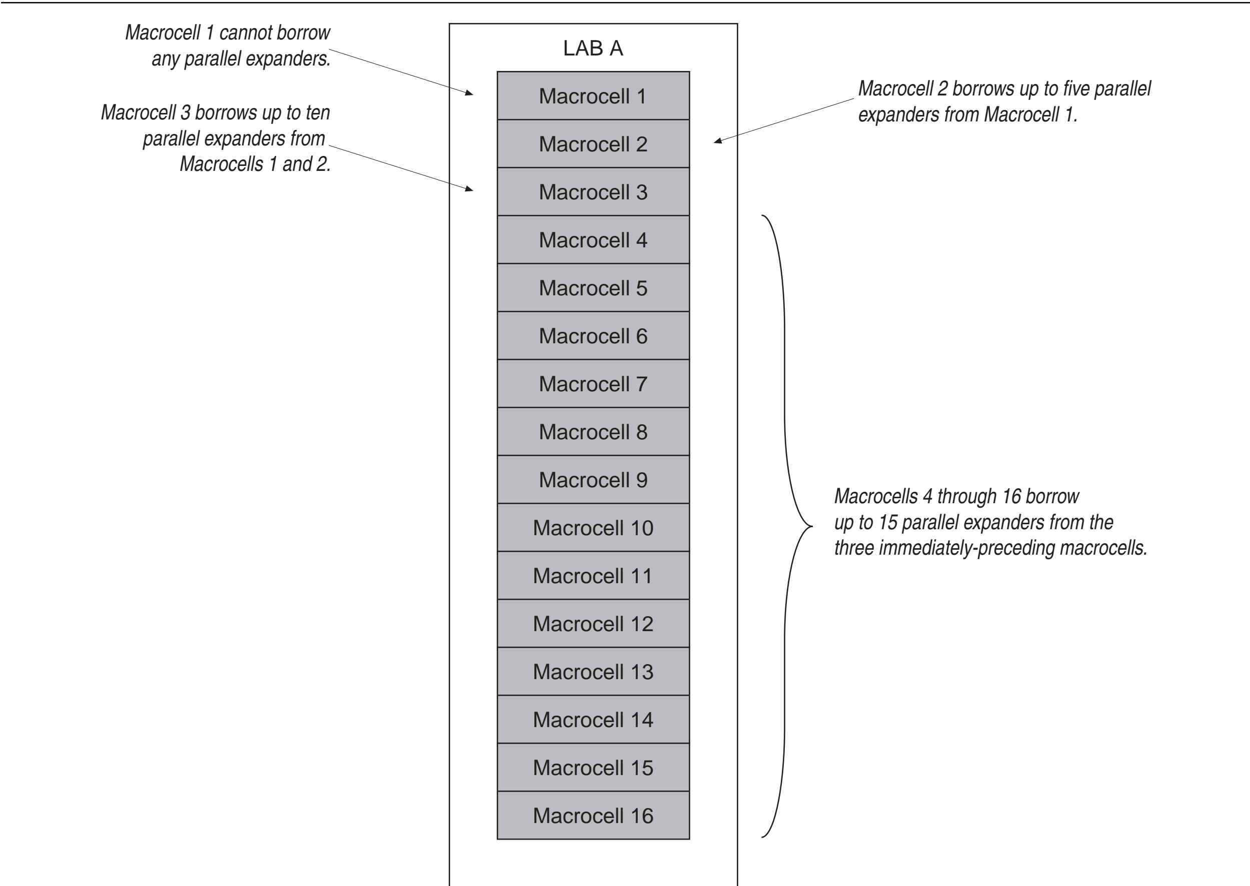
Macrocells with high fan-in can cause more placement problems for the Quartus II Fitter than those with low fan-in. The maximum fan-in per LAB should not exceed 36 in MAX 7000 and MAX 3000 devices. Therefore, estimate the fan-in of logic (such as an *x*-input AND gate) that feeds each output pin. If the total fan-in of logic that feeds each output pin in the same LAB exceeds 36, compilation can fail. To save resources and prevent compilation errors, avoid assigning pins that have high fan-in.

Outputs Using Parallel Expander Pin Assignments

Figure 10–13 illustrates how parallel expanders are used within a LAB. MAX 7000 and MAX 3000 devices contain chains that can lend or borrow parallel expanders. The Quartus II Fitter places macrocells in a location that allows them to lend and borrow parallel expanders appropriately.

As shown in Figure 10–13, only macrocells 2 through 16 can borrow parallel expanders. Therefore, assign output pins that might require parallel expanders to pins adjacent to macrocells 4 through 16. Altera recommends using macrocells 4 through 16 because they can borrow the largest number of parallel expanders.

Figure 10–13. LAB Macrocells and Parallel Expander Associations



Resolving Resource Utilization Problems

Two common Quartus II compilation fitting issues cause errors: excessive macrocell usage and lack of routing resources. Macrocell usage errors occur when the total number of macrocells in the design exceed the available macrocells in the device. Routing errors occur when the available routing resources are insufficient to implement the design. Check the Message window for the compilation results.

Messages in the Messages window are also copied in the Report Files. Right-click on a message and click **Help** for more information.

Resolving Macrocell Usage Issues

Occasionally, a design requires more macrocell resources than are available in the selected device, which results in the design not fitting. The following list provides tips for resolving macrocell usage issues as well as tips to minimize the number of macrocells used.

- On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and turn off **Auto Parallel Expanders**. If the design's clock frequency (f_{MAX}) is not an important design requirement, turn off parallel expanders for all or part of the project. The design usually requires more macrocells if parallel expanders are turned on.
- Change **Optimization Technique** from **Speed** to **Area**. Selecting **Area** instructs the compiler to give preference to area utilization rather than speed (f_{MAX}). On the Assignments menu, click **Settings**. In the **Category** list, change the **Optimization Technique** option in the **Analysis & Synthesis Settings** page.
- Use D-type flipflops instead of latches. Altera recommends that you always use D-type flipflops instead of latches in your design because D-type flipflops can reduce the macrocell fan-in, and thus reduce macrocell usage. The Quartus II software uses extra logic to implement latches in MAX 7000 and MAX 3000 designs because MAX 7000 and MAX 3000 macrocells contain D-type flipflops instead of latches.
- Use asynchronous clear and preset instead of synchronous clear and preset. To reduce the product term usage, use asynchronous clear and preset in your design whenever possible. Using other control signals such as synchronous clear produces macrocells and pins with higher fan-out.

After following the suggestions in this section, if your project still does not fit the targeted device, consider using a larger device. When upgrading to a different density, the vertical package-migration feature of the MAX 7000 and MAX 3000 device families allows pin assignments to be maintained.

Resolving Routing Issues

Routing is another resource that can cause design fitting issues. For example, if the total fan-in into a LAB exceeds the maximum allowed, a no-fit error can occur during compilation. If your design does not fit the targeted device because of routing issues, consider the following suggestions.

- Use dedicated inputs/global signals for high fan-out signals. The dedicated inputs in MAX 7000 and MAX 3000 devices are designed for speed-critical and high fan-out signals. Always assign high fan-out signals to dedicated inputs/global signals.
- Change the **Optimization Technique** option from **Speed** to **Area**. This option can resolve routing resource and macrocell usage issues. Refer to "Resolving Macrocell Usage Issues" on page 10–58.
- Reduce the fan-in per cell. If you are not limited by the number of macrocells used in the design, you can use the **Fan-in per cell (%)** option to reduce the fan-in per cell. The allowable values are 20–100%; the default value is 100%. Reducing the fan-in can reduce localized routing congestion but increase the macrocell count. You can set this logic option in the Assignment Editor or under **More Settings** in the **Analysis & Synthesis Settings** page of the **Settings** dialog box.
- On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and turn off **Auto Parallel Expanders**. By turning off the parallel expanders, you give the Quartus II software more fitting flexibility for each macrocell, allowing macrocells to be relocated. For example, each macrocell (previously grouped together in the same LAB) can be moved to a different LAB to reduce routing constraints.
- Insert logic cells. Inserting logic cells reduces fan-in and shared expanders used per macrocell, increasing routability. By default, the Quartus II software automatically inserts logic cells when necessary. Otherwise, **Auto Logic Cell** can be disabled as follows. On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**. Under More **Settings**, turn off **Auto Logic Cell Insertion**. Refer to "Using LCELL Buffers to Reduce Required Resources" for more information.
- Change pin assignments. If you want to discard your pin assignments, you can let the Quartus II Fitter ignore some or all of the assignments.

If you prefer reassigning pins to increase routing efficiency, refer to "Pin Assignment Guidelines and Procedures" on page 10–56.

Using LCELL Buffers to Reduce Required Resources

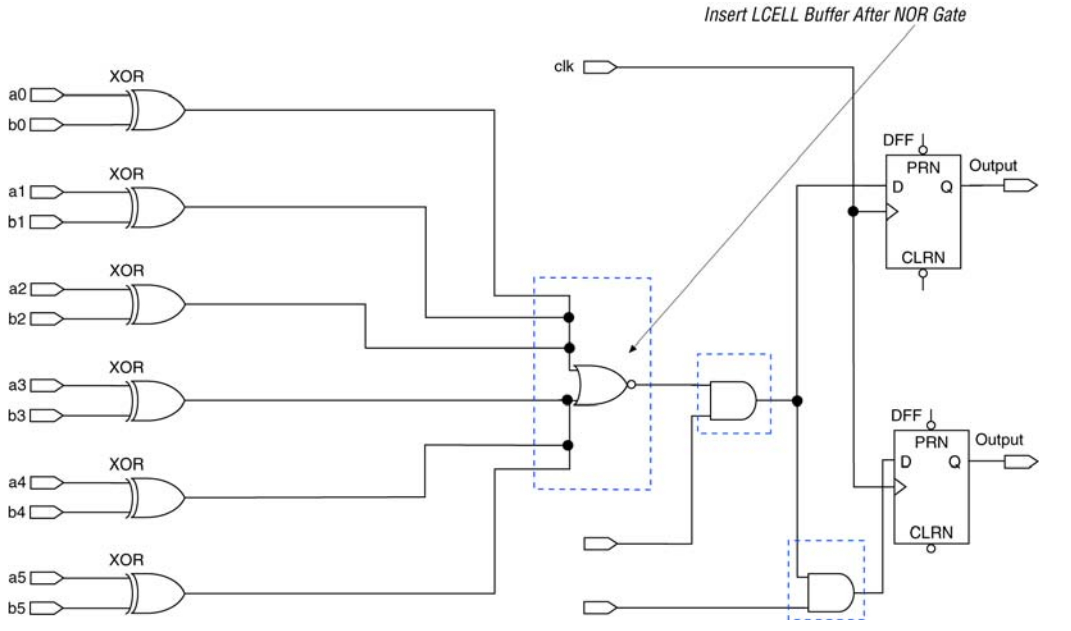
Complex logic, such as multilevel XOR gates, are often implemented with more than one macrocell. When this occurs, the Quartus II software automatically allocates shareable expanders—or additional macrocells (called synthesized logic cells)—to supplement the logic resources that are available in a single macrocell. You can also break down complex logic by inserting logic cells in the project to reduce the average fan-in and the total number of shareable expanders required. Manually inserting logic cells can provide greater control over speed-critical paths.

Instead of using the Quartus II software's **Auto Logic Cell Insertion** option, you can manually insert logic cells. However, Altera recommends that you use the **Auto Logic Cell Insertion** option unless you know which part of the design is causing the congestion.

A good location to manually insert LCELL buffers is where a single complex logic expression feeds multiple destinations in your design. You can insert an LCELL buffer just after the complex expression; the Quartus II Fitter extracts this complex expression and places it in a separate logic cell. Rather than duplicate all the logic for each destination, the Quartus II software feeds the single output from the logic cell to all destinations.

To reduce fan-in and prevent no-fit compilations caused by routing resource issues, insert an LCELL buffer after a NOR gate (Figure 10–14). The design in Figure 10–14 was compiled for a MAX 7000AE device. Without the LCELL buffer, the design requires two macrocells and eight shareable expanders, and the average fan-in is 14.5 macrocells. However, with the LCELL buffer, the design requires three macrocells and eight shareable expanders, and the average fan-in is just 6.33 macrocells.

Figure 10–14. Reducing the Average Fan-In by Inserting LCELL Buffers



Timing Optimization Techniques (Macrocell-Based CPLDs)

After resource optimization, design optimization focuses on timing. Ensure that you have made the appropriate assignments as described in "Initial Compilation: Required Settings" on page 10–3, and that the resource utilization is satisfactory before proceeding with timing optimization.

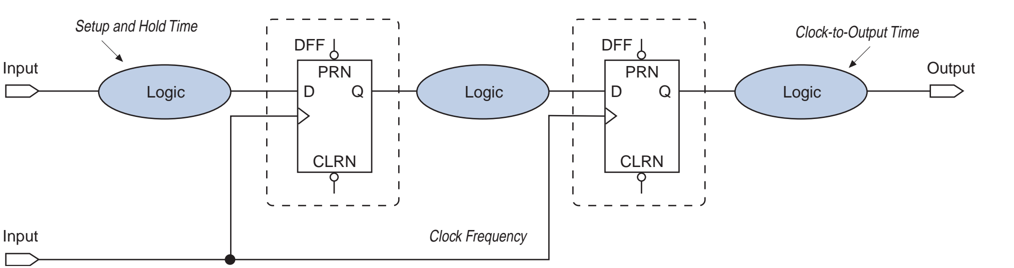
Maintaining system performance at or above certain timing requirements is an important goal of circuit designs. The following five timing parameters are primarily responsible for a design's performance:

- Setup time (t_{SU})—the propagation time for input data signals
- Hold time (t_H)—the propagation time for input data signals
- Clock-to-output time (t_{CO})—the propagation time for output signals
- Pin-to-pin delays (t_{PD})—the time required for a signal from an input pin to propagate through combinational logic and appear at an external output pin

- Maximum clock frequency (f_{MAX})—the internal register-to-register performance

This section provides guidelines to improve the timing if the timing requirements are not met. Figure 10–15 shows the parts of the design that determine the t_{SU}, t_H, t_{CO}, t_{PD}, and f_{MAX} timing parameters.

Figure 10–15. Main Timing Parameters that Determine the System's Performance



Timing results for t_{SU}, t_H, t_{CO}, t_{PD}, and f_{MAX} are found in the Compilation Report for the Quartus II Classic Timing Analyzer, as discussed in "Design Analysis" on page 10–11.

When you are analyzing a design to improve performance, be sure to consider the two major contributors to long delay paths:

- Excessive levels of logic
- Excessive loading (high fan-out)

When a MAX 7000 or MAX 3000 device signal drives more than one LAB, the programmable interconnect array (PIA) delay increases by 0.1 ns per additional LAB fan-out. Therefore, to minimize the added delay, concentrate the destination macrocells into fewer LABs, minimizing the number of LABs that are driven. The main cause of long delays in circuit design is excessive levels of logic.

Improving Setup Time

Sometimes the t_{SU} timing reported by the Quartus II Fitter does not meet your timing requirements. To improve the t_{SU} timing, refer to the following guidelines:

- Turn on the **Fast Input Register** option using the Assignment Editor. The **Fast Input Register** option allows input pins to directly drive macrocell registers via the fast-input path, thus minimizing the pin-to-register delay. This option is useful when a pin drives a D-type flipflop and there is no combinational logic between the pin and the register.
- Reduce the amount of logic between the input and the register. Excessive logic between the input pin and register causes more delays. To improve setup time, Altera recommends that you reduce the amount of logic between the input pin and the register whenever possible.
- Reduce fan-out. The delay from input pins to macrocell registers increases when the fan-out of the pins increases. To improve the setup time, minimize the fan-out.

Improving Clock-to-Output Time

To improve a design's clock-to-output time, minimize the register-to-output-pin delay. To improve the t_{CO} timing, refer to the following guidelines:

- Use the global clock. In addition to minimizing the delay from a register to an output pin, minimizing the delay from the clock pin to the register can also improve t_{CO} timing. Always use the global clock for low-skew and speed-critical signals.
- Reduce the amount of logic between the register and output pin. Excessive logic between the register and the output pin causes more delay. Always minimize the amount of logic between the register and output pin for faster clock-to-output time.

Table 10–6 shows the timing results for an EPM7064AETC100-4 device when a combination of the **Fast Input Register** option, global clock, and minimal logic is used. When the **Fast Input Register** option is turned on, the t_{SU} timing is improved (t_{SU} decreases from 1.6 ns to 1.3 ns and from 2.8 ns to 2.5 ns). The t_{CO} timing is improved when the global clock is used for low-skew and speed-critical signals (t_{CO} decreases from 4.3 ns to 3.1 ns). However, if there is additional logic used between the input pin and the register or the register and the output pin, the t_{SU} and t_{CO} delays increase.

Table 10–6. EPM7064AETC100-4 Device Timing Results

Number of Registers	t_{SU} (ns)	t_H (ns)	t_{CO} (ns)	Global Clock Used	Fast Input Register Option	D Input Location	Q Output Location	Additional Logic Between:	
								D Input Location & Register	Register & Q Output Location
1	1.3	1.2	4.3	—	On	LAB A	LAB A	—	—
1	1.6	0.3	4.3	—	Off	LAB A	LAB A	—	—
1	2.5	0	3.1	✓	On	LAB A	LAB A	—	—
1	2.8	0	3.1	✓	Off	LAB A	LAB A	—	—
1	3.6	0	3.1	✓	Off	LAB A	LAB A	✓	—
1	2.8	0	7.0	✓	Off	LAB D	LAB A	—	✓
16 with the same D and clock inputs	2.8	0	All 6.2	✓	Off	LAB D	LAB A, B	—	—
32 with the same D and clock inputs	2.8	0	All 6.4	✓	Off	LAB C	LAB A, B, C	—	—

Improving Propagation Delay (t_{PD})

Achieving fast propagation delay (t_{PD}) timing is required in many system designs. However, if there are long delay paths through complex logic, achieving fast propagation delays can be difficult. To improve your design's t_{PD}, refer to the following guidelines.

- On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and turn on **Auto Parallel Expanders**. Turning on the parallel expanders for individual nodes or sub-designs can increase the performance of complex logic functions. However, if the project's pin or logic cell assignments use parallel expanders placed physically together with macrocells (which can reduce routability), parallel expanders can cause the Quartus II Fitter to have difficulties

finding and optimizing a fit. Additionally, the number of macrocells required to implement the design increases and results in a no-fit error during compilation if the device resources are limited. For more information about turning on the **Auto Parallel Expanders** option, refer to "Resolving Macrocell Usage Issues" on page 10–58.

- Set the **Optimization Technique** to **Speed**. By default, the Quartus II software sets the **Optimization Technique** option to **Speed** for MAX 7000 and MAX 3000 devices. Reset the **Optimization Technique** option to **Speed** only if you previously set it to **Area**. On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, and turn on **Speed** under **Optimization Technique**.

Improving Maximum Frequency (f_{MAX})

Maintaining the system clock at or above a certain frequency is a major goal in circuit design. For example, if you have a fully synchronous system that must run at 100 MHz, the longest delay path from the output of any register to the inputs of the registers it feeds must be less than 10 ns. Maintaining the system clock speed can be difficult if there are long delay paths through complex logic. Altera recommends that you perform the following guidelines to improve your design's clock speed (f_{MAX}).

- On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings** and turn on **Auto Parallel Expanders**. Turning on the parallel expanders for individual nodes or subdesigns can increase the performance of complex logic functions. However, if the project's pin or logic cell assignments use parallel expanders placed physically together with macrocells (which can reduce routability), parallel expanders can cause the Quartus II compiler to have difficulties finding and optimizing a fit. Additionally, the number of macrocells required to implement the design also increases and can result in a no-fit error during compilation if the device's resources are limited. For more information about using the **Auto Parallel Expanders** option, refer to "Resolving Macrocell Usage Issues" on page 10–58.
- Use global signals or dedicated inputs. Altera MAX 7000 and MAX 3000 devices have dedicated inputs that provide low skew and high speed for high fan-out signals. Minimize the number of control signals in the design and use the dedicated inputs to implement them.
- Set the **Optimization Technique** to **Speed**. By default, the Quartus II software sets the **Optimization Technique** option to **Speed** for MAX 7000 and MAX 3000 devices. Reset the **Optimization Technique** option to **Speed** only if you have previously set it to **Area**. You can reset the **Optimization Technique** option. In the **Category** list, choose **Analysis & Synthesis Settings**, and turn on **Speed** under **Optimization Technique**.
- Pipeline the design. Pipelining, which increases clock frequency (f_{MAX}), refers to dividing large blocks of combinational logic by inserting registers.

Optimizing Source Code—Pipelining for Complex Register Logic

If the methods described in the preceding sections do not sufficiently improve your results, modify the design at the source to achieve the desired results. Using a pipelining technique consumes device resources, but it also lowers the propagation delay between registers, allowing you to maintain high system clock speed.

Compilation-Time Optimization Techniques

If reducing the compilation time of your design is important, use the techniques in this section. Be aware that reducing compilation time with some of these techniques can reduce the overall quality of results. A Compilation Time Advisor is also available in the Quartus II software, which helps you to reduce the compilation time. You can run the Compilation Time Advisor on the Tools menu by pointing to **Advisors** and clicking **Compilation Time Advisor**. You can find all the compilation time optimizing techniques described in this section in the Compilation Time Advisor as well.

If you open the Compilation Time Advisor after compilation, it displays recommendations on settings that can reduce the compilation time. Some of the recommendations from different advisors can contradict each other; Altera recommends evaluating the options, and choosing the settings that best suit your design requirements.

Incremental Compilation

The incremental compilation feature can speed up design iteration time by an average of 60% when making changes to the design and helps you reach design timing closure more efficiently. Using incremental compilation allows you to organize your design into logical and physical partitions for design synthesis and fitting. Design iterations can be made dramatically faster by recompiling only a particular design partition and merging results with previous compilation results from other partitions. You can also use physical synthesis optimization techniques for specific design partitions while leaving other modules untouched to preserve performance.

If you are using a third-party synthesis tool, you can create separate atom netlist files for parts of your design that you already have synthesized and optimized so that you update only the parts of the design that change.

Regardless of your synthesis tool, you can use full incremental compilation along with LogicLock regions to preserve your placement and routing results for unchanged partitions while working on other partitions. This ability provides the most reduction in compilation time and run-time memory usage because neither synthesis nor fitting is performed for unchanged partitions in the design.

You can also perform a bottom-up compilation in which parts of the design are compiled completely independently in separate Quartus II projects, and then exported into the top-level design. This flow is useful in team-based designs or when incorporating third-party IP.

For information about the full incremental compilation flow in the Quartus II software, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For information about creating multiple netlist files in third-party tools for use with incremental compilation, refer to the appropriate chapter in *Section III. Synthesis* in volume 1 of the *Quartus II Handbook*.

Use Multiple Processors for Parallel Compilation

The Quartus II software can run some algorithms in parallel to take advantage of multiple processors and reduce compilation time when more than one processor is available to compile the design. Parallel compilation is turned on by default in the Quartus II software and the software can detect if multiple processors are available in a computer used for compiling your design. You can also specify the maximum number of processors that the software can use if you want to reserve some of the available processors for other tasks. The Quartus II software supports up to four processors. The software does not necessarily use all the processors that you specify during a given compilation, but it never uses more than the specified number of processors. This allows you to work on other tasks on your computer without it becoming slow or less responsive. However, for interactive tasks such as word processing, it is typically not necessary to restrict the number of processors in this manner.

By allowing the Quartus II software to use two processors, you might be able to reduce the compilation time by up to 10% on systems with two processing cores and by up to 15% on systems with four cores. With certain design flows in which timing analysis runs alone, using multiple processors can reduce the time required for timing analysis by an average of 12% when using two processors. This reduction can reach an average of 15% when using four processors.

The actual reduction in compilation time depends on the design and on the specific settings used for compilation. For example, compilations with fast-corner optimization turned on benefit more from using multiple processors than do compilations that do not use fast-corner optimization. The runtime requirement is not reduced for some other compilation goals, such as Analysis and Synthesis. The Fitter (`quartus_fit`), the Classic Timing Analyzer (`quartus_tan`), and the TimeQuest Timing Analyzer (`quartus_sta`) stages in the compilation might benefit from the use of multiple processors. The average number of processors used for these stages is shown in the Compilation Report, on the **Flow Elapsed Time** panel. A more detailed breakdown of processor usage is also shown in the Parallel Compilation panel of the appropriate report, such as the Fit report. This panel is only displayed if parallel compilation is enabled.

This feature is available for Arria GX, Stratix, Cyclone, and MAX II series of devices.

Do not consider processors with Intel Hyper-Threading to be more than one processor. If you have a single processor with Intel Hyper-Threading enabled, you should set the number of processors to one. Altera recommends that you do not use the Intel Hyper-Threading feature for Quartus II compilations, as it can increase runtimes.

Many factors can impact the performance of parallel compilation. For detailed information and instructions that can help improve the performance of this feature, refer to the solution to the problem "How can I improve the compilation time performance of the parallel compilation feature in the Quartus II software?" on the Altera website, www.altera.com.

The Quartus II software can detect the number of processors available on a computer and use all the processors (up to 4) to reduce compilation time. You can also control the number of processors used during a compilation on a per user basis by performing the following steps:

1. On the Tools menu, click **Options**. The **Options** dialog box appears.
2. In the **Category** list, select **Processing**. The **Processing** page appears.
3. Under **Parallel compilation**, select **Use all available processors** or specify the **Maximum processors allowed** for compilation.

These settings are applicable to all projects unless you override it with other local project-specific settings.

To control the number of processors used during compilation for a specific project, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Compilation Process Settings**. The **Compilation Process Settings** page appears.
3. Under **Parallel compilation**, choose **Use global parallel compilation setting** if you want a global setting for parallel compilation. If you want a different option for this project, select the **Use all available processors** which utilizes all processors. If you do not want to run the compilation on all available processors, choose the **Maximum processors allowed** and type in the number of processors to be used for compilation. The default value for the number of processors is `1`.

Using multiple processors does not affect the quality of the fit. For a given Fitter seed on a specific design, the fit is exactly the same, regardless of whether the Quartus II software uses one processor or multiple processors. The only difference between such compilations using a different number of processors is the compilation time.

You can also set the number of processors available for Quartus II compilation using the following Tcl command in your script.

```
set_global_assignment -name NUM_PARALLEL_PROCESSORS <value> ↵
```

In this case, <*value*> is an integer from `1` to `4`.

If you want the Quartus II software to detect the number of processors and use all of them for running the compilation, use the following Tcl command in your script:

```
set_global_assignment -name NUM_PARALLEL_PROCESSORS ALL ↵
```

Reduce Synthesis Time and Synthesis Netlist Optimization Time

You can reduce synthesis time by reducing your use of netlist optimizations and by using incremental compilation. Use incremental compilation (with Netlist Type set to **Post-Synthesis**) to reduce the synthesis time, without affecting the Fitter time. For more ideas about reducing synthesis time in third-party EDA synthesis tools, refer to your synthesis software's documentation.

Synthesis Netlist Optimizations

You can use Quartus II integrated synthesis to synthesize and optimize HDL designs, and you can use synthesis netlist optimizations to optimize netlists that were synthesized by third-party EDA software. Using these netlist optimizations can cause the Analysis and Synthesis module to take much longer to run. Look at the Analysis and Synthesis messages to find out how much time these optimizations take. The compilation time spent in Analysis and Synthesis is typically small compared to the compilation time spent in the Fitter.

If your design meets your performance requirements without synthesis netlist optimizations, turn off the optimizations to save time. If you require synthesis netlist optimizations to meet performance, you can optimize parts of your design hierarchy separately to reduce the overall time spent in analysis and synthesis.

Check Early Timing Estimation before Fitting

The Quartus II software can provide an estimate of your timing results after synthesis, before the design is fully processed by the Fitter. In cases where you want a quick estimate of your design results before proceeding with further design or synthesis tasks, this feature can save you significant compilation time. For more information, refer to "Early Timing Estimation" on page 10–7.

After you perform analysis and synthesis in the Quartus II software, on the Processing menu, point to **Start**, and click **Start Early Timing Estimate**.

Reduce Placement Time

The time required to place a design depends on two factors: the number of ways the logic in the design can be placed in the device and the settings that control how hard the placer works to find a good placement. You can reduce the placement time in two ways:

- Change the settings for the placement algorithm
- Use incremental compilation to preserve the placement for parts of the design

Sometimes there is a trade-off between placement time and routing time. Routing time can increase if the placer does not run long enough to find a good placement. When you reduce placement time, make sure that it does not increase routing time and negate the overall time reduction.

Fitter Effort Setting

Standard fit takes the most runtime and usually does not yield a better result than **Auto Fit**. To switch from **Standard** to **Auto Fit**, on the Assignments menu, click **Settings**. In the **Category** list, select **Fitter Settings**, and use the **Fitter effort** setting to shorten runtime by changing the effort level to **Auto Fit**. If you are certain that your design has only easy-to-meet timing constraints, you can select **Fast Fit** for an even greater runtime saving.

Placement Effort Multiplier Settings

You can control the amount of time the Fitter spends in placement by reducing one aspect of placement effort with the **Placement Effort Multiplier** option. On the Assignments menu, click **Settings**. Select **Fitter Settings**, and click **More Settings**. Under **Existing Option Settings**, select **Placement Effort Multiplier**. The default is `1.0`. Legal values must be greater than 0 and can be non-integer values. Numbers between 0 and 1 can reduce fitting time, but also can reduce placement quality and design performance. Numbers higher than 1 increase placement time and placement quality, but can reduce routing time for designs with routing congestion. For example, a value of 4 increases placement time by approximately 2 to 4 times, but might result in better placement, which can result in reduced routing time.

Final Placement Optimization Levels

The **Final Placement Optimization Level** option specifies whether the Fitter performs final placement optimizations. This can be set to **Always**, **Never**, and **Automatically**. Performing optimizations can improve register-to-register timing and fitting, but might require longer compilation times. The default setting of **Automatically** can be used with the **Auto Fit** Fitter Effort Level (also the default) to let the Fitter decide whether these optimizations should run based on the routability and timing requirements of the design.

Setting the **Final Placement Optimization Level** to **Never** often reduces your compilation time, but typically affects routability negatively and reduces timing performance.

To change the **Final Placement Optimization Level**, on the Assignments menu, choose **Settings**. The **Settings** dialog box appears. From the **Category** list, select **Fitter Settings**. Click the **More Settings** button. Select **Final Placement Optimization Level**, and then from the list, select the required setting.

Physical Synthesis Effort Settings

You can use the physical synthesis options to optimize your post-synthesis netlist and improve your timing performance. These options, which affect placement, can significantly increase compilation time.

If your design meets your performance requirements without physical synthesis options, turn them off to save time. You also can use the **Physical synthesis effort** setting on the **Physical Synthesis Optimizations** page under **Fitter Settings** in the **Category** list to reduce the amount of extra compilation time that these optimizations use. The **Fast** setting directs the Quartus II software to use a lower level of physical synthesis optimization that, compared to the normal level, can cause a smaller increase in compilation time. However, the lower level of optimization can result in a smaller increase in design performance.

Limit to One Fitting Attempt

This option causes the software to quit after one fitting attempt option, instead of repeating placement and routing with increased effort.

From the Assignments menu, select **Settings**. On the **Fitter Settings** page, turn on **Limit to one fitting attempt**.

For more details about this option, refer to "Limit to One Fitting Attempt" on page 10–9.

Preserving Placement, Incremental Compilation, and LogicLock Regions

Preserving information about previous placements can make future placements take less time. The incremental compilation provides an easy-to-use methodology for preserving placement results. For more information, refer to "Incremental Compilation" on page 10–64 and the references listed in the section.

Reduce Routing Time

The time required to route a design depends on three factors: the device architecture, the placement of the design in the device, and the connectivity between different parts of the design. Typically, the routing time is not a significant amount of the compilation time. If your design takes a long time to route, perform one or more of the following actions:

- Check for routing congestion
- Let the placer run longer to find a more routable placement
- Use incremental compilation to preserve routing information for parts of your design

Identify Routing Congestion in the Chip Planner

To identify areas of congested routing in your design, open the Chip Planner. On the Tools menu, click **Chip Planner**. To view the routing congestion in the Chip Planner, click the Layers icon located next to the Task menu. Under **Background Color Map**, select the **Routing Utilization**. Routing resource usage above 90% indicates routing congestion. You can change the connections in your design to reduce routing congestion. If the area with routing congestion is in a LogicLock region or between LogicLock regions, change or remove the LogicLock regions and recompile the design. If the routing time remains the same, the time is a characteristic of the design and the placement. If the routing time decreases, consider changing the size, location, or contents of LogicLock regions to reduce congestion and decrease routing time.

For information about identifying areas of congested routing using the Chip Planner tool, refer to the "Viewing Routing Congestion" subsection in the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Identify Routing Congestion in the Timing Closure Floorplan for Legacy Devices

If the device in your design is not supported by the Chip Planner, you must use the Timing Closure Floorplan tool to identify areas of congested routing in your design. To open the Timing Closure Floorplan, on the Assignments menu, click **Timing Closure Floorplan,** and turn on **Show Routing Congestion**. This feature is available only when you choose the **Field View** on the View menu. Routing resource usage above 90% indicates routing congestion. You can change the connections in your design to reduce routing congestion. If the area with routing congestion is in a LogicLock region or between LogicLock regions, change or remove the LogicLock regions and recompile the design. If the routing time remains the same, the time is a characteristic of the design and the placement. If the routing time decreases, consider changing the size, location, or contents of LogicLock regions to reduce congestion and decrease routing time.

Placement Effort Multiplier Setting

Some designs might be difficult to route and take a long time to route because the placement is less than optimal. In such cases, you can increase the Placement Effort Multiplier to get a better placement. This might increase the placement time, but it can reduce the routing time, and even overall compilation time in some cases.

Preserve Routing Incremental Compilation and LogicLock Regions

Preserving information about the previous routing results for part of the design can make future routing efforts take less time. The use of LogicLock regions with incremental compilation provides an easy-to-use methodology that preserves placement and routing results. For more information, refer to "Incremental Compilation" on page 10–64 and the references listed in the section.

Setting Process Priority

It might be necessary to reduce the computing resources allocated to the Quartus II compilation at the expense of increased compilation time. It can be convenient to reduce the resource allocation to the Quartus II compilation with single processor machines if you also have to run other tasks at the same time.

To run a Quartus II compilation at a reduced process priority, perform the following steps:

1. On the Tools menu, click **Options**. The **Options** dialog box appears.
2. In the **Category** list, under **General**, select **Processing**. The **Processing** page appears.
3. Turn on **Run design processing at a lower priority (recommended for single processor machines)**.

When you turn on this option, it is applied to all future compilations.

Using this option can increase your compilation time.

Other Optimization Resources

The Quartus II software has additional resources to help you optimize your design for resource, performance, compilation time, and power.

Design Space Explorer

The DSE automates the process of running multiple compilations with different settings. You can use the DSE to try the techniques described in this chapter. The DSE utility helps automate the process of finding the best set of options for your design. The DSE explores the design space by applying various optimization techniques and analyzing the results.

For more information, refer to the *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*.

Other Optimization Advisors

The Quartus II software has a Power Optimization Advisor to provide guidance for reducing power consumption. In addition, the Incremental Compilation Advisor provides suggestions to improve your quality of results when partitioning your design for a hierarchical or team-based design flow using the Quartus II incremental compilation feature.

For more information about using the Power Optimization Advisor, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*. Fore more information about using the Incremental Compilation Advisor, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II command-line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF form.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

You can specify many of the options described in this section either in an instance, or at a global level, or both.

Use the following Tcl command to make a global assignment:

```
set_global_assignment -name <.qsf variable name> <value> ↵
```

Use the following Tcl command to make an instance assignment:

```
set_instance_assignment -name <.qsf variable name> <value> \
-to <instance name> ↵
```

If the *<value>* field includes spaces (for example, “Standard Fit”), the value must be enclosed by straight double quotation marks.

Initial Compilation Settings

The Quartus II Settings File (**.qsf**) variable name is used in the Tcl assignment to make the setting along with the appropriate value. The **Type** column indicates whether the setting is supported as a global setting, an instance setting, or both.

This chapter refers to timing settings and analysis in the Quartus II Classic Timing Analyzer. For equivalent settings and analysis in the Quartus II TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* or the *Switching to the Quartus II TimeQuest Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*.

Table 10–7 lists the **.qsf** file variable name and applicable values for the settings discussed in "Initial Compilation: Required Settings" on page 10–3. Table 10–8 shows the list of advanced compilation settings.

Table 10–7. Initial Compilation Settings

Setting Name	.qsf File Variable Name	Values	Type
Device Setting	DEVICE	<*device part number*>	Global
Use Smart Compilation	SPEED_DISK_USAGE_TRADEOFF	SMART, NORMAL	Global
Optimize IOC Register Placement For Timing	OPTIMIZE_IOC_REGISTER_PLACEMENT_FOR_TIMING	ON, OFF	Global
Optimize Hold Timing	OPTIMIZE_HOLD_TIMING	OFF, IO PATHS AND MINIMUM TPD PATHS, ALL PATHS	Global
Fitter Effort	FITTER_EFFORT	STANDARD FIT, FAST FIT, AUTO FIT	Global

Table 10–8. Advanced Compilation Settings

Setting Name	.qsf File Variable Name	Values	Type
Router Effort Multiplier	ROUTER_EFFORT_MULTIPLIER	Any positive, non-zero value	Global
Router Timing Optimization level	ROUTER_TIMING_OPTIMIZATION_LEVEL	NORMAL, MINIMUM, MAXIMUM	Global
Final Placement Optimization	FINAL_PLACEMENT_OPTIMIZATION	ALWAYS, AUTOMATICALLY, NEVER	Global

Resource Utilization Optimization Techniques (LUT-Based Devices)

Table 10–9 lists the **.qsf** file variable name and applicable values for the settings discussed in "Resource Utilization Optimization Techniques (LUT-Based Devices)" on page 10–19. The **.qsf** file variable name is used in the Tcl assignment to make the setting along with the appropriate value. The **Type** column indicates whether the setting is supported as a global setting, an instance setting, or both.

Table 10–9. Resource Utilization Optimization Settings (Part 1 of 2)

Setting Name	.qsf File Variable Name	Values	Type
Auto Packed Registers *(1)*	AUTO_PACKED_REGISTERS_<*device family name*>	OFF, NORMAL, MINIMIZE AREA, MINIMIZE AREA WITH CHAINS, AUTO	Global, Instance
Perform WYSIWYG Primitive Resynthesis	ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP	ON, OFF	Global, Instance

Table 10–9. Resource Utilization Optimization Settings (Part 2 of 2)

Setting Name	.qsf File Variable Name	Values	Type
Physical Synthesis for Combinational Logic for Reducing Area	PHYSICAL_SYNTHESIS_COMBO_LOGIC_FOR_AREA	ON, OFF	Global, Instance
Physical Synthesis for Mapping Logic to Memory	PHYSICAL_SYNTHESIS_MAP_LOGIC_TO_MEMORY_FOR AREA	ON, OFF	Global, Instance
Optimization Technique	<*device family name*>_OPTIMIZATION_TECHNIQUE	AREA, SPEED, BALANCED	Global, Instance
Speed Optimization Technique for Clock Domains	SYNTH_CRITICAL_CLOCK	ON, OFF	Instance
State Machine Encoding	STATE_MACHINE_PROCESSING	AUTO, ONE-HOT, MINIMAL BITS, USER-ENCODE	Global, Instance
Preserve Hierarchy	PRESERVE_HIERARCHICAL_BOUNDARY	OFF, RELAXED, FIRM	Instance
Auto RAM Replacement	AUTO_RAM_RECOGNITION	ON, OFF	Global, Instance
Auto ROM Replacement	AUTO_ROM_RECOGNITION	ON, OFF	Global, Instance
Auto Shift Register Replacement	AUTO_SHIFT_REGISTER_RECOGNITION	ON, OFF	Global, Instance
Auto Block Replacement	AUTO_DSP_RECOGNITION	ON, OFF	Global, Instance
Number of Processors for Parallel Compilation	NUM_PARALLEL_PROCESSORS	Integer between 1 and 4 inclusive, or ALL	Global

Note to Table 10–9:

(1) Allowed values for this setting depend on the device family that is selected.

I/O Timing Optimization Techniques (LUT-Based Devices)

Table 10–10 lists the **.qsf** file variable name and applicable values for the settings discussed in "I/O Timing Optimization Techniques (LUT-Based Devices)" on page 10–73. The **.qsf** file variable name is used in the Tcl assignment to make the setting along with the appropriate value. The **Type** column indicates whether the setting is supported as a global setting, an instance setting, or both.

Table 10–10. I/O Timing Optimization Settings

Setting Name	.qsf File Variable Name	Values	Type
Optimize IOC Register Placement For Timing	OPTIMIZE_IOC_REGISTER_PLACEMENT_FOR_TIMING	ON, OFF	Global
Fast Input Register	FAST_INPUT_REGISTER	ON, OFF	Instance
Fast Output Register	FAST_OUTPUT_REGISTER	ON, OFF	Instance
Fast Output Enable Register	FAST_OUTPUT_ENABLE_REGISTER	ON, OFF	Instance
Fast OCT Register	FAST_OCT_REGISTER	ON, OFF	Instance

Register-to-Register Timing Optimization Techniques (LUT-Based Devices)

Table 10–11 lists the **.qsf** file variable name and applicable values for the settings discussed in "Register-to-Register Timing Optimization Techniques (LUT-Based Devices)" on page 10–40. The **.qsf** file variable name is used in the Tcl assignment to make the setting along with the appropriate value. The **Type** column indicates whether the setting is supported as a global setting, an instance setting, or both.

Table 10–11. Register-to-Register Timing Optimization Settings

Setting Name	.qsf File Variable Name	Values	Type
Perform WYSIWYG Primitive Resynthesis	ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP	ON, OFF	Global, Instance
Perform Physical Synthesis for Combinational Logic	PHYSICAL_SYNTHESIS_COMBO_LOGIC	ON, OFF	Global, Instance
Perform Register Duplication	PHYSICAL_SYNTHESIS_REGISTER_DUPLICATION	ON, OFF	Global, Instance
Perform Register Retiming	PHYSICAL_SYNTHESIS_REGISTER_RETIMING	ON, OFF	Global, Instance
Perform Automatic Asynchronous Signal Pipelining	PHYSICAL_SYNTHESIS_ASYNCHRONOUS_SIGNAL_PIPELINING	ON, OFF	Global, Instance
Physical Synthesis Effort	PHYSICAL_SYNTHESIS_EFFORT	NORMAL, EXTRA, FAST	Global
Fitter Seed	SEED	<*integer*>	Global
Maximum Fan-Out	MAX_FANOUT	<*integer*>	Instance
Manual Logic Duplication	DUPLICATE_ATOM	<*node name*>	Instance
Optimize Power during Synthesis	OPTIMIZE_POWER_DURING_SYNTHESIS	NORMAL, OFF EXTRA_EFFORT	Global
Optimize Power during Fitting	OPTIMIZE_POWER_DURING_FITTING	NORMAL, OFF EXTRA_EFFORT	Global

Duplicate Logic for Fan-Out Control

The manual logic duplication option accepts wildcards. This is an easy and powerful duplication technique that you can use without editing your source code. You can use this technique, for example, to make a duplicate of a large fan-out node for all of its destinations in a certain design hierarchy, such as `hierarchy_A`. To make such an assignment with Tcl, use a command similar to Example 10–1.

Example 10–1. Duplication Technique

```
set_instance_assignment -name DUPLICATE_ATOM high_fanout_to_A -from \
high_fanout_node -to *hierarchy_A*
```

Conclusion

Using the recommended techniques described in this chapter can help you close timing quickly on complex designs, reduce iterations by providing more intelligent and better linkage between analysis and assignment tools, and balance multiple design constraints including multiple clocks, routing resources, and area constraints.

The Quartus II software provides many features to achieve optimal results. Follow the techniques presented in this chapter to efficiently optimize a design for area or timing performance, or to reduce compilation time.

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Analyzing Designs with Quartus II Netlist Viewers* chapter in volume 1 of the *Quartus II Handbook*
- *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*
- *Best Practices for Incremental Compilation and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Design Analysis and Engineering Change Management with Chip Planner* chapter in volume 3 of the *Quartus II Handbook*
- *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*
- *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *Managing Metastability with the Quartus II Software* chapter in the *Quartus II Handbook*
- *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*
- *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 10–12 shows the revision history for this chapter.

Table 10–12. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Was chapter 8 in the 8.1.0 release. ■ Updated the following sections: → "Timing Analysis with the TimeQuest Timing Analyzer" on page 10–14 → "Perform WYSIWYG Resynthesis with Balanced or Area Setting" on page 10–22 → "Use Physical Synthesis Options to Reduce Area" on page 10–26 → "Metastability Analysis and Optimization Techniques" on page 10–32 → "Use Fast Regional Clock Networks and Regional Clocks Networks" on page 10–39 → "Register-to-Register Timing Optimization Techniques (LUT-Based Devices)" on page 10–40 → "Physical Synthesis Optimizations" on page 10–41 → "Duplicate Logic for Fan-Out Control" on page 10–45 → "LogicLock Assignments" on page 10–49 → "Enable Beneficial Skew Optimization" on page 10–48 → "Use Multiple Processors for Parallel Compilation" on page 10–65 ■ Removed "Analyze Your Design for Megastability" ■ Updated Table 10–11 and Table 10–9 ■ Removed Tables 8-1, 8-2, 8-3, 8-6, and 8-7 from version 8.1	Updated for the Quartus II 9.0 software release. Added Arria II GX support. Reorganized portions of this chapter.

 © March 2009 Altera Corporation

Table 10–12. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
November 2008 v8.1.0	■ Changed document to 8½” × 11” page size. ■ Updated the following sections: →“Optimizing Your Design” on page 10–2 →“Timing Requirement Settings” on page 10–4 →“Optimize Hold Timing” on page 10–8 →“Limit to One Fitting Attempt” on page 10–9 →“Auto Fit” on page 10–10 →“Fast Fit” on page 10–11 →“Ignored Timing Assignments” on page 10–12 →“I/O Timing (Including tPD)” on page 10–13 →“Register-to-Register Timing” on page 10–14 →“Timing Analysis with the TimeQuest Timing Analyzer” on page 10–14 →“Use I/O Assignment Analysis” on page 10–20 →“Flatten the Hierarchy During Synthesis” on page 10–25 →“Retarget Memory Blocks” on page 10–25 →“Use Physical Synthesis Options to Reduce Area” on page 10–26 →“Increase Placement Effort Multiplier” on page 10–30 →“Metastability Analysis and Optimization Techniques” on page 10–32 →“Synthesis Netlist Optimizations and Physical Synthesis Optimizations” on page 10–43 →“Incremental Compilation” on page 10–65 →“Use Multiple Processors for Parallel Compilation” on page 10–66 ■ Updated Table 10–9 on page 10–73 and Table 10–11 on page 10–75.	Updated for the Quartus II 8.1 software release.
May 2008 v8.0.0	■ Updated links ■ Updated Other Optimization Resources] ■ Updated Setting Process Priority ■ Updated Location Assignment and Back-Annotation ■ Updated Fitter Effort Setting ■ Updated Synthesis Netlist Optimizations and Physical Synthesis Optimizations ■ Updated Fast Fit ■ Added Metastability Analysis ■ Added Enable Beneficial Skew Optimization and Analyze Your Design for Metastability ■ Removed figures from “Optimizing Source Code—Pipelining for Complex Register Logic ■ Updated Table 8-5	Changes made to this chapter reflect the software changes made in version 8.0. Removed support for Mercury devices. Added information for Stratix IV devices.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

11. Power Optimization

QII52016-9.0.0

Introduction

The Quartus® II software offers power-driven compilation to fully optimize device power consumption. Power-driven compilation focuses on reducing your design's total power consumption using power-driven synthesis and power-driven place-and-route. This chapter describes the power-driven compilation feature and flow in detail, as well as low power design techniques that can further reduce power consumption in your design. The techniques primarily target Arria® GX, Stratix® and Cyclone® series of devices, and HardCopy® II devices. These devices utilize a low-k dielectric material that dramatically reduces dynamic power and improves performance. The latest Stratix device families include new, more efficient, logic structures called adaptive logic modules (ALMs) that obtain maximum performance while minimizing power consumption. Cyclone device families offer the optimal blend of high performance and low power in a low-cost FPGA.

For more information about Stratix IV and Stratix III device architecture, refer to the *Stratix IV Device Handbook* and *Stratix III Device Handbook*, respectively.

Altera provides the Quartus II PowerPlay Power Analyzer to aid you during the design process by delivering fast and accurate estimations of power consumption. You can minimize power consumption, while taking advantage of the industry's leading FPGA performance, by using the tools and techniques described in this chapter.

For more information about the PowerPlay Power Analyzer, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Total FPGA power consumption is comprised of I/O power, core static power, and core dynamic power. This chapter focuses on design optimization options and techniques that help reduce core dynamic power and I/O power. In addition to these techniques, there are additional power optimization techniques available for Stratix IV and Stratix III devices. These techniques include:

- Selectable Core Voltage (available only for Stratix III devices)
- Programmable Power Technology
- Device Speed Grade Selection

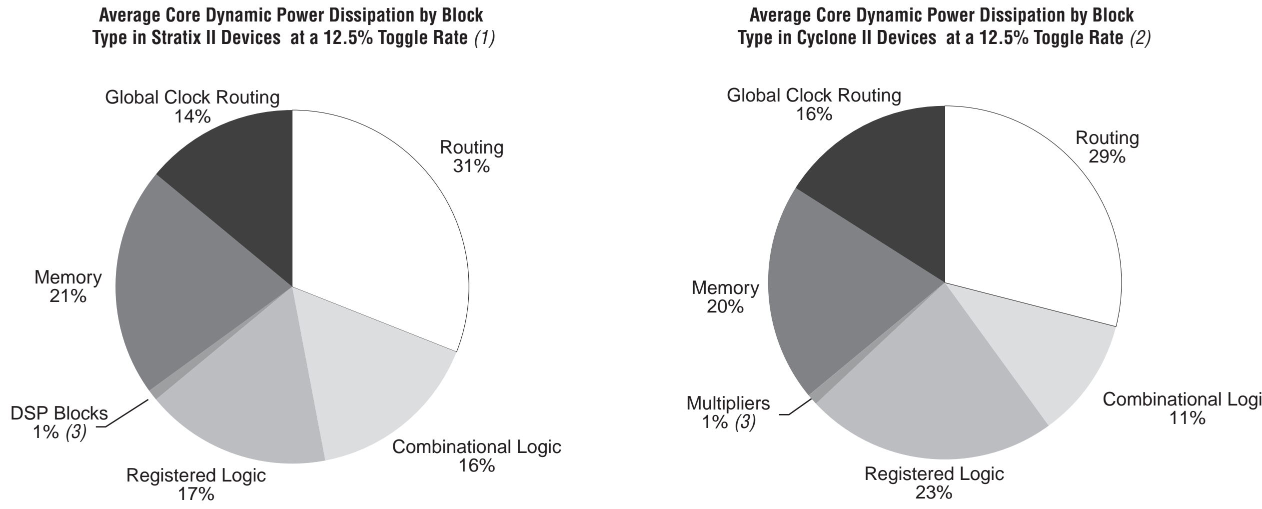
For more information about power optimization techniques available for Stratix III devices, refer to *AN 437: Power Optimization in Stratix III FPGAs*. For more information about power optimization techniques available for Stratix IV devices, refer to *AN 514: Power Optimization in Stratix IV FPGAs*.

Power Dissipation

This section describes the sources of power dissipation in Stratix II and Cyclone II devices. You can refine techniques that reduce power consumption in your design by understanding the sources of power dissipation.

Figure 11–1 shows the power dissipation of Stratix III and Cyclone III devices in different designs. All designs were analyzed at a fixed clock rate of 100 MHz and exhibited varied logic resource utilization across available resources.

Figure 11–1. Average Core Dynamic Power Dissipation



Notes to Figure 11–1:

(1) 103 different designs were used to obtain these results.
(2) 96 different designs were used to obtain these results.
(3) In designs using DSP blocks, DSPs consumed 5% of core dynamic power.

As shown in Figure 11–1, a significant amount of the total power is dissipated in routing for both Stratix III and Cyclone III devices, with the remaining power dissipated in logic, clock, and RAM blocks.

In Stratix and Cyclone device families, a series of column and row interconnect wires of varying lengths provide signal interconnections between logic array blocks (LABs), memory block structures, and digital signal processing (DSP) blocks or multiplier blocks. These interconnects dissipate the largest component of device power.

FPGA combinational logic is another source of power consumption. The basic building block of logic in the latest Stratix series devices is the ALM, and in Cyclone III and Cyclone II devices, it is the logic element (LE).

For more information about ALMs and LEs in Stratix IV, Stratix III, Stratix II, Cyclone III, and Cyclone II devices, refer to the respective device handbook.

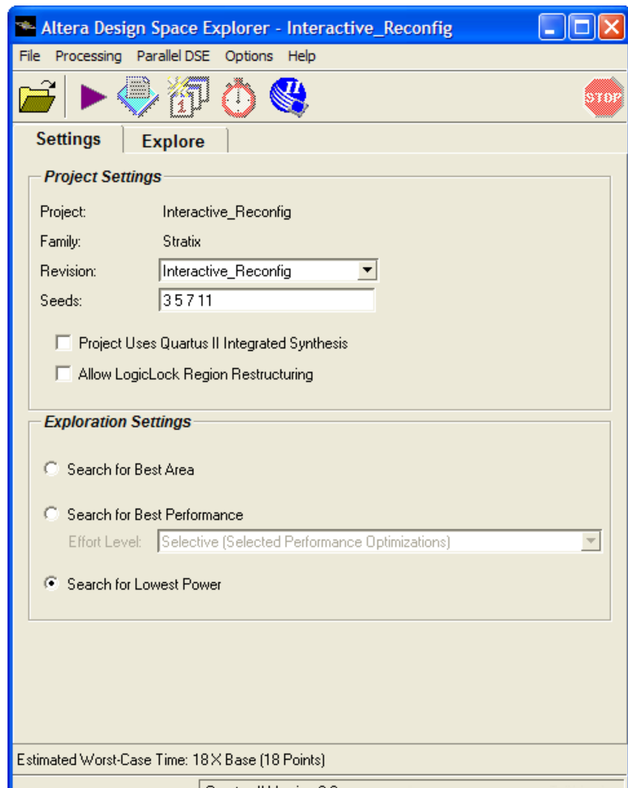
Memory and clock resources are other major consumers of power in FPGAs. Stratix II devices feature the TriMatrix memory architecture. TriMatrix memory includes 512-bit M512 blocks, 4-Kbit M4K blocks, and 512-Kbit M-RAM blocks, which are configurable to support many features. Stratix IV and Stratix III TriMatrix on-chip memory is an enhancement based upon the Stratix II FPGA TriMatrix memory and includes three sizes of memory blocks: MLAB blocks, M9K blocks, and M144K blocks. Cyclone II devices have 4-Kbit M4K memory blocks and Cyclone III devices have 9-Kbit M9K memory blocks.

Design Space Explorer

The Design Space Explorer (DSE) is a simple, easy-to-use, design optimization utility that is included in the Quartus II software. The DSE explores and reports optimal Quartus II software options for your design, targeting either power optimization, design performance, or area utilization improvements. You can use the DSE to implement the techniques described in this chapter.

Figure 11–2 shows the DSE user interface. The **Settings** tab is divided into **Project Settings** and **Exploration Settings**.

Figure 11–2. Design Space Explorer User Interface



The **Search for Lowest Power** option, under **Exploration Settings**, uses a predefined exploration space that targets overall design power improvements. This setting focuses on applying different options that specifically reduce total design thermal power. You can also set the **Optimization Goal** option for your design using the **Advanced** tab in the DSE window. You can select your design optimization goal, such as optimize for power, from the list of available settings in the **Optimization Goal** list. The DSE then uses the selection from the **Optimization Goal** list, along with the **Search for Lowest Power** selection, to determine the best compilation results.

By default, the Quartus II PowerPlay Power Analyzer is run for every exploration performed by the DSE when the **Search for Lowest Power** option is selected. This helps you debug your design and determine trade-offs between power requirements and performance optimization.

For more information about the DSE, refer to the *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*.

Power-Driven Compilation

The standard Quartus II compilation flow consists of Analysis and Synthesis, Fitter, Assembler, and Timing Analysis. Power-driven compilation takes place at the analysis and synthesis and Fitter levels. Power-driven compilation settings are divided in the **PowerPlay power optimization** list on the **Analysis & Synthesis Settings** page, and **PowerPlay power optimization** on the **Fitter Settings** page. The following section describes these power optimization options at the analysis and synthesis and Fitter levels.

Power-Driven Synthesis

Synthesis netlist optimization occurs during the synthesis stage of the compilation flow. The optimization technique makes changes to the synthesis netlist to optimize your design according to the selection of area, speed, or power optimization. This section describes power optimization techniques at the synthesis level.

The **Analysis & Synthesis Settings** page allows you to specify logic synthesis options. The **PowerPlay power optimization** option is available for the Arria GX, Stratix and Cyclone families of devices, and MAX® II devices (Figure 11–3).

To perform power optimization at the synthesis level in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis**. The **Analysis & Synthesis** page appears.
3. In the **PowerPlay power optimization** list, select your preferred setting. This option determines how aggressively Analysis and Synthesis optimizes the design for power.

Figure 11–3. Analysis & Synthesis Settings Page

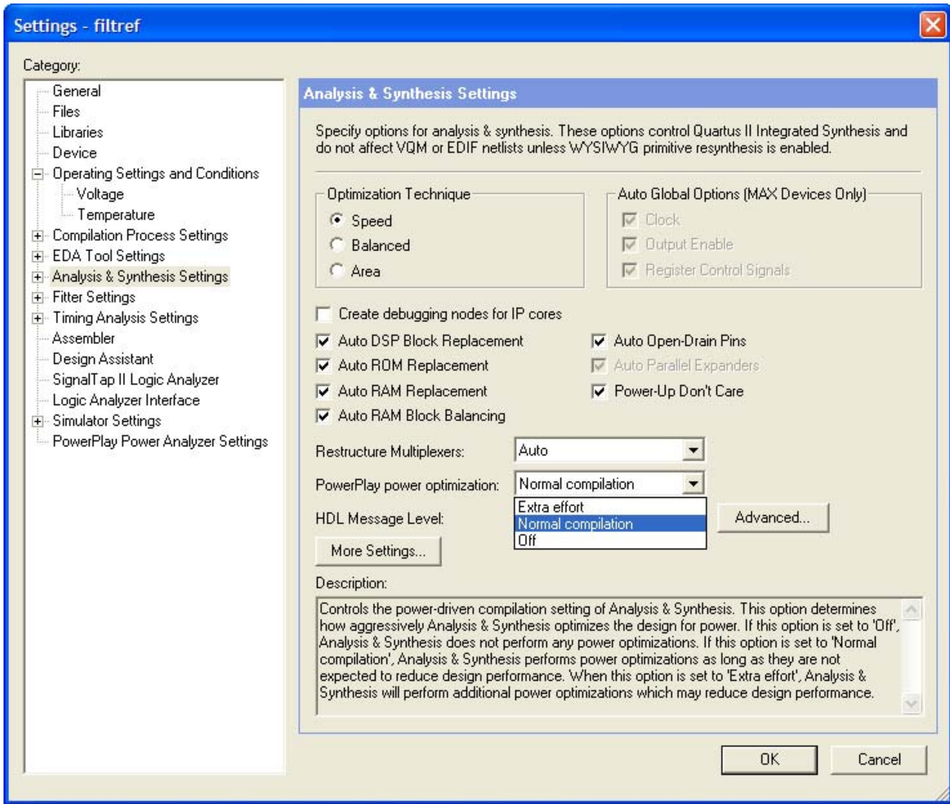


Table 11–1 shows the settings in the **PowerPlay power optimization** list. You can apply these settings on a project or entity level.

Table 11–1. Optimize Power During Synthesis Options

Settings	Description
Off	No netlist, placement, or routing optimizations are performed to minimize power.
Normal compilation (Default)	Low compute effort algorithms are applied to minimize power through netlist optimizations as long as they are not expected to reduce design performance.
Extra effort	High compute effort algorithms are applied to minimize power through netlist optimizations. Max performance might be impacted.

The **Normal compilation** setting is turned on by default. This setting performs memory optimization and power-aware logic mapping during synthesis.

Memory blocks can represent a large fraction of total design dynamic power as described in "Reducing Memory Power Consumption" on page 11–14. Minimizing the number of memory blocks accessed during each clock cycle can significantly reduce memory power. Memory optimization involves effective movement of user-defined read/write enable signals to associated read-and-write clock enable signals for all memory types (Figure 11–4).

Figure 11–4. Memory Transformation

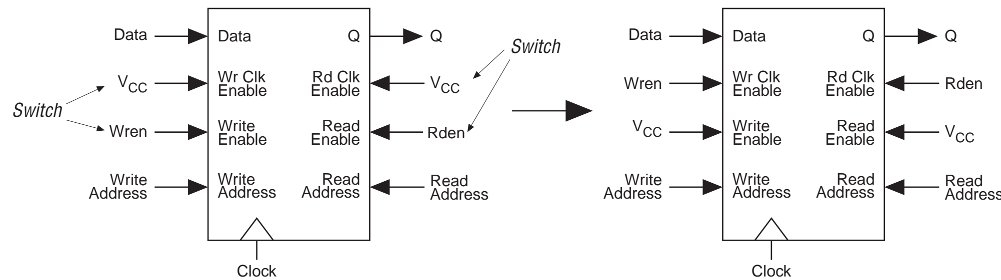


Figure 11–4 shows a default implementation of a simple dual-port memory block in which write-clock enable and read-clock enable signals are connected to V_{CC}, making both read-and-write memory ports active during each clock cycle. Memory transformation effectively moves the read-enable and write-enable signals to the respective read-clock enable and write-clock enable signals. By using this technique, memory ports are shut down when they are not accessed. This significantly reduces your design's memory power consumption. For more information about clock enable signals, refer to "Reducing Memory Power Consumption" on page 11–14. For Stratix IV and Stratix III devices, the memory transformation takes place at the Fitter level by selecting the **Normal compilation** settings for the power optimization option.

In Stratix III and Cyclone III devices, the specified read-during-write behavior can significantly impact the power of single-port and bidirectional dual-port RAMs. It is best to set the read-during-write parameter to "Don't care" (at the HDL level), as it allows an optimization whereby the `read-enable` signal can be set to the inversion of the existing `write-enable` signal (if one exists). This allows the core of the RAM to shut down (that is, not toggle), which saves a significant amount of power.

The other type of power optimization that takes place with the **Normal compilation** setting is power-aware logic mapping. The power-aware logic mapping reduces power by rearranging the logic during synthesis to eliminate nets with high toggle rates.

The **Extra effort** setting performs the functions of the **Normal compilation** setting and other memory optimizations to further reduce memory power by shutting down memory blocks that are not accessed. This level of memory optimization can require extra logic, which can reduce design performance.

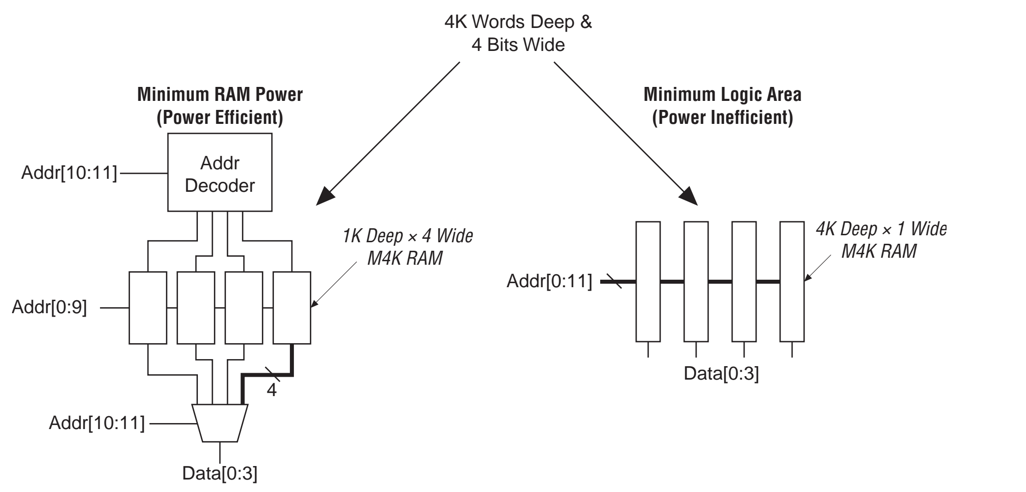
The **Extra effort** setting also performs power-aware memory balancing. Power-aware memory balancing automatically chooses the best memory configuration for your memory implementation and provides optimal power saving by determining the number of memory blocks, decoder, and multiplexer circuits required. If you have not previously specified target-embedded memory blocks for your design's memory functions, the power-aware balancer automatically selects them during memory implementation.

Figure 11–5 shows an example of a 4K × 4 (4K deep and 4 bits wide) memory implementation in two different configurations using M4K memory blocks available in Stratix II devices. The minimum logic area implementation uses M4K blocks configured as 4K × 1. This implementation is the default in the Quartus II software because it has the minimum logic area (0 logic cells) and the highest speed. However,

all four M4K blocks are active on each memory access in this implementation, which increases RAM power. The minimum RAM power implementation is created by selecting **Extra effort** in the **PowerPlay power optimization** list. This implementation automatically uses four M4K blocks configured as 1K × 4 for optimal power saving. An address decoder is implemented by the RAM megafunction to select which of the four M4K blocks should be activated on a given cycle, based on the state of the top two user address bits. The RAM megafunction automatically implements a multiplexer to feed the downstream logic by choosing the appropriate M4K output. This implementation reduces RAM power because only one M4K block is active on any cycle, but it requires extra logic cells, costing logic area and potentially impacting design performance.

There is a trade-off between power saved by accessing fewer memories and power consumed by the extra decoder and multiplexor logic. The Quartus II software automatically balances the power savings against the costs to choose the lowest power configuration for each logical RAM. The benchmark data shows that the power-driven synthesis can reduce memory power consumption by as much as 60% in Stratix devices.

Figure 11–5. 4K × 4 Memory Implementation Using Multiple M4K Blocks



Memory optimization options can also be controlled by the `Low_Power_Mode` parameter in the **Default Parameters** page of the **Settings** dialog box. The settings for this parameter are **None**, **Auto**, and **ALL**. **None** corresponds to the **Off** setting in the **PowerPlay power optimization** list. **Auto** corresponds to the **Normal compilation** setting and **ALL** corresponds to the **Extra effort** setting, respectively. You can apply PowerPlay power optimization either on a compiler basis or on individual entities. The `Low_Power_Mode` parameter always takes precedence over the **Optimize Power for Synthesis** option for power optimization on memory.

You can also set the MAXIMUM_DEPTH parameter manually to configure the memory for low power optimization. This technique is the same as the power-aware memory balancer, but it is manual rather than automatic like the **Extra effort** setting in the **PowerPlay power optimization** list. You can set the MAXIMUM_DEPTH parameter for memory modules manually in the megafunction instantiation or in the MegaWizard™ Plug-In Manager for power optimization as described in "Reducing Memory Power Consumption" on page 11–14. The MAXIMUM_DEPTH parameter always takes precedence over the **Optimize Power for Synthesis** options for power optimization on memory optimization.

Power-Driven Fitter

The **Fitter Settings** page enables you to specify options for fitting (Figure 11–6). The **PowerPlay power optimization** option is available for Arria GX, Stratix IV, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II devices.

To perform power optimization at the Fitter level, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Fitter Settings**. The **Fitter Settings** page appears.
3. In the **PowerPlay power optimization** list, select your preferred setting. This option determines how aggressively the Fitter optimizes the design for power.

Figure 11–6. Fitter Settings Page

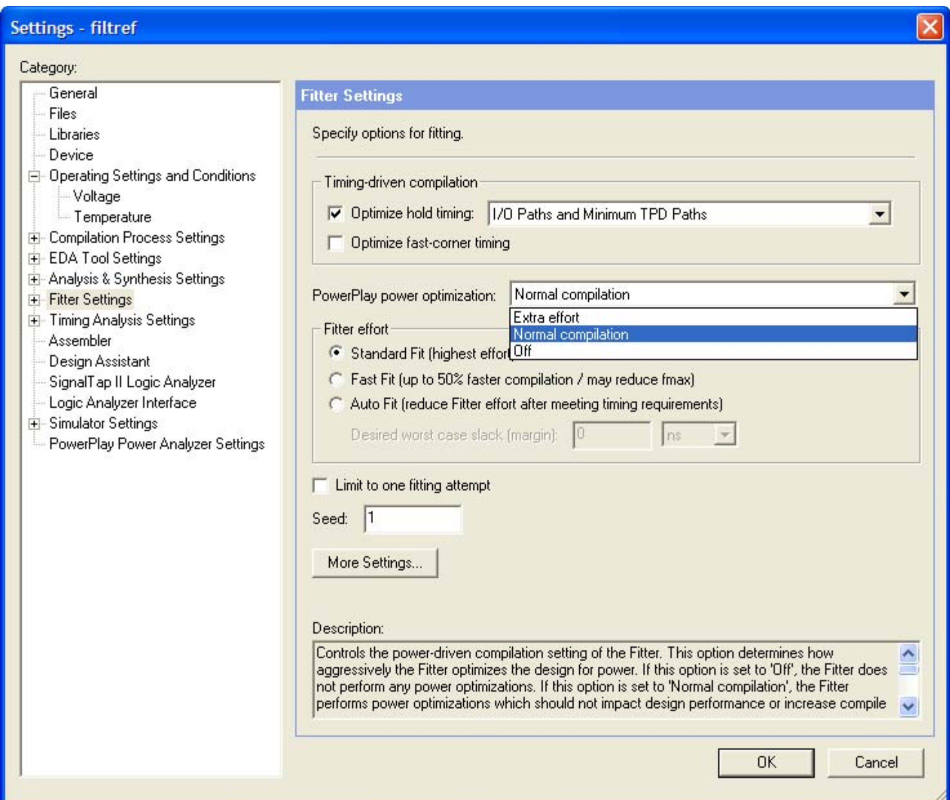


Table 11–2 lists the settings in the **PowerPlay power optimization** list. These settings can only be applied on a project-wide basis. The **Extra effort** setting for the Fitter requires extensive effort to optimize the design for power and can increase the compilation time.

Table 11–2. Power-Driven Fitter Option

Settings	Description
Off	No netlist, placement, or routing optimizations are performed to minimize power.
Normal compilation (Default)	Low compute effort algorithms are applied to minimize power through placement and routing optimizations as long as they are not expected to reduce design performance.
Extra effort	High compute effort algorithms are applied to minimize power through placement and routing optimizations. Max performance might be impacted.

The **Normal compilation** setting is selected by default and performs DSP optimization by creating power-efficient DSP block configurations for your DSP functions. For Stratix III devices, this setting, which is based on timing constraints entered for the design, enables the Programmable Power Technology to configure tiles as high-speed mode or low-power mode. Programmable Power Technology is always turned **ON** even when the **OFF** setting is selected for the **Fitter PowerPlay power optimization** option. Tiles are the combination of LAB and MLAB pairs (including the adjacent routing associated with LAB and MLAB), which can be configured to operate in high-speed or low-power mode. This level of power optimization does not have any affect on the fitting, timing results, or compile time. Also, for Stratix III devices, this setting enables the memory transformation as described in "Power-Driven Synthesis" on page 11–4.

For more information about Stratix III power optimization, refer to *AN 437: Power Optimization in Stratix III FPGAs*. For more information about Stratix IV power optimization, refer to *AN 514: Power Optimization in Stratix IV FPGAs*.

The **Extra effort** setting performs the functions of the **Normal compilation** setting and other place-and-route optimizations during fitting to fully optimize the design for power. The Fitter applies an extra effort to minimize power even after timing requirements have been met by effectively moving the logic closer during placement to localize high-toggling nets, and using routes with low capacitance. However, this effort can increase the compilation time.

The **Extra effort** setting uses a Signal Activity File (**.saf**) or Verilog Value Change Dump File (**.vcd**) that guides the Fitter to fully optimize the design for power, based on the signal activity of the design. The best power optimization during fitting results from using the most accurate signal activity information. Signal activities from full post-fit netlist (timing) simulation provide the highest accuracy because all node activities reflect the actual design behavior, provided that supplied input vectors are representative of typical design operation. If you do not have a **.saf** file (from simulation or other source), the Quartus II software uses assignments, clock assignments, and vectorless estimation values (PowerPlay Power Analyzer Tool settings) to estimate the signal activities. This information is used to optimize your design for power during fitting. The benchmark data shows that the power-driven Fitter technique can reduce power consumption by as much as 19% in Stratix devices.

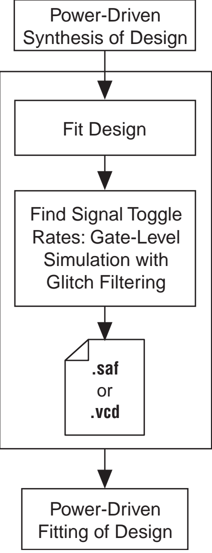
Only the **Extra effort** setting in the **PowerPlay power optimization** list for the Fitter option uses the signal activities (from **.vcd** or **.saf** file) during fitting. The settings made in the **PowerPlay Power Analyzer Settings** page in the **Settings** dialog box are used to calculate the signal activity of your design.

For more information about **.saf** and **.vcd** files, and how to create them, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Recommended Flow for Power-Driven Compilation

Figure 11–7 shows the recommended design flow to fully optimize your design for power during compilation. This flow utilizes the power-driven synthesis and power-driven Fitter options. On average, you can reduce core dynamic power by 16% with the extra effort synthesis and extra effort fitting settings, as compared to the **Off** settings in both synthesis and Fitter options for power-driven compilation.

Figure 11–7. Recommended Flow for Power-Driven Compilation



Area-Driven Synthesis

Using area optimization rather than timing or delay optimization during synthesis saves power because you use fewer logic blocks. Using less logic usually means less switching activity. The Quartus II integrated synthesis tool provides **Speed**, **Balanced**, or **Area** for the **Optimization Technique** option. You can also specify this logic option for specific modules in your design with the Assignment Editor in cases where you want to reduce area using the **Area** setting (potentially at the expense of register-to-register timing performance) while leaving the default **Optimization Technique** setting at **Balanced** (for the best trade-off between area and speed for certain device families). The **Speed Optimization Technique** can increase the resource usage of your design if the constraints are too aggressive, and can also result in increased power consumption.

The benchmark data shows that the area-driven technique can reduce power consumption by as much as 31% in Stratix devices and as much as 15% in Cyclone devices.

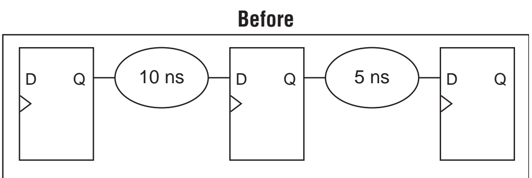
Gate-Level Register Retiming

You can also use gate-level register retiming to reduce circuit switching activity. Retiming shuffles registers across combinational blocks without changing design functionality. The **Perform gate-level register retiming** option in the Quartus II software enables the movement of registers across combinational logic to balance timing, allowing the software to trade off the delay between timing critical and non-critical timing paths.

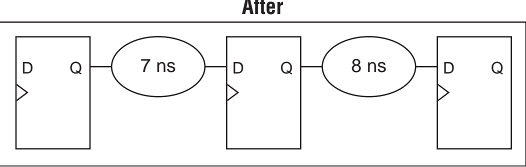
Retiming uses fewer registers than pipelining. Figure 11–8 shows an example of gate-level register retiming, where the 10 ns critical delay is reduced by moving the register relative to the combinational logic, resulting in the reduction of data depth and switching activity.

Figure 11–8. Gate-Level Register Retiming

Before



After



Gate-level register retiming makes changes at the gate level. If you are using an atom netlist from a third-party synthesis tool, you must also select the **Perform WYSIWYG primitive resynthesis** option to undo the atom primitives to gates mapping (so that register retiming can be performed), and then to remap gates to Altera® primitives. When using the Quartus II integrated synthesis, retiming occurs during synthesis before the design is mapped to Altera primitives. The benchmark data shows that the combination of WYSIWYG remapping and gate-level register retiming techniques can reduce power consumption by as much as 6% in Stratix devices and as much as 21% in Cyclone devices.

For more information about register retiming, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

Design Guidelines

Several low-power design techniques can reduce power consumption when applied during FPGA design implementation. This section provides detailed design techniques for Stratix III, Stratix II, Cyclone III, and Cyclone II devices that affect overall design power. The results of these techniques might be different from design to design.

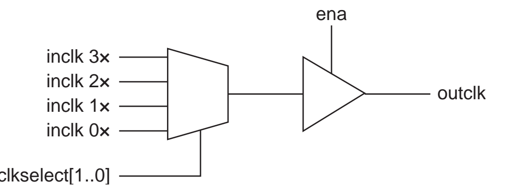
Clock Power Management

Clocks represent a significant portion of dynamic power consumption due to their high switching activity and long paths. Figure 11–1 on page 11–2 shows a 14% average contribution to power consumption for global clock routing in Stratix III devices and 17% in Cyclone III devices. Actual clock-related power consumption is higher than this because the power consumed by local clock distribution within logic, memory, and DSP or multiplier blocks is included in the power consumption for the respective blocks.

Clock routing power is automatically optimized by the Quartus II software, which only enables those portions of the clock network that are required to feed downstream registers. Power can be further reduced by gating clocks when they are not required. It is possible to build clock-gating logic, but this approach is not recommended because it is difficult to generate a glitch-free clock in FPGAs using ALMs or LEs.

Arria GX, Stratix IV, Stratix III, Stratix II, Cyclone III, and Cyclone II devices use clock control blocks that include an enable signal. A clock control block is a clock buffer that lets you dynamically enable or disable the clock network and dynamically switch between multiple sources to drive the clock network. You can use the Quartus II MegaWizard Plug-In Manager to create this clock control block with the ALTCLKCTRL megafunction. Arria GX, Stratix IV, Stratix III, Stratix II, Cyclone III, and Cyclone II devices provide clock control blocks for global clock networks. In addition, Stratix IV, Stratix III and Stratix II devices have clock control blocks for regional clock networks. The dynamic clock enable feature lets internal logic control the clock network. When a clock network is powered down, all the logic fed by that clock network does not toggle, thereby reducing the overall power consumption of the device. Figure 11–9 shows a 4-input clock control block diagram.

Figure 11–9. Clock Control Block Diagram

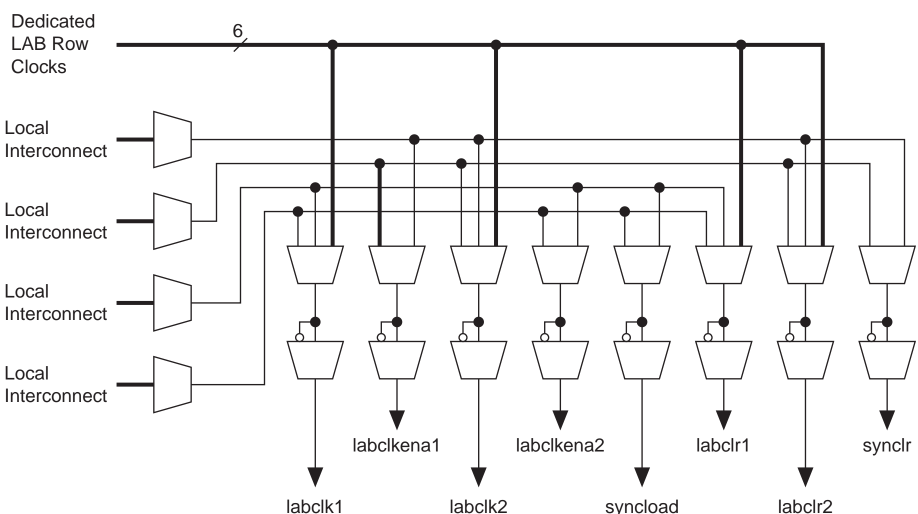


The enable signal is applied to the clock signal before being distributed to global routing. Therefore, the enable signal can either have a significant timing slack (at least as large as the global routing delay) or it can reduce the f_{MAX} of the clock signal.

For more information about using clock control blocks, refer to the *Clock Control Block Megafunction User Guide (ALTCLKCTRL)*.

Another contributor to clock power consumption is the LAB clock that distributes a clock to the registers within a LAB. LAB clock power can be the dominant contributor to overall clock power. For example, in Cyclone III and Cyclone II devices, each LAB can use two clocks and two clock enable signals, as shown in Figure 11–10. Each LAB's clock signal and clock enable signal are linked. For example, an LE in a particular LAB using the `labclk1` signal also uses the `labclkena1` signal.

Figure 11–10. LAB-Wide Control Signals



To reduce LAB-wide clock power consumption without disabling the entire clock tree, use the LAB-wide clock enable to gate the LAB-wide clock. The Quartus II software automatically promotes register-level clock enable signals to the LAB-level. All registers within an LAB that share a common clock and clock enable are controlled by a shared gated clock. To take advantage of these clock enables, use a clock enable construct in the relevant HDL code for the registered logic.

LAB-Wide Clock Enable Example

The VHDL code in Example 11–1 makes use of a LAB-wide clock enable. This clock-gating logic is automatically turned into an LAB-level clock enable signal.

Example 11–1.

```
IF clk'event AND clock = '1' THEN
    IF logic_is_enabled = '1' THEN
        reg <= value;
    ELSE
        reg <= reg;
    END IF;
END IF;
```

For more information about LAB-wide control signals, refer to the *Stratix II Architecture*, *Cyclone III Device Family Overview*, or *Cyclone II Architecture* chapters in the respective device handbook.

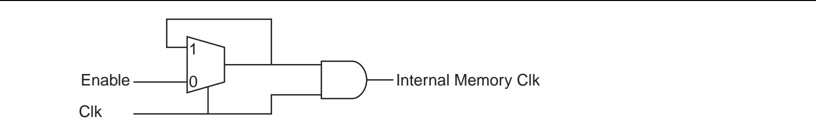
Reducing Memory Power Consumption

The memory blocks in FPGA devices can represent a large fraction of typical core dynamic power. Memory represents 21% of the core dynamic power in a typical Stratix III device design and 20% in a Cyclone III device design. Memory blocks are unlike most other blocks in the device because most of their power is tied to the clock rate, and is insensitive to the toggle rate on the data and address lines.

When a memory block is clocked, there is a sequence of timed events that occur within the block to execute a read or write. The circuitry controlled by the clock consumes the same amount of power regardless of whether or not the address or data has changed from one cycle to the next. Thus, the toggle rate of input data and the address bus have no impact on memory power consumption.

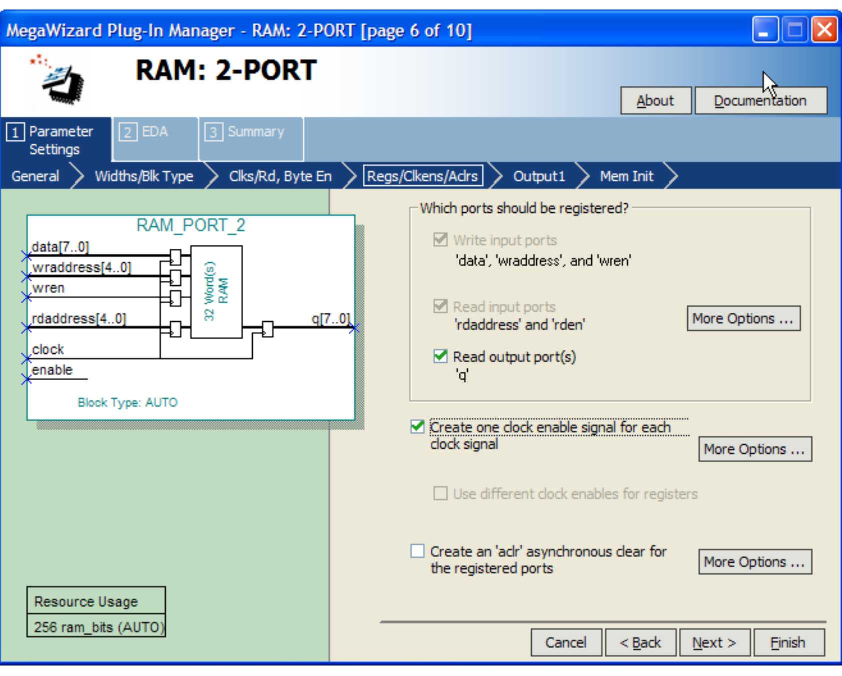
The key to reducing memory power consumption is to reduce the number of memory clocking events. You can achieve this through clock network-wide gating described in "Clock Power Management" on page 11–12, or on a per-memory basis through use of the clock enable signals on the memory ports. Figure 11–11 shows the logical view of the internal clock of the memory block. Use the appropriate enable signals on the memory to make use of the clock enable signal instead of gating the clock.

Figure 11–11. Memory Clock Enable Signal



Using the clock enable signal enables the memory only when necessary and shuts it down for the rest of the time, reducing the overall memory power consumption. You can use the MegaWizard Plug-In Manager to create these enable signals by selecting the **Clock enable signal** option for the appropriate port when generating the memory block function (Figure 11–12).

Figure 11–12. MegaWizard Plug-In Manager RAM 2-Port Clock Enable Signal Selectable Option

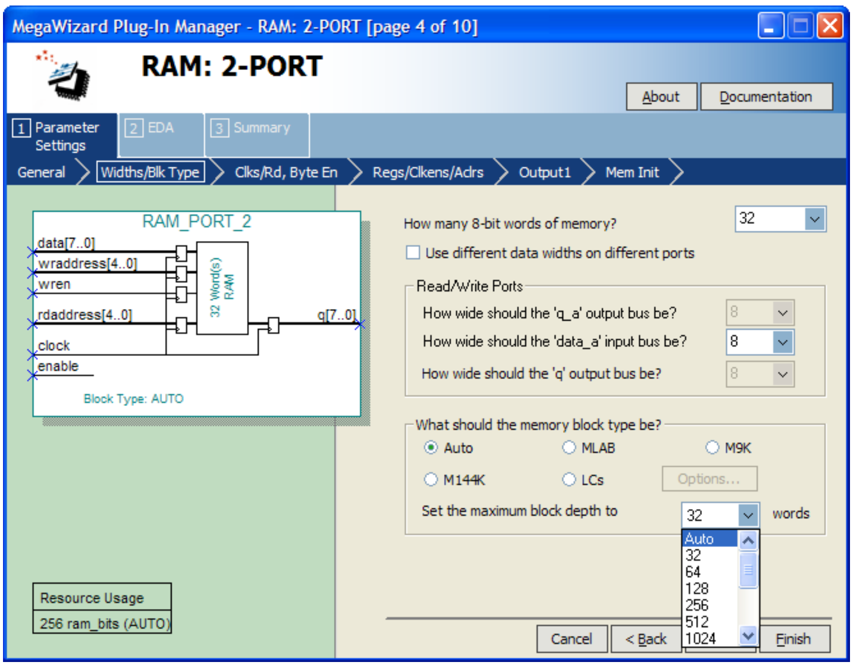


For example, consider a design that contains a 32-bit-wide M4K memory block in ROM mode that is running at 200 MHz. Assuming that the output of this block is only required approximately every four cycles, this memory block will consume 8.45 mW of dynamic power according to the demands of the downstream logic. By adding a small amount of control logic to generate a read clock enable signal for the memory block only on the relevant cycles, the power can be cut 75% to 2.15 mW.

You can also use the `MAXIMUM_DEPTH` parameter in your memory megafunction to save power in Stratix IV, Stratix III, Stratix II, Cyclone III, and Cyclone II devices; however, this approach might increase the number of LEs required to implement the memory and affect design performance.

You can set the `MAXIMUM_DEPTH` parameter for memory modules manually in the megafunction instantiation or in the MegaWizard Plug-In Manager (Figure 11–13). The Quartus II software automatically chooses the best design memory configuration for optimal power, as described in "Power-Driven Compilation" on page 11–4.

Figure 11–13. MegaWizard Plug-In Manager RAM 2-Port Maximum Depth Selectable Option



Memory Power Reduction Example

Table 11–3 shows power usage measurements for a 4K × 36 simple dual-port memory implemented using multiple M4K blocks in a Stratix II EP2S15 device. For each implementation, the M4K blocks are configured with a different memory depth.

Table 11–3. 4K × 36 Simple Dual-Port Memory Implemented Using Multiple M4K Blocks

M4K Configuration	Number of M4K Blocks	ALUTs
4K × 1 (Default setting)	36	0
2K × 2	36	40
1K × 4	36	62
512 × 9	32	143
256 × 18	32	302
128 × 36	32	633

Figure 11–14 shows the amount of power saved using the `MAXIMUM_DEPTH` parameter. For all implementations, a user-provided read enable signal is present to indicate when read data is required. Using this power-saving technique can reduce power consumption by as much as 60%.

Figure 11–14. Power Savings Using the MAXIMUM_DEPTH Parameter

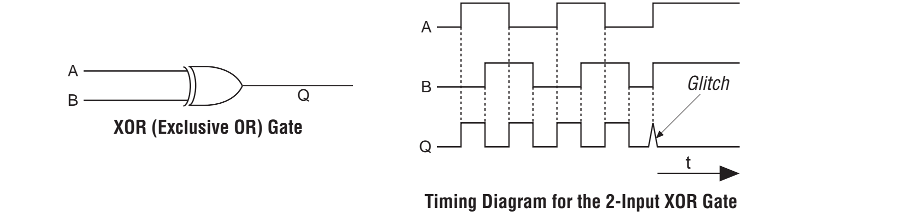
As the memory depth becomes more shallow, memory dynamic power decreases because unaddressed M4K blocks can be shut off using a decoded combination of address bits and the read enable signal. For a 128-deep memory block, power used by the extra LEs starts to outweigh the power gain achieved by using a more shallow memory block depth. The power consumption of the memory blocks and associated LEs depends on the memory configuration.

Pipelining and Retiming

Designs with many glitches consume more power because of faster switching activity. Glitches cause unnecessary and unpredictable temporary logic switches at the output of combinational logic. A glitch usually occurs when there is a mismatch in input signal timing leading to unequal propagation delay.

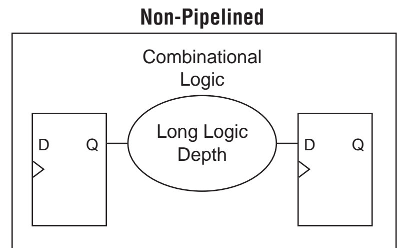
For example, consider an input change on one input of a 2-input XOR gate from `1` to `0`, followed a few moments later by an input change from `0` to `1` on the other input. For a moment, both inputs become `1` (high) during the state transition, resulting in `0` (low) at the output of the XOR gate. Subsequently, when the second input transition takes place, the XOR gate output becomes `1` (high). During signal transition, a glitch is produced before the output becomes stable, as shown in Figure 11–15. This glitch can propagate to subsequent logic and create unnecessary switching activity, increasing power consumption. Circuits with many XOR functions, such as arithmetic circuits or cyclic redundancy check (CRC) circuits, tend to have many glitches if there are several levels of combinational logic between registers.

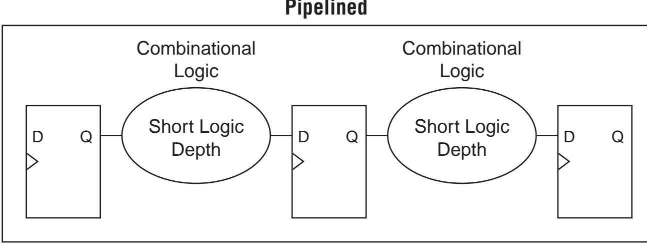
Figure 11–15. XOR Gate Showing Glitch at the Output



Pipelining can reduce design glitches by inserting flipflops into long combinational paths. Flipflops do not allow glitches to propagate through combinational paths. Therefore, a pipelined circuit tends to have less glitching. Pipelining has the additional benefit of generally allowing higher clock speed operations, although it does increase the latency of a circuit (in terms of the number of clock cycles to a first result). Figure 11–16 shows an example where pipelining is applied to break up a long combinational path.

Figure 11–16. Pipelining Example





Pipelining is very effective for glitch-prone arithmetic systems because it reduces switching activity, resulting in reduced power dissipation in combinational logic. Additionally, pipelining allows higher-speed operation by reducing logic-level numbers between registers. The disadvantage of this technique is that if there are not many glitches in your design, pipelining can increase power consumption by adding unnecessary registers. Pipelining can also increase resource utilization. The benchmark data shows that pipelining can reduce dynamic power consumption by as much as 31% in Stratix devices and as much as 30% in Cyclone devices.

Architectural Optimization

You can use design-level architectural optimization by taking advantage of specific device architecture features. These features include dedicated memory and DSP or multiplier blocks available in FPGA devices to perform memory or arithmetic-related functions. You can use these blocks in place of LUTs to reduce power consumption. For example, you can build large shift registers from RAM-based FIFO buffers instead of building the shift registers from the LE registers.

The Stratix device family allows you to efficiently target small, medium, and large memories with the TriMatrix memory architecture. Each TriMatrix memory block is optimized for a specific function. The M512 memory blocks available in Stratix II devices are useful for implementing small FIFO buffers, DSP, and clock domain transfer applications. M512 memory blocks are more power-efficient than the distributed memory structures in some competing FPGAs. The M4K memory blocks are used to implement buffers for a wide variety of applications, including processor code storage, large look-up table implementation, and large memory applications. The M-RAM blocks are useful in applications where a large volume of data must be stored on-chip. Effective utilization of these memory blocks can have a significant impact on power reduction in your design.

The latest Stratix and Cyclone device families have configurable M9K memory blocks that provide various memory functions such as RAM, FIFO buffers, and ROM.

For more information about using DSP and memory blocks efficiently, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

I/O Power Guidelines

Non-terminated I/O standards such as LVTTL and LVCMOS have a rail-to-rail output swing. The voltage difference between logic-high and logic-low signals at the output pin is equal to the V_{CCIO} supply voltage. If the capacitive loading at the output pin is known, the dynamic power consumed in the I/O buffer can be calculated as shown in Equation 11–1:

Equation 11–1.

$$P = 0.5 \times F \times C \times V^2$$

In this equation, *F* is the output transition frequency and *C* is the total load capacitance being switched. *V* is equal to V_{CCIO} supply voltage. Because of the quadratic dependence on V_{CCIO}, lower voltage standards consume significantly less dynamic power. In addition, lower pin capacitance is an important factor in considering I/O power consumption. Hardware and simulation data show that Stratix II device I/O pins have half the pin capacitance of the nearest competing FPGA. Cyclone II devices exhibit 20% less I/O power consumption than competitive, low-cost, 90 nm FPGAs.

Transistor-to-transistor logic (TTL) I/O buffers consume very little static power. As a result, the total power consumed by a LVTTL or LVCMOS output is highly dependent on load and switching frequency.

When using resistively terminated I/O standards like SSTL and HSTL, the output load voltage swings by a small amount around some bias point. The same dynamic power equation is used, where *V* is the actual load voltage swing. Because this is much smaller than V_{CCIO}, dynamic power is lower than for non-terminated I/O under similar conditions. These resistively terminated I/O standards dissipate significant static (frequency-independent) power, because the I/O buffer is constantly driving current into the resistive termination network. However, the lower dynamic power of these I/O standards means they often have lower total power than LVCMOS or LVTTL for high-frequency applications. Use the lowest drive strength I/O setting that meets your speed and waveform requirements to minimize I/O power when using resistively terminated standards.

You can save a small amount of static power by connecting unused I/O banks to the lowest possible V_{CCIO} voltage of 1.2 V.

Table 11–4 shows the total supply and thermal power consumed by outputs using different I/O standards for Stratix II devices. The numbers are for an I/O pin transmitting random data clocked at 200 MHz with a 10 pF capacitive load.

Table 11–4. I/O Power for Different I/O Standards in Stratix II Devices (Part 1 of 2)

Standard	Total Supply Current Drawn from V_{CCIO} Supply (mA)	Total On-Chip Thermal Power Dissipation (mW)
3.3-V LVTTL	2.42	9.87
2.5-V LVCMOS	1.9	6.69

Table 11–4. I/O Power for Different I/O Standards in Stratix II Devices (Part 2 of 2)

Standard	Total Supply Current Drawn from V_{CCIO} Supply (mA)	Total On-Chip Thermal Power Dissipation (mW)
1.8-V LVCMOS	1.34	4.18
1.5-V LVCMOS	1.18	3.58
3.3-V PCI	2.47	10.23
SSTL-2 class I	6.07	4.42
SSTL-2 class II	10.72	5.1
SSTL-18 class I	5.33	3.28
SSTL-18 class II	8.56	4.06
HSTL-15 class I	6.06	3.49
HSTL-15 class II	11.08	4.87
HSTL-18 class I	6.87	4.09
HSTL-18 class II	12.33	5.82

For this configuration, non-terminated standards generally use less power, but this is not always the case. If the frequency or the capacitive load is increased, the power consumed by non-terminated outputs increases faster than the power of terminated outputs.

For more information about I/O Standards, refer to the *Selectable I/O Standards in Stratix II Devices and Stratix II GX Devices* chapter in volume 2 of the *Stratix II Device Handbook* or the *Selectable I/O Standards in Cyclone II Devices* chapter in the *Cyclone II Device Handbook* or the *Cyclone III Device Handbook.*

When calculating I/O power, the PowerPlay Power Analyzer uses the default capacitive load set for the I/O standard in the **Capacitive Loading** tab of the **Device & Pin Options** dialog box. For Stratix II, if **Enable Advanced I/O Timing** is turned on, I/O power is measured using an equivalent load calculated as the sum of the near capacitance, the transmission line distributed capacitance, and the far-end capacitance as defined in the **Board Trace Model** tab of the **Device & Pin Options** dialog box or the Board Trace Model view in the Pin Planner. Any other components defined in the board trace model are not taken into account for the power measurement.

For Stratix IV, Stratix III, and Cyclone III devices, advanced I/O power, which uses the full board trace model, is always used.

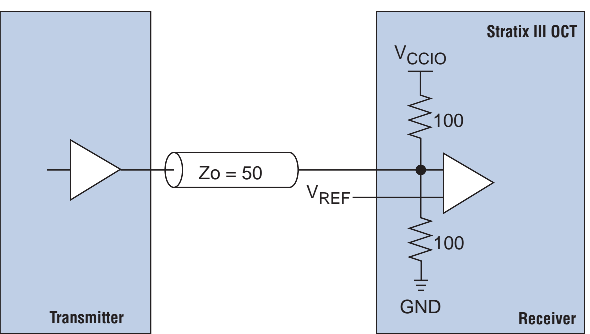
For information about using Advanced I/O Timing and configuring a board trace model, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Dynamically-Controlled On-Chip Terminations

Stratix IV and Stratix III FPGAs offer dynamic on-chip termination (OCT). Dynamic OCT enables series termination (RS) and parallel termination (RT) to dynamically turn on/off during the data transfer. This feature is especially useful when Stratix IV and Stratix III FPGAs are used with external memory interfaces, such as interfacing with DDR memories.

Compared to conventional termination, dynamic OCT reduces power consumption significantly as it eliminates the constant DC power consumed by parallel termination when transmitting data. Parallel termination is extremely useful for applications that interface with external memories where I/O standards, such as HSTL and SSTL, are used. Parallel termination supports dynamic OCT, which is useful for bidirectional interfaces (see Figure 11–17).

Figure 11–17. Stratix III On-Chip Parallel Termination



The following is an example of power saving for a DDR3 interface using on-chip parallel termination.

The static current consumed by parallel OCT is equal to the V_{CCIO} voltage divided by 100 Ω . For DDR3 interfaces that use SSTL-15, the static current is 1.5 V / 100 Ω = 15 mA per pin. Therefore, the static power is 1.5 V ×15 mA = 22.5 mW. For an interface with 72 DQ and 18 DQS pins, the static power is 90 pins × 22.5 mW = 2.025 W. Dynamic parallel OCT disables parallel termination during write operations, so if writing occurs 50% of the time, the power saved by dynamic parallel OCT is 50% × 2.025 W = 1.0125 W.

For more information about dynamic OCT in Stratix IV and Stratix III devices, refer to the *Stratix III Device I/O Features* chapter in the *Stratix III Device Handbook* and the *Stratix IV Device I/O Features* chapter in the *Stratix IV Device Handbook*, respectively.

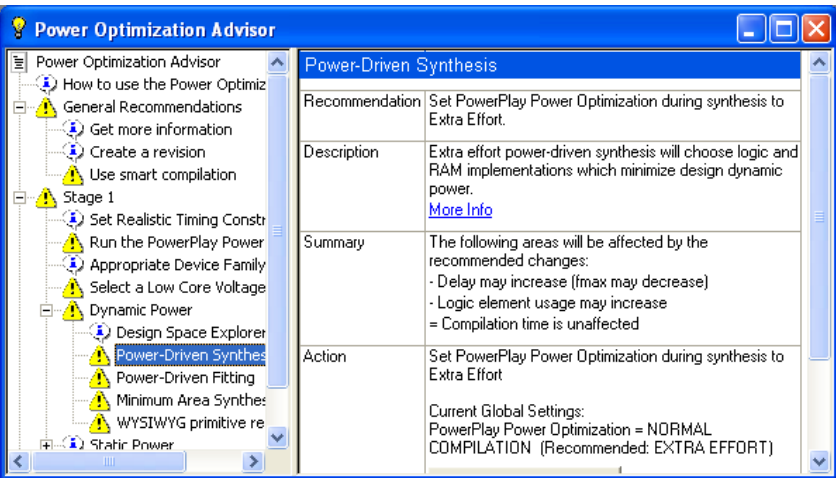
Power Optimization Advisor

The Quartus II software includes the Power Optimization Advisor, which provides specific power optimization advice and recommendations based on the current design project settings and assignments. The advisor covers many of the suggestions listed in this chapter. The following example shows how to reduce your design power with the Power Optimization Advisor.

Power Optimization Advisor Example

After compiling your design, run the PowerPlay Power Analyzer to determine your design power and to see where power is dissipated in your design. Based on this information, you can run the Power Optimization Advisor to implement recommendations that can reduce design power. Figure 11–18 shows the Power Optimization Advisor after compiling a design that is not fully optimized for power.

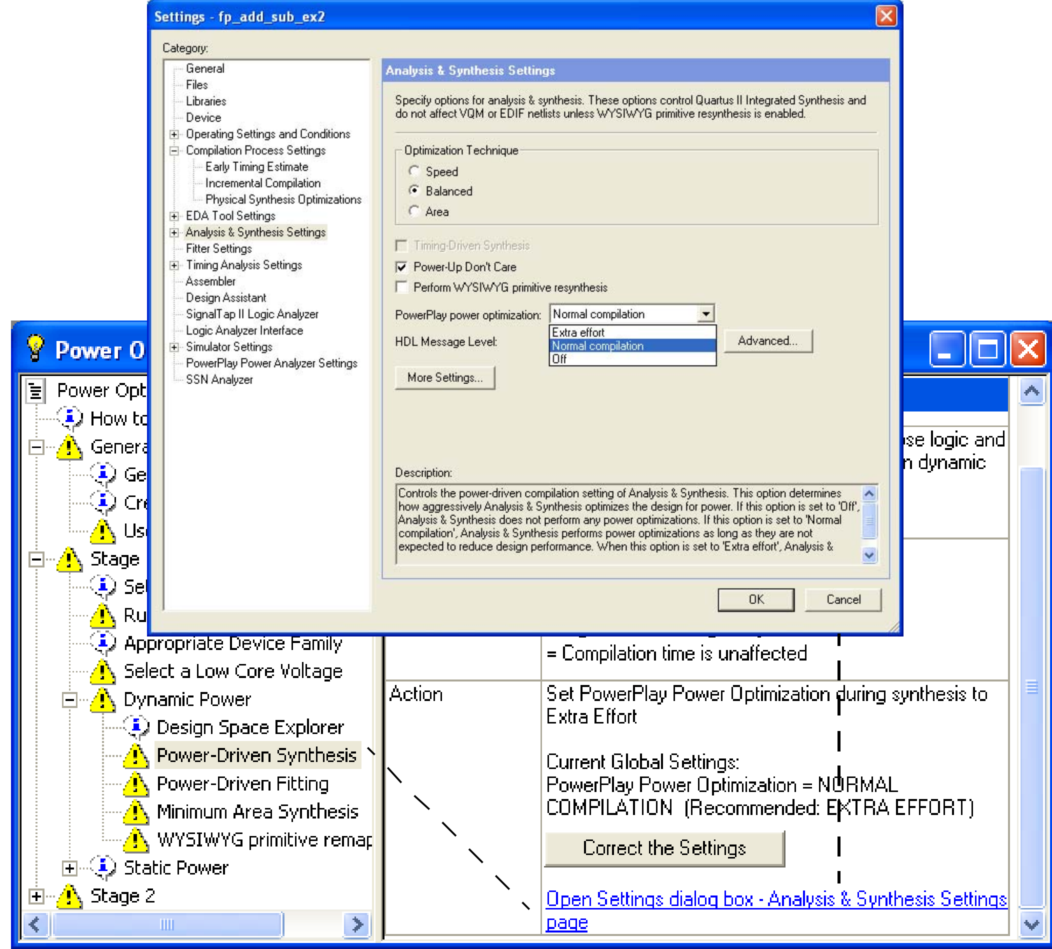
Figure 11–18. Power Optimization Advisor



The Power Optimization Advisor shows the recommendations that can reduce power in your design. The recommendations are split into stages to show the order in which you should apply the recommended settings. The first stage shows mostly CAS setting options that are easy to implement and highly effective in reducing design power. An icon indicates whether each recommended setting is made in the current project. In Figure 11–18, the checkmark icon for Stage 1 shows the recommendations that are already implemented. The warning icons indicate recommendations that are not followed for this compilation. The information icon shows the general suggestions. Each recommendation includes the description, summary of the affect of the recommendation, and the action required to make the appropriate setting.

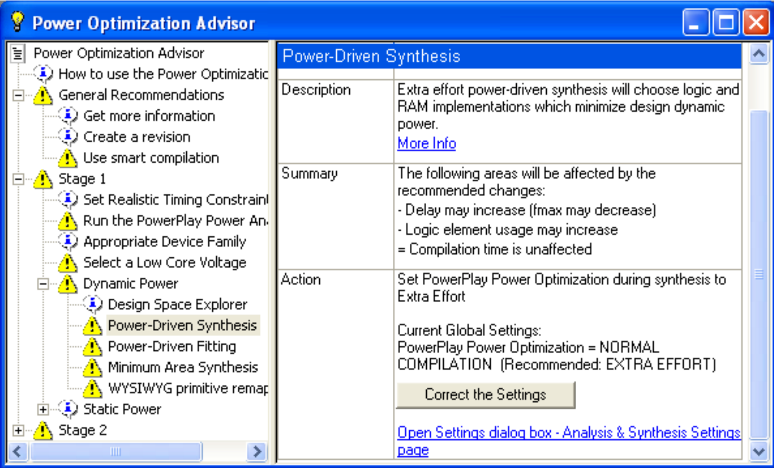
There is a link from each recommendation to the appropriate location in the Quartus II user interface where you can change the setting, such as the **Power-Driven Synthesis** setting. You can change the **Power-Driven Synthesis** setting by clicking **Open Settings dialog box - Analysis & Synthesis Settings page** (Figure 11–19). The **Settings** dialog box is shown with the **Analysis & Synthesis Settings** page selected, where you can change the **PowerPlay power optimization** settings.

Figure 11–19. Analysis & Synthesis Settings Page



After making the recommended changes, recompile your design. The Power Optimization Advisor indicates with green checkmarks that the recommendations were implemented successfully (Figure 11–20). You can use the PowerPlay Power Analyzer to verify your design power results.

Figure 11–20. Implementation of Power Optimization Advisor Recommendations



The recommendations listed in Stage 2 generally involve design changes, rather than CAD settings changes as in Stage 1. You can use these recommendations to further reduce your design power consumption. Altera recommends that you implement Stage 1 recommendations first, then the Stage 2 recommendations.

Conclusion

The combination of a smaller process technology, the use of low-k dielectric material, and reduced supply voltage significantly reduces dynamic power consumption in the latest FPGAs. To further reduce your dynamic power, use the design recommendations presented in this chapter to optimize resource utilization and minimize power consumption.

Referenced Documents

This chapter references the following documents:

- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *AN 437: Power Optimization in Stratix III FPGAs*
- *AN 514: Power Optimization in Stratix IV FPGAs*
- *Clock Control Block Megafunction User Guide (ALTCLKCTRL)*
- *Cyclone III Device Family Overview* chapter in the *Cyclone III Device Handbook*
- *Cyclone II Architecture* chapter in the *Cyclone II Device Handbook*
- *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*
- *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*
- *Stratix II Architecture* chapter in volume 1 in the *Stratix II Device Handbook*

- *Selectable I/O Standards in Stratix II Devices and Stratix II GX Devices* chapter in volume 2 of the *Stratix II Device Handbook*
- *Selectable I/O Standards in Cyclone II Devices* chapter in the *Cyclone II Device Handbook*
- *Selectable I/O Standards in Cyclone II Devices* chapter in the *Cyclone II Device Handbook*
- *Stratix IV Device Handbook*
- *Stratix III Device Handbook*
- *Stratix II Device Handbook*

Document Revision History

Table 11–5 shows the revision history for this chapter.

Table 11–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Was chapter 9 in the 8.1.0 release. ■ Updated for the Quartus II software release. ■ Added benchmark results. ■ Removed several sections. ■ Updated Figure 11–1, Figure 11–18, Figure 11–19, and Figure 11–20.	Updated for the Quartus II 9.0 software release.
November 2008 v8.1.0	■ Changed to 8½” × 11” page size. ■ Changed references to altsyncram to RAM. ■ Minor editorial updates	Updated for the Quartus II 8.1 software release.
May 2008 v8.0.0	■ Updated Table 9–1 and 9–9. ■ Updated “Architectural Optimization” on page 9–22 ■ Added “Dynamically-Controlled On-Chip Terminations” on page 9–26 ■ Updated “Referenced Documents” on page 9–29 ■ Updated references	Added support for Stratix IV devices.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

12. Analyzing and Optimizing the Design Floorplan

QII52006-9.0.0

Introduction

As FPGA designs grow larger and larger in density, analyzing the design for performance, routing congestion, and logic placement to meet the design requirements becomes critical.

You can use the Chip Planner tool in the Quartus® II software to perform design analysis and create a design floorplan. With some of the older device families, you must use the Timing Closure Floorplan in the Quartus II software to analyze the device floorplan.

This chapter discusses how to analyze the design floorplan with both the Chip Planner and the Timing Closure Floorplan. It also shows you how to create LogicLock™ regions in the design floorplan and how to use the LogicLock design methodology as a performance preservation tool for older device families.

You can use the Design Partition Planner along with the Chip Planner to floorplan your design. For more information, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* and the *Best Practices for Incremental Compilation Partition and Floorplan Assignments* chapters in volume 1 of the *Quartus II Handbook*.

This chapter includes the following topics:

- "Chip Planner Overview" on page 12–2
- "LogicLock Regions" on page 12–6
- "Using LogicLock Regions in the Chip Planner" on page 12–18
- "Design Floorplan Analysis Using the Chip Planner" on page 12–19
- "Design Analysis Using the Timing Closure Floorplan" on page 12–40
- "Using LogicLock Methodology for Older Device Families" on page 12–52
- "Scripting Support" on page 12–57

To view and create assignments for a design floorplan in supported devices, use the Chip Planner. To make I/O assignments, use the Pin Planner tool.

For more information about the Pin Planner tool, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Table 12–1 lists the device families supported by the Chip Planner and the Timing Closure Floorplan.

Table 12–1. Chip Planner Device Support (Part 1 of 2)

Device Family	Timing Closure Floorplan	Chip Planner
Arria® II GX	—	✓
Arria GX	—	✓
Stratix® IV	—	✓

Table 12–1. Chip Planner Device Support (Part 2 of 2)

Device Family	Timing Closure Floorplan	Chip Planner
Stratix III	—	✓
Stratix II	—	✓
Stratix II GX	—	✓
Stratix	—	✓
Stratix GX	—	✓
Cyclone® III	—	✓
Cyclone II	—	✓
Cyclone	—	✓
HardCopy® II	—	✓
MAX® IIZ	—	✓
MAX II	—	✓
MAX 7000	✓	—
ACEX®	✓	—
APEX™ II APEX 20KC APEX 20KE	✓	—
FLEX® 10K FLEX 10KA FLEX 10KE FLEX 6000	✓	—

This chapter describes how to use the Chip Planner and the Timing Closure Floorplan (for devices not supported by the Chip Planner) for design analysis.

Chip Planner Overview

The Chip Planner provides a visual display of chip resources. It can show logic placement, LogicLock regions, relative resource usage, detailed routing information, fan-in and fan-out connections between nodes, timing paths between registers, and delay estimates for paths. With the Chip Planner, you can view critical path information, physical timing estimates, and routing congestion.

You can also perform assignment changes with the Chip Planner, such as creating and deleting resource assignments, and post-compilation changes like creating, moving, and deleting logic cells and I/O atoms. By using the Chip Planner in conjunction with the Resource Property Editor, you can change connections between resources and make post-compilation changes to the properties of logic cells, I/O elements, PLLs, and RAM and digital signal processing (DSP) blocks. With the Chip Planner, you can view and create assignments for a design floorplan, perform power and design analyses, and implement ECOs in a single tool.

For details about how to implement ECOs in your design using the Chip Planner in the Quartus II software, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Starting the Chip Planner

To start the Chip Planner, on the Tools menu, click **Chip Planner (Floorplan & Chip Editor)**. The following methods can also be used to start the Chip Planner:

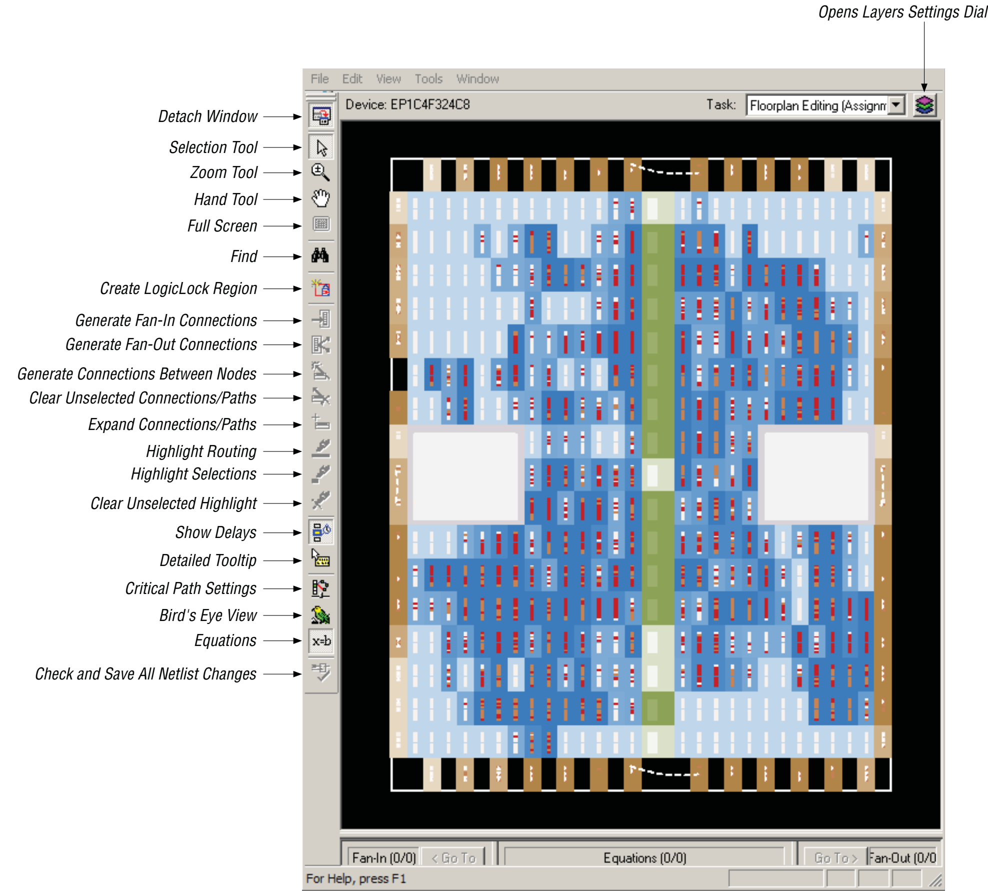
- Click the Chip Planner icon on the Quartus II software toolbar
- Use the right-click menu from the following sources and use the Locate menu:
  - Design Partition Planner
  - Compilation Report
  - LogicLock Regions window
  - Technology Map Viewer
  - Project Navigator window
  - RTL source code
  - Node Finder
  - Simulation Report
  - RTL Viewer
  - Report Timing panel of the TimeQuest Timing Analyzer

If the device in your project is not supported by the Chip Planner and you attempt to start the Chip Planner, the following message appears: `Can't display Chip Planner: the current device family is unsupported.` Use the Timing Closure Floorplan for these devices.

The Chip Planner Toolbar

The Chip Planner gives you powerful capabilities for design analysis with a user-friendly GUI. Many functions within the Chip Planner can be selected or performed from the menu items or by clicking the icons on the toolbar. Figure 12–1 shows an example of the Chip Planner toolbar and provides descriptions for commonly used icons available from the Chip Planner toolbar.

Figure 12–1. Chip Planner Toolbar



You can customize the icons present on the Chip Planner toolbar by clicking **Customize Chip Planner** on the Tools menu (if the Chip Planner window is attached), or by clicking **Customize** on the Tools menu (if the Chip Planner window is detached).

Chip Planner Tasks and Layers

The Chip Planner has predefined tasks that enable you to quickly implement ECO changes or manipulate assignments for the floorplan of the device. To select a task, click on the task name in the Task menu. The predefined tasks in the Chip Planner are:

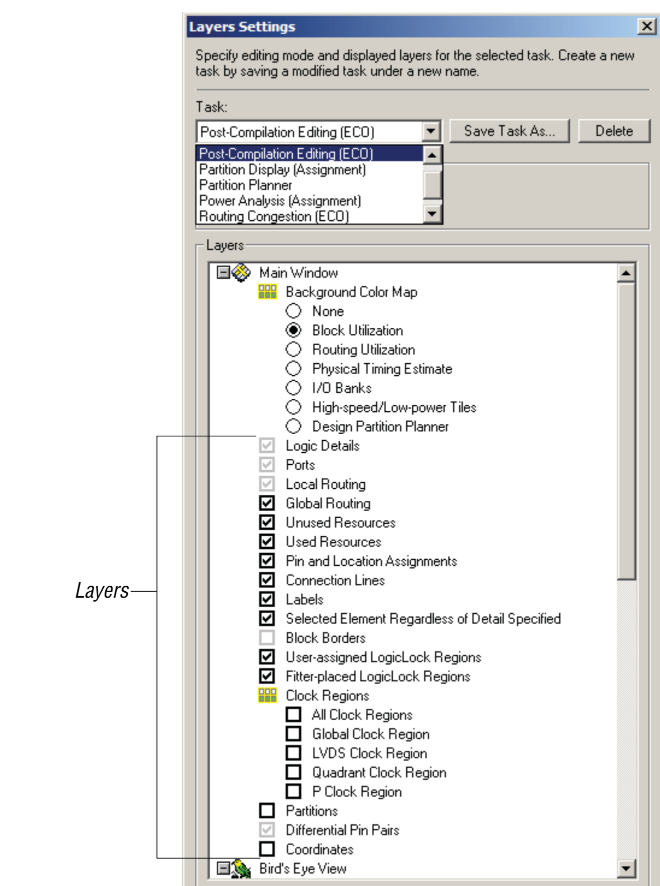
- **Floorplan Editing (Assignment)**
- **Post-Compilation Editing (ECO)**
- **Partition Display (Assignment)**
- **Partition Planner**
- **Routing Congestion (ECO)**
- **Clock Regions (Assignment)**—available for Stratix IV, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, and HardCopy II devices only

- **Power Analysis (Assignment)—**available for Stratix IV and Stratix III devices only

In the Chip Planner, layers allow you to specify the graphic elements that are displayed for a given task. You can turn off the display of specific graphic elements to increase the window refresh speed and reduce visual clutter when viewing complex designs. The **Background Color Map** can indicate the **Block Utilization**, **Routing Utilization**, **Physical Timing Estimate**, **I/O Banks**, or the High speed-Low power Tiles. When you select **Design Partition Planner** in the **Background Color Map** settings, the resources used by each partition are displayed in the Chip Planner with the same colors used for these partitions in the Design Partition Planner. You can access the Design Partition Planner on the Tools menu. For example, **Routing Utilization** indicates the relative routing utilization, and **Physical Timing Estimate** indicates the relative physical timing.

Each predefined task in the Chip Planner has a **Background Color Map**, a set of displayed layers, and an editing mode associated with it. Click the Layers icon (shown in Figure 12–1) to display the Layers Settings window (Figure 12–2). In this window you can select the layers and background color map for each task.

Figure 12–2. Layers in the Chip Planner



The Chip Planner operates in either **Assignment** or **ECO** mode. You can use either of these modes to perform design analyses. Use the **Floorplan Editor (Assignment)** task in the **Assignment** mode to manipulate LogicLock regions and location assignments in your design. The **Post Compilation Changes (ECO)** task in **ECO** mode allows you to implement ECO changes in your design. The **Partition Display (Assignment)** task allows you to view the placement of nodes and color-codes the nodes based on their partition. When you select the **Clock Regions (Assignment)** task, you can see the regions in your device that are driven by global clock networks. The **Power Analysis (Assignment)** task allows you to view high and low power resources in Stratix IV and Stratix III devices.

For more information about the **ECO** mode of operation, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

You can also create and save your own custom tasks. When you create a custom task, you can turn on or off any layer in your task. Layers can be turned on or off by checking the appropriate box located next to each layer. You can also select different **Background Color Maps** to be used for your custom task. After selecting the required settings, click **Save Task As** to save your custom task.

LogicLock Regions

LogicLock regions are user-defined regions within the device. Normally, LogicLock region are rectangular in shape. However, you can also create regions that are effectively non-rectangular in shape.

You can use LogicLock regions to create a floorplan for your design. Your floorplan can contain several LogicLock regions. A LogicLock region is defined by its size (height and width) and location (where the region is located on the device). You can specify the size or location of a region, or both, or the Quartus II software can generate these properties automatically. The Quartus II software bases the size and location of a region on the region's contents and the module's timing requirements. Table 12–2 describes the options for creating LogicLock regions.

Table 12–2. Types of LogicLock Regions

Properties	Values	Behavior
State	Floating (default), Locked	Floating regions allow the Quartus II software to determine the region's location on the device. Locked regions represent user-defined locations for a region and are shown with a solid boundary in the floorplan. A locked region must have a fixed size.
Size	Auto (default), Fixed	Auto-sized regions allow the Quartus II software to determine the appropriate size of a region given its contents. Fixed regions have a user-defined shape and size.
Reserved	Off (default), On, Limited	The reserved property allows you to define whether the Fitter can use the resources within a region for entities that are not assigned to the region. If the reserved property is turned on, only items assigned to the region can be placed within its boundaries. When you set it to limited, the Fitter does not place any logic from the parent region.
Origin	Any Floorplan Location	The origin is the origin of the LogicLock region's placement on the floorplan. For Arria GX, Stratix, and Cyclone series devices, and MAX II devices, the origin is located in the lower left corner. For other Altera® device families, the origin is located in the upper left corner.

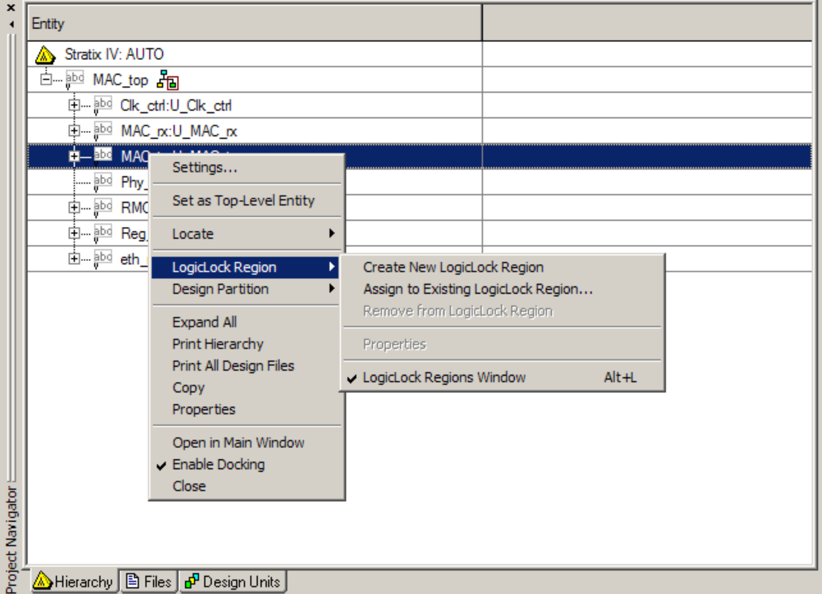
The Quartus II software cannot automatically define a region's size if the location is locked. Therefore, if you want to specify the exact location of the region, you must also specify the size.

You can use the Design Partition Planner in conjunction with LogicLock regions to create a floorplan for your design. For more information about using the Design Partition Planner, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Designs* and the *Best Practices for Incremental Compilation Partition and Floorplan Assignments* chapters in volume 1 of the *Quartus II Handbook*.

Creating LogicLock Regions

After you perform either a full compilation or analysis and elaboration on the design, the Quartus II software displays the hierarchy of the design. On the View menu, click **Project Navigator.** With the hierarchy of the design fully expanded, as shown in Figure 12–3, right-click on any design entity in the design and click **Create New LogicLock Region** to create a LogicLock region.

Figure 12–3. Using the Project Navigator to Create LogicLock Regions



Placing LogicLock Regions

A fixed region must contain all resources required for the module. Although the Quartus II software can automatically place and size LogicLock regions to meet resource and timing requirements, you can manually place and size regions to meet your design requirements. To do so, follow these guidelines:

- Place LogicLock regions with pin assignments on the periphery of the device, adjacent to the pins. For the Arria GX, Stratix, and Cyclone series of devices and MAX II devices, you must also include the I/O block within the LogicLock Region.
- Floating LogicLock regions cannot overlap.

- Avoid creating fixed and locked regions that overlap.

If you want to import multiple instances of a module into a top-level design, you must ensure that the device has two or more locations with exactly the same device resources. If the device does not have another area with exactly the same resources, the Quartus II software generates a fitting error during compilation of the top-level design.

When you import a LogicLock region, the Quartus II software changes the property to "floating" and assigns a new unique name. You can change the property to "fixed" to guarantee the same placement achieved previously. You can import or export LogicLock regions across devices within a family but not between families.

Placing Device Features into LogicLock Regions

A LogicLock region includes all device resources within its boundaries, including memory and pins. You can assign pins to LogicLock regions; however, this placement puts location constraints on the region. When the Quartus II software places a floating auto-sized region, it places the region in an area that meets the requirements of the LogicLock region's contents.

Pin assignments to LogicLock regions are effective only in fixed and locked regions. Pins assigned to floating regions do not influence the region's placement.

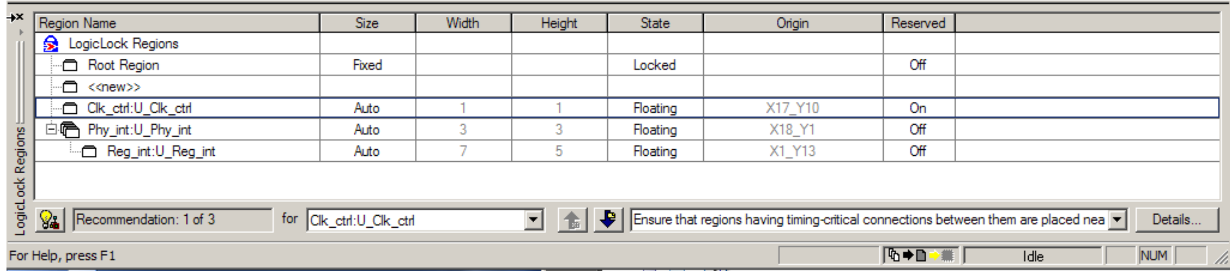
Only one LogicLock region can claim a device resource. If the boundary includes part of a device resource, the Quartus II software allocates the entire resource to the LogicLock region.

LogicLock Regions Window

The LogicLock window consists of the LogicLock Regions window (Figure 12–4) and the **LogicLock Region Properties** dialog box. Use the LogicLock Regions window to create LogicLock regions and assign nodes and entities to them. The dialog box provides a summary of all LogicLock regions in your design. In the LogicLock Regions window, you can modify a LogicLock region's properties size, state, width, height, origin, and whether the region is a reserved region. The LogicLock Regions window also has a recommendations toolbar at the bottom. Select a LogicLock region from the drop-down list in the recommendations toolbar to display the relevant suggestions to optimize that LogicLock region. When the region is back-annotated, the placement of each node within the region is relative to the region's origin, and the region's node placements are maintained during subsequent compilations.

The origin location varies, depending on the device family. For Arria GX, Stratix, and Cyclone series devices, and MAX II devices, the LogicLock region's origin is located at the lower-left corner of the region. For all other supported devices, the origin is located at the upper-left corner of the region.

Figure 12–4. LogicLock Regions Window



You can customize the LogicLock Regions window by dragging and dropping the various columns. Columns can also be hidden.

For designs that target Arria GX, Stratix, and Cyclone series and MAX II devices, the Quartus II software automatically creates a LogicLock region that encompasses the entire device. This default region is labelled `Root_region`, and it is effectively locked and fixed.

Use the **LogicLock Region Properties** dialog box to obtain detailed information about your LogicLock region, such as which entities and nodes are assigned to your region and what resources are required. The **LogicLock Region Properties** dialog box shows the properties of the current selected regions. You can also modify the settings for the regions you have created in this dialog box.

To open the **LogicLock Region Properties** dialog box, double-click any region in the LogicLock Regions window, or right-click the region and click **Properties**.

Hierarchical (Parent and Child) LogicLock Regions

You can define a hierarchy for a group of regions by declaring parent and child regions. The Quartus II software places a child region completely within the boundaries of its parent region, allowing you to further constrain module locations. Additionally, parent and child regions allow you to further improve a module's performance by constraining the nodes in the module's critical path.

To make one LogicLock region a child of another LogicLock region, in the LogicLock Regions window, select the new child region and drag and drop it in its new parent region.

The LogicLock region hierarchy does not have to be the same as the design hierarchy.

You can create both fixed and floating LogicLock regions within a fixed parent LogicLock region. A floating child region's location can float within its parent. If a child region is type fixed, its location remains locked relative to its parent's origin. A locked parent region's location is locked relative to the device. If the child's location is locked and the parent's location is changed, the child's origin changes but maintains the same placement relative to the origin of its parent. Either you or the Quartus II

software can determine a child region's size; however, the child region must fit entirely within the parent region. Although there is no limitation to the levels of hierarchy in LogicLock regions, remember that very complicated hierarchical regions might result in some LABs not being utilized; thus, effectively increasing the resource utilization in the device.

Reserved LogicLock Region

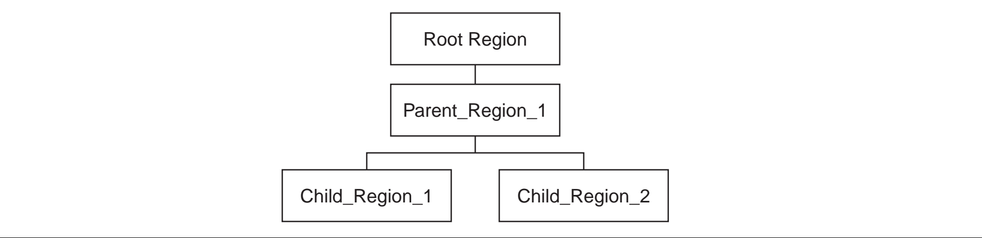
The Quartus II software honors all entity and node assignments to LogicLock regions. Occasionally, entities and nodes do not occupy an entire region, which leaves some of the region's resources unoccupied. To increase the region's resource utilization and performance, the Quartus II software's default behavior fills the unoccupied resources with other nodes and entities that have not been assigned to another region. You can prevent this behavior by turning on the **Reserved** option on the **General** tab of the **LogicLock Region Properties** dialog box. When this option is turned on, your LogicLock region only contains the entities and nodes that you specifically assigned to your LogicLock region. When you set the reserved property for a LogicLock region to **Limited**, the Fitter prevents logic from the immediate parent LogicLock region to be placed within the assigned area, but it might place other logic in the associated region.

In a team-based design environment, the **Limited** option is helpful in creating a device floorplan. When this option is turned on, each team can be assigned a portion of the device floorplan where placement and optimization of each submodule occurs. Device resources can be distributed to every module without affecting the performance of other modules.

Creating Non-Rectangular LogicLock Regions

You can use the reserved property of LogicLock regions to create regions that are effectively non-rectangular in shape. For example, consider a case in which there is one LogicLock region under the Root region and two child regions under this region (Figure 12–5).

Figure 12–5. Example 1

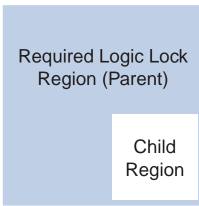


You can set the **Reserved** property of a LogicLock region to **On**, **Off**, or **Limited**. If you create a LogicLock region for Child_Region_1 with its **Reserved** property set to **Limited**, the Fitter does not place nodes that are members of Parent_Region_1 or Child_Region_2 into the boundary of Child_Region_1. However, the Fitter can place nodes that are not members of Parent_Region_1 or Child_Region_1 (such as members of the Root_Region) into Child_Region_1. On the other hand, if Child_Region_1 is set to exclude all non-members, the Fitter can only place nodes that are members of Child_Region_1 into the region.

If the Parent Region's reserved property is turned off, then the Fitter might place other logic in the allocated region.

If you want to create a non-rectangular region as shown in Figure 12–6, you can create two rectangular hierarchical LogicLock regions. Turn off the reserved property on the parent LogicLock and set the reserved property on the child LogicLock region to **Limited** to prevent the Fitter from placing any logic of the module assigned to the parent LogicLock region. Because of this, logic that is external to the parent LogicLock region might be placed in the area allocated to the child region. This gives a non-rectangular LogicLock region.

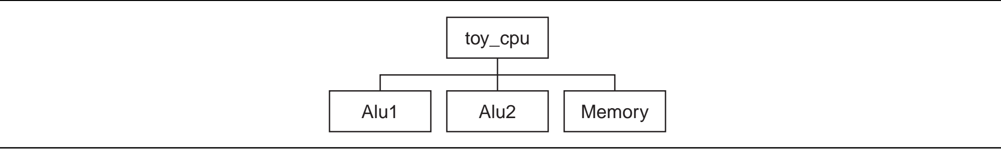
Figure 12–6. Non-Rectangular Region



Examples of Non-Rectangular LogicLock Regions Using Reserved Property

The following examples use the design hierarchy shown in Figure 12–7.

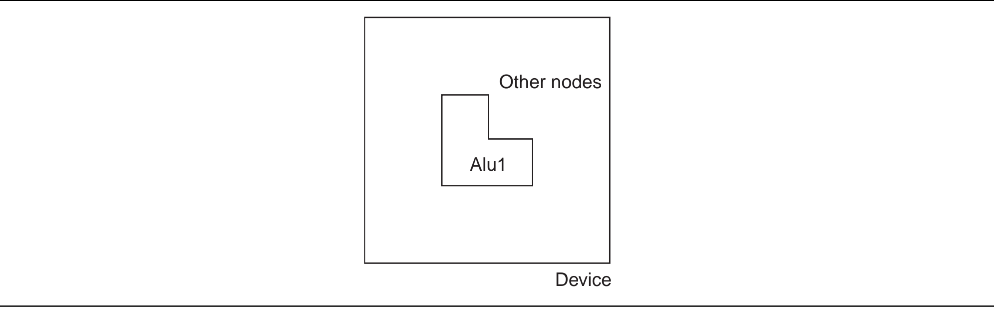
Figure 12–7. An Example Design Hierarchy



Example 1: Creating an L-Shaped Region

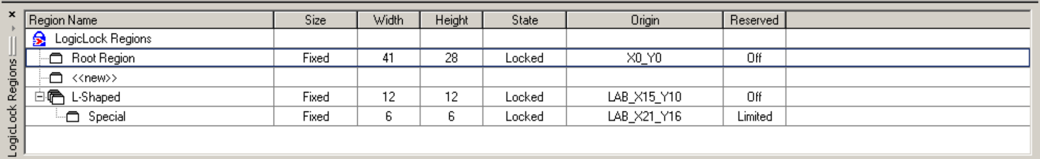
In the design hierarchy example in Figure 12–7, suppose you want to create an L-shaped region, such that the Alu1 module is placed completely inside the region, and the non-Alu1 nodes can be placed anywhere on the chip (as shown in Figure 12–8).

Figure 12–8. Creating an L-Shaped Region



The L-shaped region defines a rectangular region that is carved out by a child LogicLock region (**Special**) to achieve the L-shape effect. The **Reserved** property of this child LogicLock region is set to **Limited**, such that the Fitter does not require logic from members of Alu1 (which is the parent region of the region named **Special**) inside it while letting other nodes in. Not displayed in Figure 12–8, the Alu1 entity instance is assigned as a member to the L_Shaped region. This can be achieved by creating a hierarchical LogicLock region as shown in Figure 12–9.

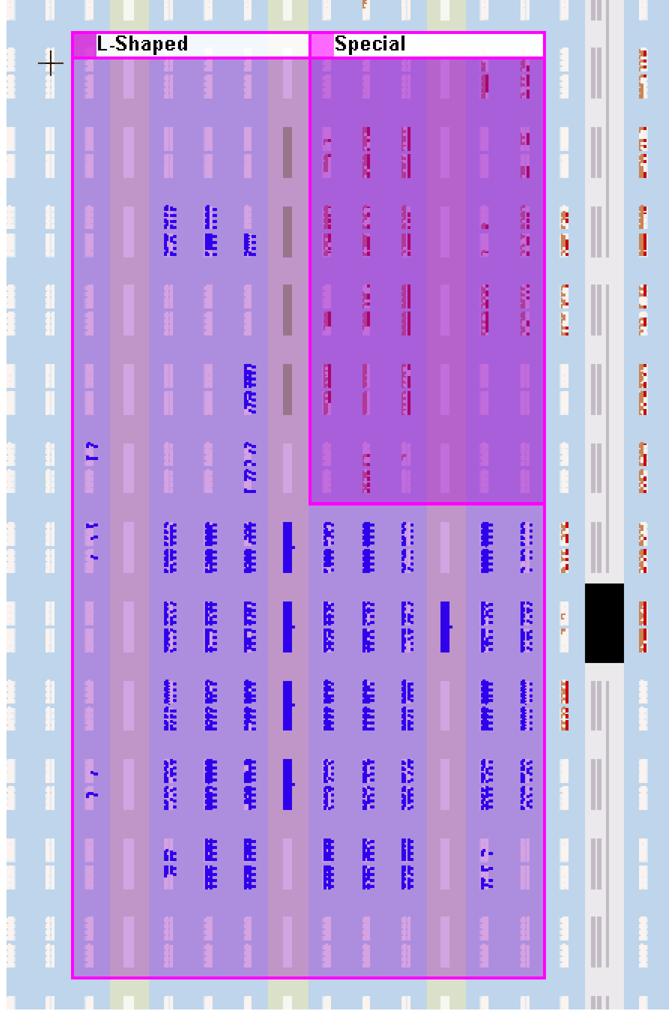
Figure 12–9. Hierarchical LogicLock Region



Region Name	Size	Width	Height	State	Origin	Reserved
LogicLock Regions						
Root Region	Fixed	41	28	Locked	X0_Y0	Off
<<new>>						
L-Shaped	Fixed	12	12	Locked	LAB_X15_Y10	Off
Special	Fixed	6	6	Locked	LAB_X21_Y16	Limited

Figure 12–10 illustrates the expected fitting results with these LogicLock regions. Nodes from the Alu1 entity instance are colored blue, while nodes from the rest of the design are colored red.

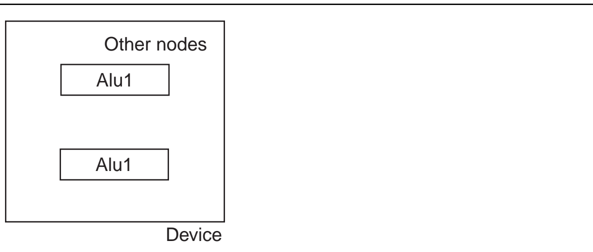
Figure 12–10. Expected Fitting Results with LogicLock Regions



Example 2: Region with Disjoint Areas

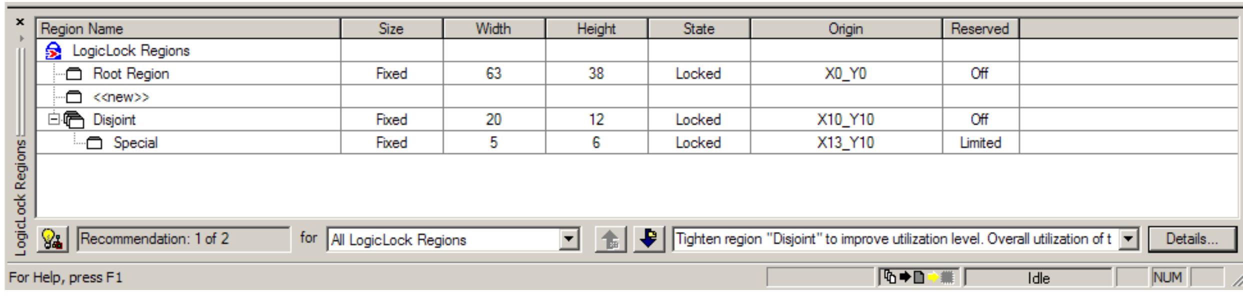
Suppose you want to create a region consisting of two disjoint rectangles (or any number of disjoint areas), such that the Alu1 module is placed completely inside the region, and the non-Alu1 nodes can be placed anywhere on the chip as shown in Figure 12–11.

Figure 12–11. Region Consisting of Two Disjointed Rectangles



You can achieve a region with disjoint areas using the following region hierarchy example in Figure 12–12.

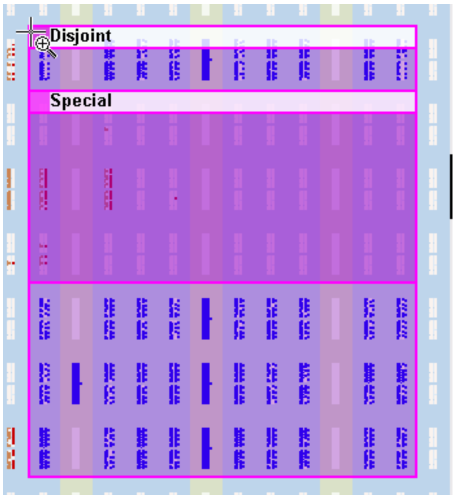
Figure 12–12. Region with Disjoint Areas



The disjoint region defines a rectangular region that is carved out by the Special child region to achieve the disjoint effect. Notice that the Special region is set to reserved "from members of parent region hierarchy" to prevent the Alu1 nodes from being placed inside it, while letting other nodes in. The Alu1 entity instance should be assigned to the Disjoint LogicLock region.

Figure 12–13 shows the expected fitting results with the LogicLock regions. Nodes from the Alu1 entity instance are colored blue, while nodes from the rest of the design are colored red and brown.

Figure 12–13. Expected Fitting Results with the LogicLock Regions



Excluded Resources

The Excluded Resources feature allows you to easily exclude specific device resources such as DSP blocks or M4K memory blocks from a LogicLock region. For example, you can specify resources that belong to a specific entity that are assigned to a LogicLock region, and specify that these resources be included with the exception of the DSP blocks. Use the Excluded Resources feature on a per-LogicLock region member basis.

To exclude certain device resources from an entity, in the **LogicLock Region Properties** dialog box, highlight the entity in the **Design Element** column, and click **Edit**. In the **Edit Node** dialog box, under **Excluded Element Types**, click the Browse button. In the **Excluded Resources Element Types** dialog box, you can select the device resources you want to exclude from the entity. When you have selected the resources to exclude, the **Excluded Resources** column is updated in the **LogicLock Region Properties** dialog box to reflect the excluded resources.

The Excluded Resources feature prevents certain resource types from being included in a region, but it does not prevent the resources from being placed inside the region unless the region's **Reserved** property is set to **On**. To inform the Fitter that certain resources are not required inside a LogicLock region, define a resource filter.

Additional Quartus II LogicLock Design Features

To complement the **LogicLock Regions** dialog box and **Device Floorplan** view, the Quartus II software has additional features to help you design with the LogicLock regions.

Tooltips

When you move the mouse pointer over a LogicLock region name on the **LogicLock Regions** dialog box, or over the top bar of the LogicLock region in the Chip Planner or Timing Closure Floorplan, the Quartus II software displays a tooltip with information about the properties of the LogicLock region.

Placing the mouse pointer over **Fitter-placed LogicLock regions** displays the maximum routing delay within the LogicLock region. To turn this feature on, on the View menu, point to **Routing** and click **Show Intra-region Delay**.

Show Critical Paths

You can display the critical paths in the design by turning on the **Show Critical Paths** option. Use this option with the **Critical Paths Settings** option to display paths based on the Timing Analysis report.

Analysis and Synthesis Resource Utilization by Entity

The Compilation Report contains an **Analysis and Synthesis Resource Utilization by Entity** section, which reports accurate resource usage statistics, including entity-level information. This feature is useful when manually creating LogicLock regions.

Path-Based Assignments

You can assign paths to LogicLock regions based on source and destination nodes, allowing you to easily group critical design nodes into a LogicLock region. Any of the following types of nodes can be the source and destination nodes:

- Valid register-to-register path, meaning that the source and destination nodes must be registers
- Valid pin-to-register path, meaning the source node is a pin and the destination node is a register
- Valid register-to-pin path, meaning that the source node is a register and the destination node is a pin

- Valid pin-to-pin path, meaning that both the source and destination nodes are pins

To open the **Paths** dialog box, on the **General** tab of the **Logic Lock Regions** dialog box, click **Add Path**.

Both "*" and "?" wildcard characters are allowed for the source and destination nodes. When creating path-based assignments, you can exclude specific nodes using the **Name exclude** field in the **Paths** dialog box. The Quartus II software ignores all paths passing through the nodes that match the setting in the **Name exclude** field. For example, consider a case with two paths between the source and destination—one passing through node A and the other passing through node B. If you specify node B in the **Name exclude** field, only the path assignment through node A is valid.

You can also use the Quartus II Timing Analysis Report to create path-based assignments by following these steps:

1. Expand the **Timing Analyzer** section in the Compilation Report.
2. Select any of the clocks in the section labeled "Clock Setup:<*clock name*>."
3. Locate a path that you want to assign to a LogicLock region. Drag this path from the Report window and drop it in the appropriate row in the LogicLock Region pane in the Quartus II GUI.

This operation creates a path-based assignment from the source register to the destination register as shown in the Timing Analysis Report.

Quartus II Revisions Feature

When you evaluate different LogicLock regions in your design, you might require experiments with different configurations to achieve your desired results. The Quartus II software Revisions feature provides a convenient way to organize the same project with different settings until an optimum configuration is found.

On the Project menu, click **Revisions**. In the **Revisions** dialog box, you can create and specify revisions. Revision can be based on the current design or any previously created revisions. Each revision can have an associated description. Revisions are a convenient way to organize the placement constraints created for your LogicLock regions.

LogicLock Assignment Precedence

Conflicts can arise during the assignment of entities and nodes to LogicLock regions. For example, an entire top-level entity might be assigned to one region and a node within this top-level entity assigned to another region. To resolve conflicting assignments, the Quartus II software maintains an order of precedence for LogicLock assignments. The following order of precedence, from highest to lowest, applies:

1. Exact node-level assignments
2. Path-based and wildcard assignments
3. Hierarchical assignments

However, conflicts can also occur within path-based and wildcard assignments. Path-based and wildcard assignment conflicts arise when one path-based or wildcard assignment contradicts another path-based or wildcard assignment. For example, a path-based assignment is made containing a node labeled `X` and assigned to LogicLock region `PATH_REGION`. A second assignment is made using wildcard assignment `X*` with node `X` being placed into region `WILDCARD_REGION`. As a result of these two assignments, node `X` is assigned to two regions: `PATH_REGION` and `WILDCARD_REGION`.

To resolve this type of conflict, the Quartus II software maintains the order in which the assignments were made and treats the most recent assignment created with the highest priority.

Open the **Priority** dialog box by selecting **Priority** on the **General** tab of the **LogicLock properties** dialog box. You can change the priority of path-based and wildcard assignments by using the **Up** and **Down** buttons in the **Priority** dialog box. To prioritize assignments between regions, you must select multiple LogicLock regions. Once the regions have been selected, you can open the **Priority** dialog box from the LogicLock Properties window.

Normally all nodes assigned to a particular LogicLock region reside within the boundaries of that region.

Virtual Pins

When you compile a design in the Quartus II software, all I/O ports are directly mapped to pins on the targeted device. The I/O port mapping can create problems for a modular and hierarchical design as lower-level modules can have I/O ports that exceed device pins available on the targeted device. Also, the I/O ports might not directly feed a device pin, but instead drive other internal nodes. The Quartus II software accommodates this situation by supporting virtual pins.

The Virtual Pin assignment communicates to the Quartus II software which I/O ports of the design module are internal nodes in the top-level design. These assignments prevent the number of I/O ports in the lower-level modules from exceeding the total number of available device pins. Every I/O port that is designated a virtual pin is mapped to either an LCELL or an adaptive logic module (ALM) depending on the target device.

Bidirectional, registered I/O pins, and I/O pins with output enable signals cannot be virtual pins.

In the top-level design, these virtual pins are connected to an internal node of another module. Making assignments to virtual pins allows you to place them in the same location or region on the device as that of the corresponding internal nodes in the top-level module. This feature has the added benefit of providing accurate timing information during lower-level module optimization.

Use the following guidelines for creating virtual pins in the Quartus II software:

- Do not declare clock pins
- Nodes or signals that drive physical device pins in the top-level design should not be declared as virtual pins

In the Node Finder, setting **Filter Type** to **Pins: Virtual** allows you to display all assigned virtual pins in the design. From the Assignment Editor, to access the Node Finder, double-click the **To** field; when the arrow appears on the right side of the field, click the arrow and select **Node Finder**.

Using LogicLock Regions in the Chip Planner

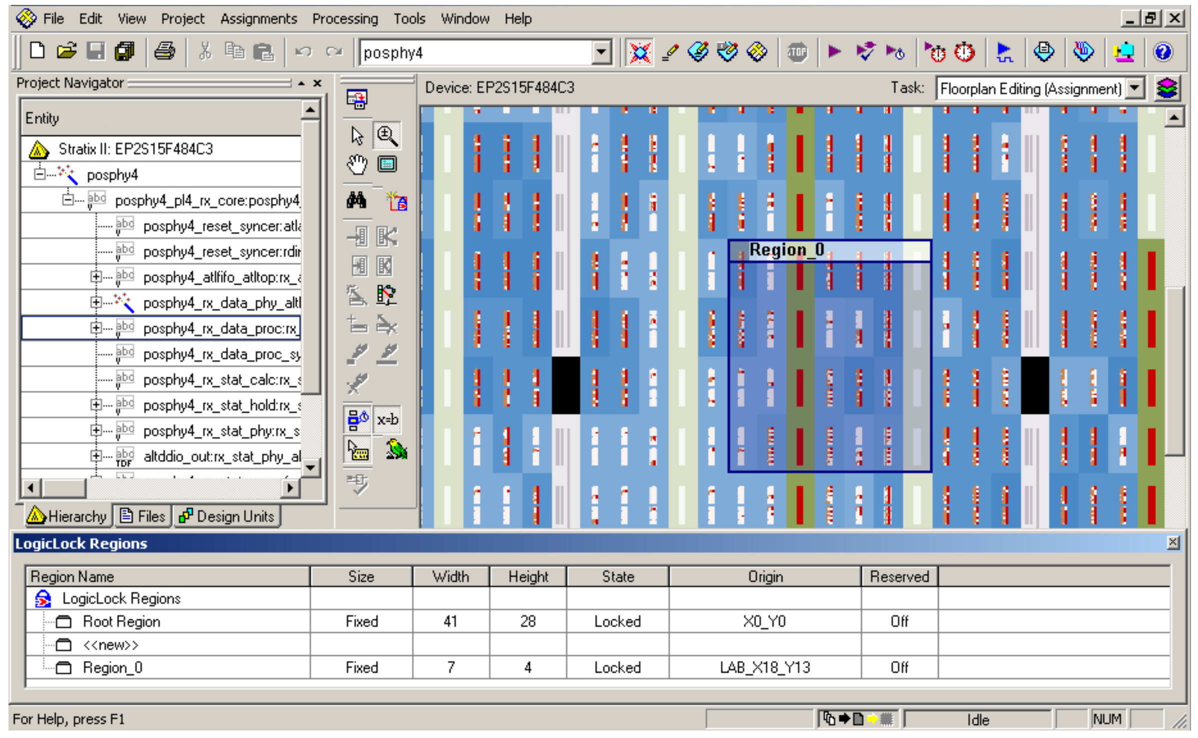
You can easily edit properties of existing LogicLock regions or assign resources to them in the Chip Planner. You can also create new LogicLock regions using the Chip Planner.

Assigning LogicLock Region Content

When you have defined a LogicLock region, you must assign resources to it using the Chip Planner, the **LogicLock Regions** dialog box, or a Tcl script.

You can drag selected logic displayed in the **Hierarchy** tab of the Project Navigator, in the Node Finder, or in a schematic design file, and drop it into the Chip Planner or the **LogicLock Regions** dialog box. Figure 12–14 shows logic that has been dragged from the **Hierarchy** tab of the Project Navigator and dropped into a LogicLock region in the Chip Planner.

Figure 12–14. Drag and Drop Logic in the Chip Planner



You can also drag logic from the **Hierarchy** tab of the Project Navigator and drop it in the LogicLock Regions **Properties** dialog box. Logic can also be dropped into the **Design Element Assigned** column of the **Contents** tab of the **LogicLock Region Properties** box.

You must assign pins to a LogicLock region manually. The Quartus II software does not include pins automatically when you assign an entity to a region. The software only obeys pin assignments to locked regions that border the periphery of the device. For the Stratix and Cyclone series of devices, and MAX II devices, the locked regions must include the I/O pins as resources.

Creating LogicLock Regions with the Chip Planner

In the View menu of the Chip Planner, click **Create LogicLock Region**. In the Chip Planner, click and drag to create a region of your preferred location and size.

Viewing Connections Between LogicLock Regions in the Chip Planner

You can view and edit LogicLock regions using the Chip Planner. Select the **Floorplan Editing (Assignment)** task or select any task with **User-assigned LogicLock regions** setting enabled to manipulate LogicLock regions.

The Chip Planner shows the connections between LogicLock regions. Rather than showing multiple connection lines from one LogicLock region to another, you can select the option of viewing connections between LogicLock regions as a single bundled connection. To use this option, open the Chip Planner floorplan and on the View menu, click **Generate Inter-region Bundles**.

In the **Generate Inter-region Bundles** dialog box, specify the **Source node to region fanout less than** and the **Bundle width greater than** values.

For more information about the parameters in the **Generate Inter-region Bundles** dialog box, refer to the Quartus II Help.

Design Floorplan Analysis Using the Chip Planner

The Chip Planner assists you in visually analyzing the floorplan of your design at any stage of your design cycle. With the Chip Planner, you can view post-compilation placement, connections, and routing paths. You can also create LogicLock regions and location assignments. The Chip Planner allows you to create new logic cells and I/O atoms and to move existing logic cells and I/O atoms using the architectural floorplan of your design. You can also see global and regional clock regions within the device, and the connections between both I/O atoms and PLLs and the different clock regions.

From the Chip Planner, you can launch the Resource Property Editor. The Resource Property Editor lets you make changes to the properties and parameters of device resources, and modify connectivity between certain types of device resources. Changes that you perform are logged within the Change Manager. The Change Manager helps you track the various changes you make on your design floorplan, so that you can selectively undo the changes if necessary.

For more information about the Resource Property Editor and the Change Manager, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

The following sections present design analysis procedures and views of the Chip Planner, which you can use with any predefined task of the Chip Planner (unless explicitly stated that a certain procedure requires a particular task or editing mode).

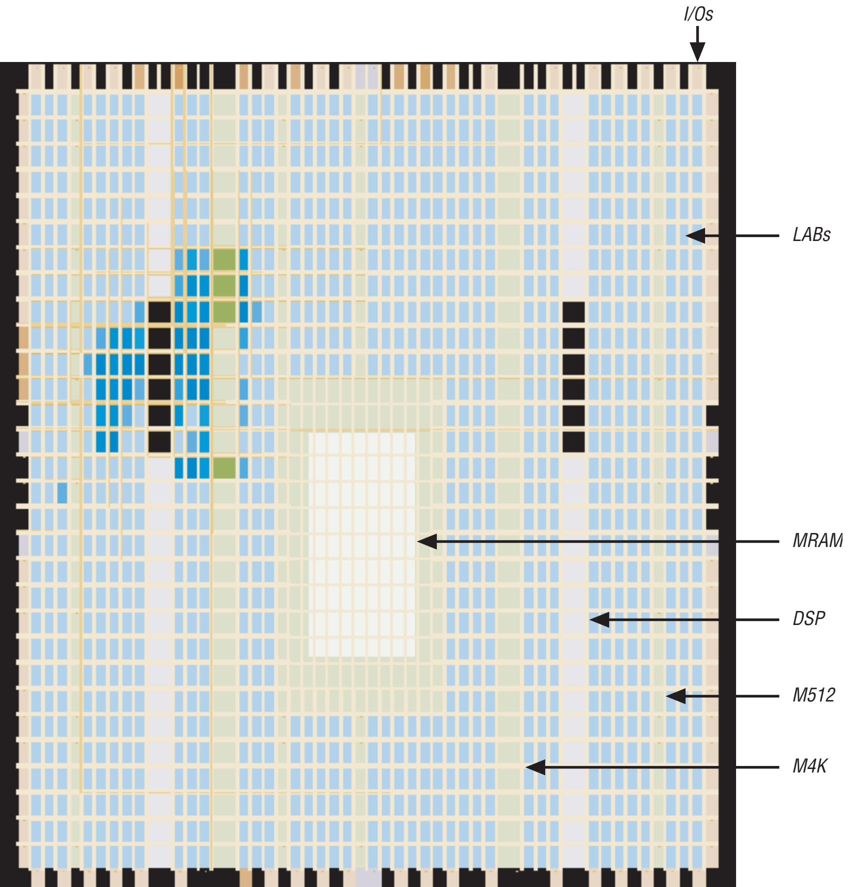
Chip Planner Floorplan Views

The Chip Planner uses a hierarchical zoom viewer that shows various abstraction levels of the targeted Altera device. As you increase the zoom level, the level of abstraction decreases, thus revealing more detail about your design.

First-Level View

The first level provides a high-level view (LAB level view) of the entire device floorplan. You can locate a node and view the placement of that node in your design. Figure 12–15 shows the Chip Planner's Floorplan first-level view of a Stratix device.

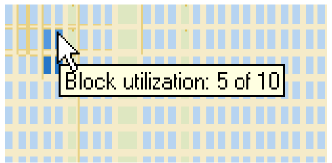
Figure 12–15. Chip Planner's First-Level Floorplan View



Each resource is shown in a different color. The Chip Planner floorplan uses a gradient color scheme in which the color becomes darker as the utilization of a resource increases. For example, as more LEs are used in the logic array block (LAB), the color of the LAB becomes darker.

When you place the mouse pointer over a resource at this level, a tooltip appears that describes, at a high level, the utilization of the resource (Figure 12–16).

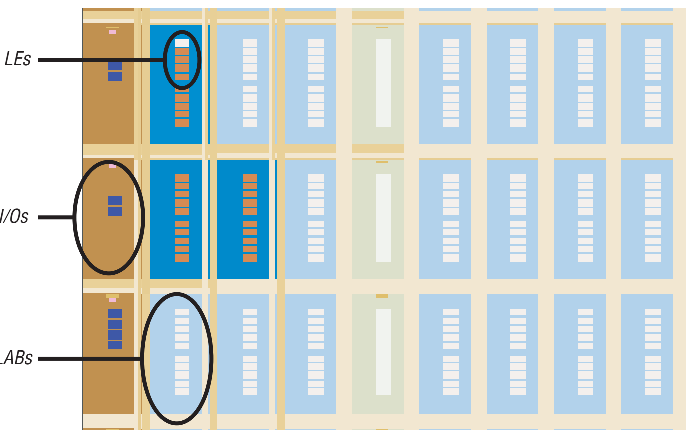
Figure 12–16. Tooltip Message: First-Level View



Second-Level View

As you zoom in, the level of detail increases. Figure 12–17 shows the second-level view of the Chip Planner Floorplan for a Stratix device.

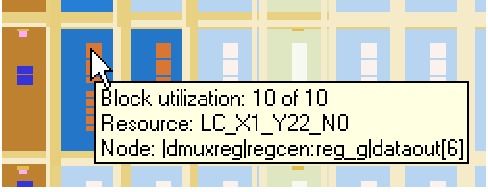
Figure 12–17. Chip Planner's Second-Level Floorplan View



At this level, the contents of LABs and I/O banks and the routing channels that are used to connect resources are all visible.

When you place the mouse pointer over an LE or ALM at this level, a tooltip is displayed (Figure 12–18) that shows the name of the LE/ALM, the location of the LE/ALM, and the number of resources that are used with that LAB. When you place the mouse pointer over an interconnect, the tooltip shows the routing channels that are used by that interconnect. At this level, you can move LEs, ALMs, and I/Os from one physical location to another.

Figure 12–18. Tooltip Message: Second-Level View



Third-Level View

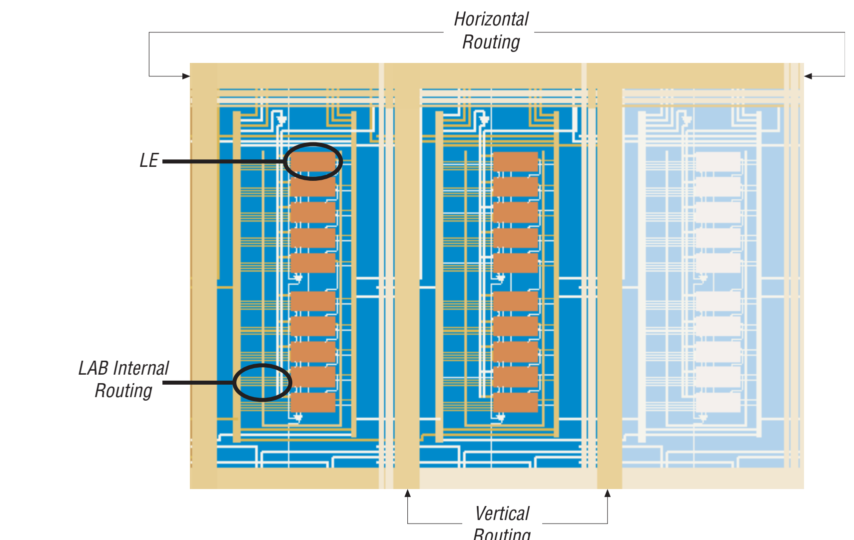
At the third level, which provides a more detailed view, you can see each routing resource that is used within a LAB in the FPGA. Figure 12–19 shows the level of detail at the third-level view for a Stratix device.

From the third level, you can move LEs, ALMs, and I/Os from one physical location to another. You can move a resource by selecting, dragging, and dropping it into the desired location. At this level, you can also create new LEs and I/Os when you are in the post-compilation (ECO) mode.

You can delete a resource only after all of its fan-out connections are removed. Moving nodes in the **Floorplan Editing (Assignment)** task creates an assignment. However, if you move logic nodes in the **Post-Compilation Editing (ECO)** task, that change is considered an ECO change. For more information about Floorplan Assignments, refer to "Viewing Assignments in the Chip Planner" on page 12–37.

For more information about performing ECOs, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook.*

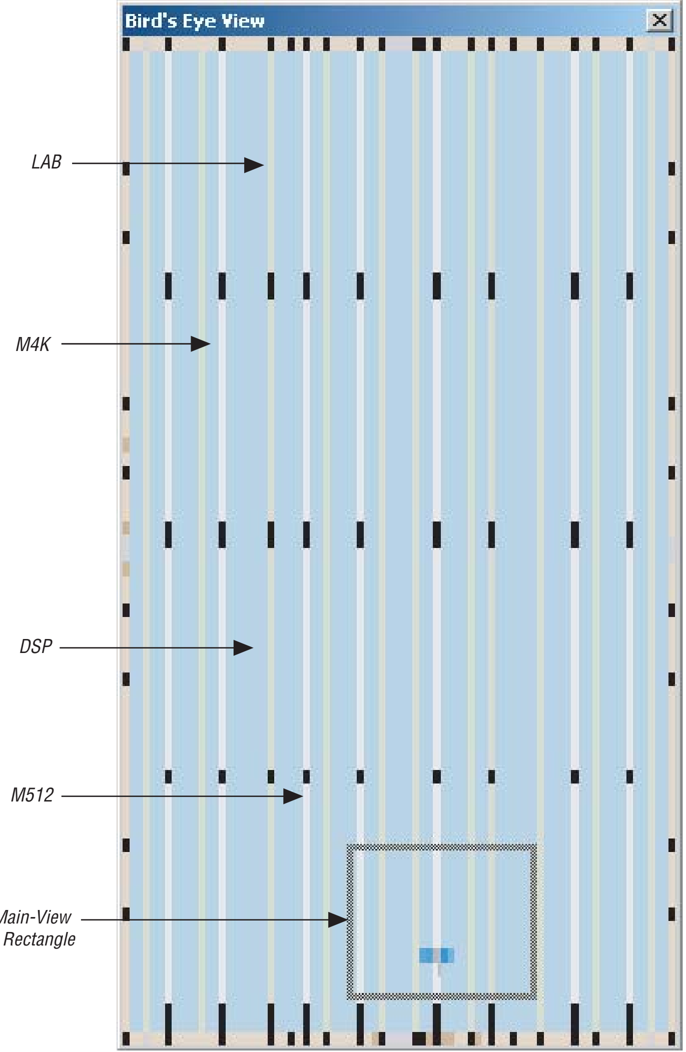
Figure 12–19. Chip Planner's Third-Level Floorplan View



Bird's Eye View

The Bird's Eye View (Figure 12–20) displays a high-level picture of resource usage for the entire chip and provides a fast and efficient way to navigate between areas of interest in the Chip Planner.

Figure 12–20. Bird's Eye View



The Bird's Eye View is displayed as a separate window that is linked to the Chip Planner floorplan. When you select an area of interest in the Bird's Eye View, the Chip Planner floorplan automatically refreshes to show that region of the device. As you change the size of the main-view rectangle in the Bird's Eye View window, the main Chip Planner floorplan window also zooms in (or zooms out). You can make the main-view rectangle smaller in the Bird's Eye View to see more detail on the Chip Planner floorplan window by right-clicking and dragging inside the Bird's Eye View.

The Bird's Eye View is particularly useful when parts of your design that you are interested in are at opposite ends of the chip, and you want to quickly navigate between resource elements without losing your frame of reference.

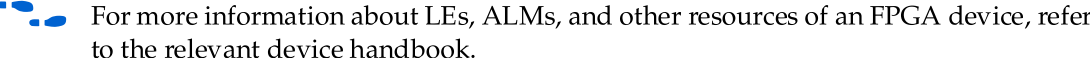
Viewing Architecture-Specific Design Information

With the Chip Planner, you can view the following architecture-specific information related to your design:

- **Device routing resources used by your design**—View how blocks are connected, as well as the signal routing that connects the blocks.
- **LE configuration**—View how a logic element (LE) is configured within your design. For example, you can view which LE inputs are used; if the LE utilizes the register, the look-up table (LUT), or both; as well as the signal flow through the LE.
- **ALM configuration**—View how an ALM is configured within your design. For example, you can view which ALM inputs are used, if the ALM utilizes the registers, the upper LUT, the lower LUT, or all of them. You can also view the signal flow through the ALM.
- **I/O configuration**—View how the device I/O resources are used. For example, you can view which components of the I/O resources are used, if the delay chain settings are enabled, which I/O standards are set, and the signal flow through the I/O.
- **PLL configuration**—View how a phase-locked loop (PLL) is configured within your design. For example, you can view which control signals of the PLL are used with the settings for your PLL.
- **Timing**—View the delay between the inputs and outputs of FPGA elements. For example, you can analyze the timing of the `DATAB` input to the `COMBOUT` output.

In addition, you can modify the following properties of an Altera device with the Chip Planner:

- LEs and ALMs
- I/O cells
- PLLs
- Registers in RAM and DSP blocks
- Connections between elements
- Placement of elements



For more information about LEs, ALMs, and other resources of an FPGA device, refer to the relevant device handbook.

Viewing Available Clock Networks in the Device

When you select the **Clock Regions (Assignment)** from the **Task** list, you can view the areas of the chip that are driven by the global and regional clock networks available in the device. This global clock display feature is available only for Stratix IV, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone III, and HardCopy II device families.

When you select the **Clock Regions** task, the Chip Planner displays various types of regional and global clocks and the regions they cover in the device. The connectivity between clock regions, pins, and PLLs is also shown. Clock regions are shown with rectangular overlay boxes with name labels of clock type and index.You can select each clock network region by clicking on it. The clock-shaped icon at the top-left corner indicates that the region represents a clock network region.

Clock types are listed in the Layer Settings window. You can change the color of the clock network in the Chip Planner on the **Options** page of the Tools menu.

You can customize your view of the global clock networks by using the layers setting in the Chip Planner GUI. You can turn on or off the display of all clock regions with the **All types** option. When the selected device does not contain a specific clock region, the option for that category is turned off in the dialog box. Figure 12–21 shows the potential fan-in in the Chip Planner.

Figure 12–21. Potential Fan-In

To trace the possible connectivity to each clock network region, select the clock network region and use the **Generate Potential Fan-In** and **Generate Potential Fan-Out** commands.

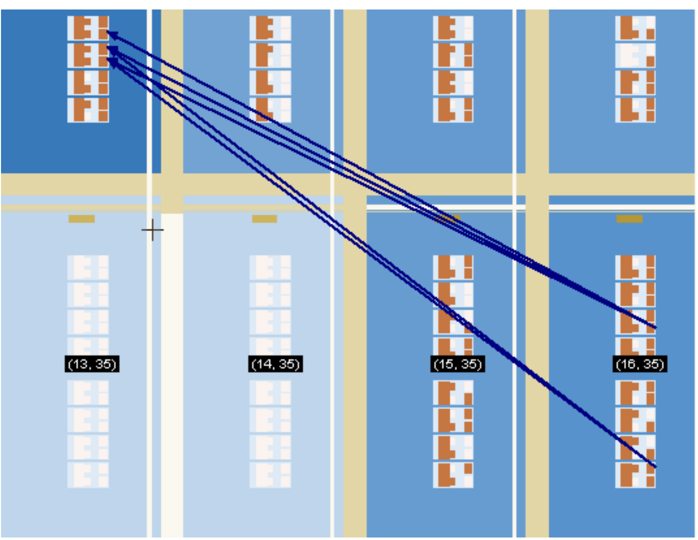
If you are interested in locating the clock regions that a pin or PLL can feed, select the pin or the PLL, then use the **Generate Fan-Out Connections** command. Connection arrows are drawn from the selected pins or PLLs to their clock regions.

When you use the **Generate Fan-In Connections** and **Generate Fan-Out Connections** commands, the Chip Planner shows connections that are actually used in the netlist for the selected clock region.

Viewing Critical Paths

Critical paths are timing paths in your design that have a negative slack. These timing paths can span from device I/Os to internal registers, registers-to-registers, or registers-to-devices I/Os. The View Critical Paths feature displays routing paths in the Chip Planner, as shown in Figure 12–22. The criticality of a path is determined by its slack and is shown in the timing analysis report. Design analysis for timing closure is a fundamental requirement for optimal performance in highly complex designs. The Quartus II Chip Planner helps you close timing on complex designs with its analytical capability.

Figure 12–22. Chip Planner Showing Critical Path

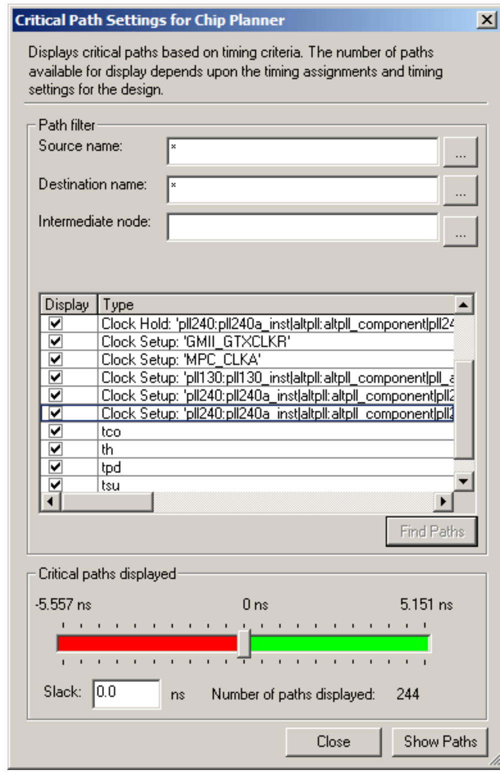


Viewing critical paths in the Chip Planner helps you analyze why a specific path is failing. You can see if any modification in the placement can potentially reduce the negative slack. You can display details of a path (to expand/collapse the path to/from the connections in the path) by clicking on Expand Connections/Paths button in the toolbar, or by clicking on the "+/-" on the label.

To view critical paths in the Chip Planner, on the View menu, click **Critical Path Settings**. In the **Critical Path Settings** dialog box, click **Show Path** (refer to Figure 12–23 on page 12–27).

If you are using the TimeQuest Timing Analyzer, you can locate the failing paths starting from the timing report. To locate the critical paths, run the Report Timing task from the Custom Reports group in the Tasks pane of the TimeQuest GUI. From the View pane, which lists the failing paths, you can right-click on any failing path or node, and select **Locate Path**. From the pop-up dialog box, select **Chip Planner** to see the failing path in the Chip Planner.

Figure 12–23. Critical Path Settings for the Chip Planner



When viewing critical paths, you can specify the clock in the design you want to view. You determine the paths to be displayed by specifying the slack threshold in the slack field of the **Critical Path Settings** dialog box. This dialog box also helps you to filter specific paths based on the source and destination registers.

Timing settings must be made and a timing analysis performed for paths to be displayed in the floorplan.

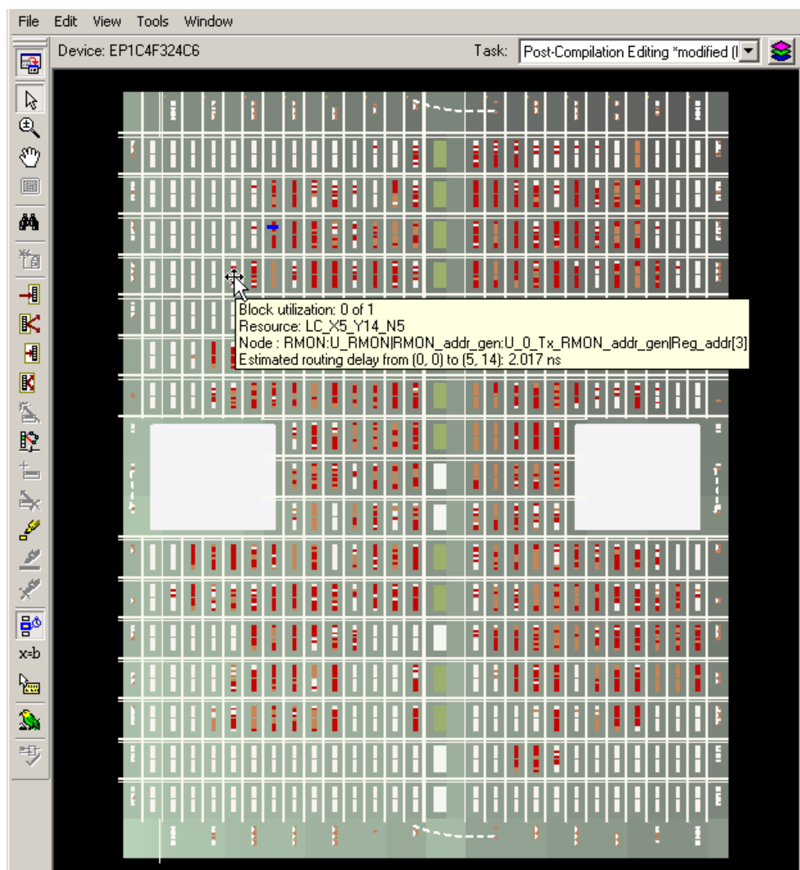
For more information about performing static timing analysis with the Quartus II Classic Timing Analyzer, refer to the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*. For more information about performing static timing analysis with the Quartus II TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Viewing Physical Timing Estimates

In the Chip Planner, you can select a resource and see the approximate delay to any other resource on the device. After you select a resource, the delay is represented by the color of potential destination resources. The lighter the color of the resource, the longer the delay.

To see the physical timing map of the device, in the Chip Planner, click the **Layers** icon located next to the Task menu. **Under Background Color Map**, select **Physical Timing Estimate**. Select a source and move your cursor to a destination resource. The Chip Planner displays the approximate routing delay between your selected source and destination register (Figure 12–24).

Figure 12–24. Chip Planner Displaying Routing Delay



You can use the physical timing estimate information when attempting to improve the Fitter results by manually moving logic in a device or when creating LogicLock regions to group logic together. This feature allows you to estimate the physical routing delay between different nodes so that you can place critical nodes and modules closer together, and move non-critical or unrelated nodes and modules further apart.

In addition to reducing delay between critical nodes, you can make placement assignments to reduce the routing congestion between critical and noncritical entities and modules. This allows the Quartus II Fitter to meet the design timing requirements.

Moving logic and creating manual placements is an advanced technique to meet timing requirements and must be done after careful analysis of the design. Moving nodes in the **Floorplan Editing (Assignment)** task creates an assignment. However, if you move logic nodes in the **Post-Compilation Editing (ECO)** task, that change is considered an ECO change.

For more information about Floorplan Assignments, refer to "Viewing Assignments in the Chip Planner" on page 12–37.

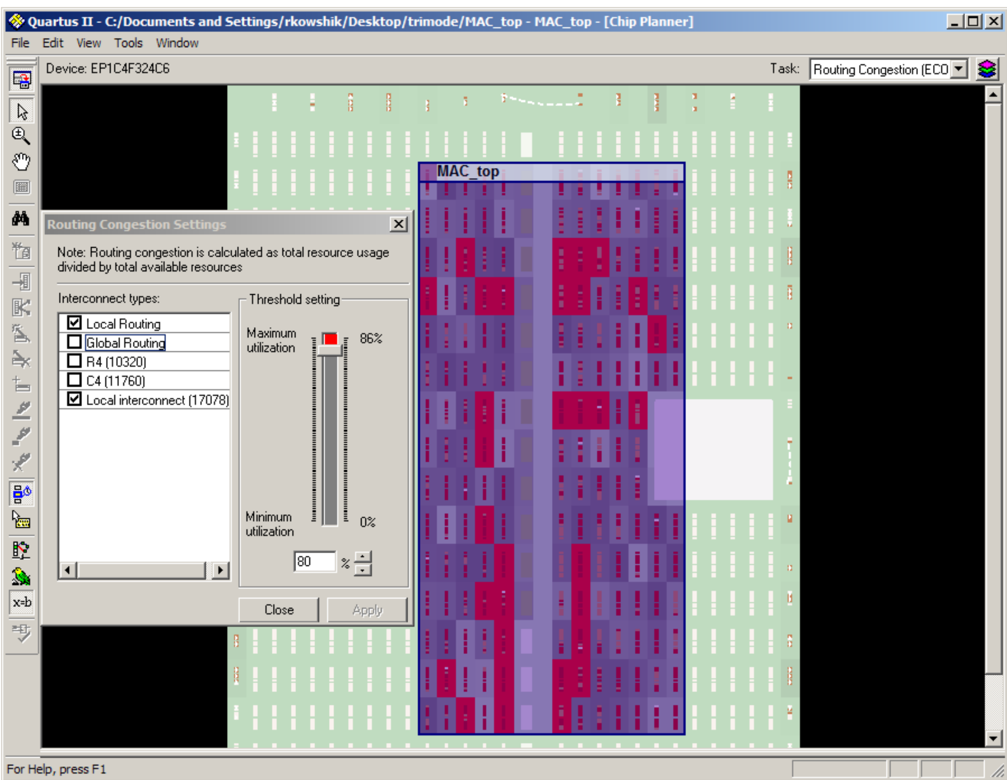
For more information about performing ECOs, refer to the *Engineering Change Management* with the Chip Planner chapter in volume 2 of the *Quartus II Handbook*.

Viewing Routing Congestion

The Routing Congestion view allows you to determine the percentage of routing resources used after a compilation. This feature identifies where there is a lack of routing resources. This information helps you make decisions about design changes that might be necessary to ease the routing congestion and thus meet design requirements. The congestion is visually represented by the color and shading of logic resources. The darker shading represents a greater routing resource utilization.

You can set a routing congestion threshold to identify areas of high routing congestion with the **Routing Congestion Settings** dialog box by selecting the **Routing Congestion (ECO)** task from the drop-down task list or when you select **Routing Utilization** from the layers setting. In the **Routing Congestion Settings** dialog box, set the threshold level for congestion indication and click **Apply**. You can also select the interconnect type you are interested in. All areas that exceed the threshold you have set appear in red (Figure 12–25).

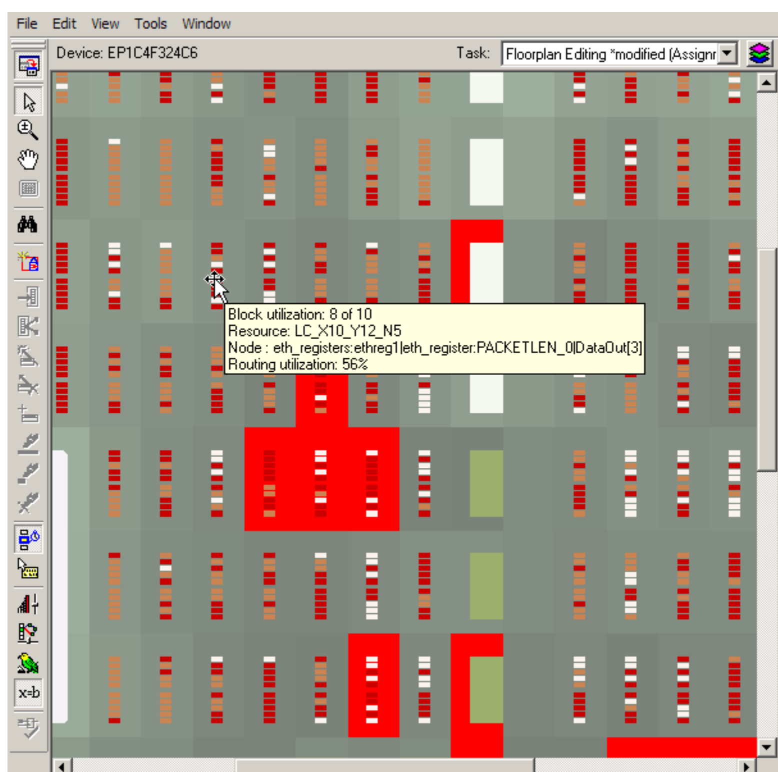
Figure 12–25. Areas Exceeding Threshold



If you are using a HardCopy II device, turn on **Routing Congestion** to see the routing congestion in the device by selecting **Routing Utilization** from the Layers Settings window.

To view the routing congestion in the Chip Planner, click the Layers icon located next to the Task menu. Under **Background Color Map**, select the **Routing Utilization** map (Figure 12–26). Any areas that exceed the threshold appear red. Use this congestion information to evaluate if you could modify the floorplan, or make changes to the RTL to reduce routing congestion.

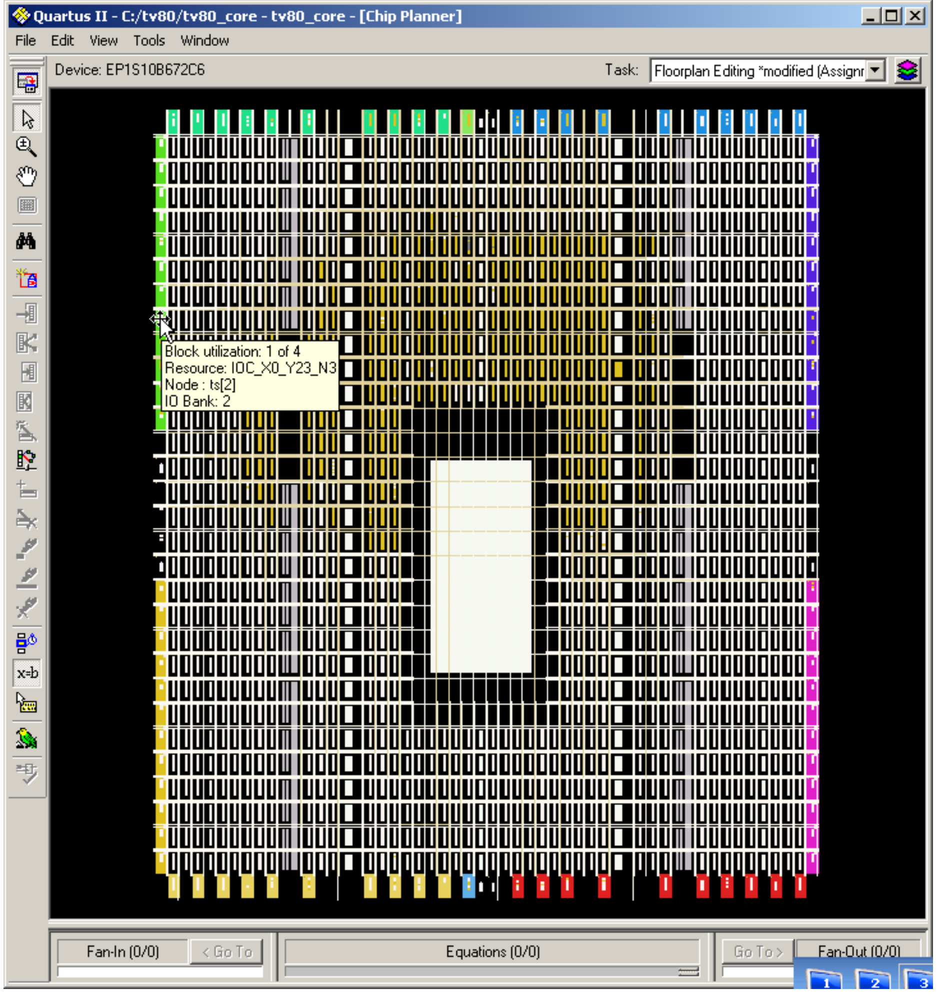
Figure 12–26. Viewing Routing Congestion Map in the Chip Planner



Viewing I/O Banks

The Chip Planner can show all of the I/O banks of the device. To see the I/O bank map of the device, click the Layers icon located next to the Task menu. Under **Background Color Map**, select **I/O Banks**. Refer to Figure 12–27.

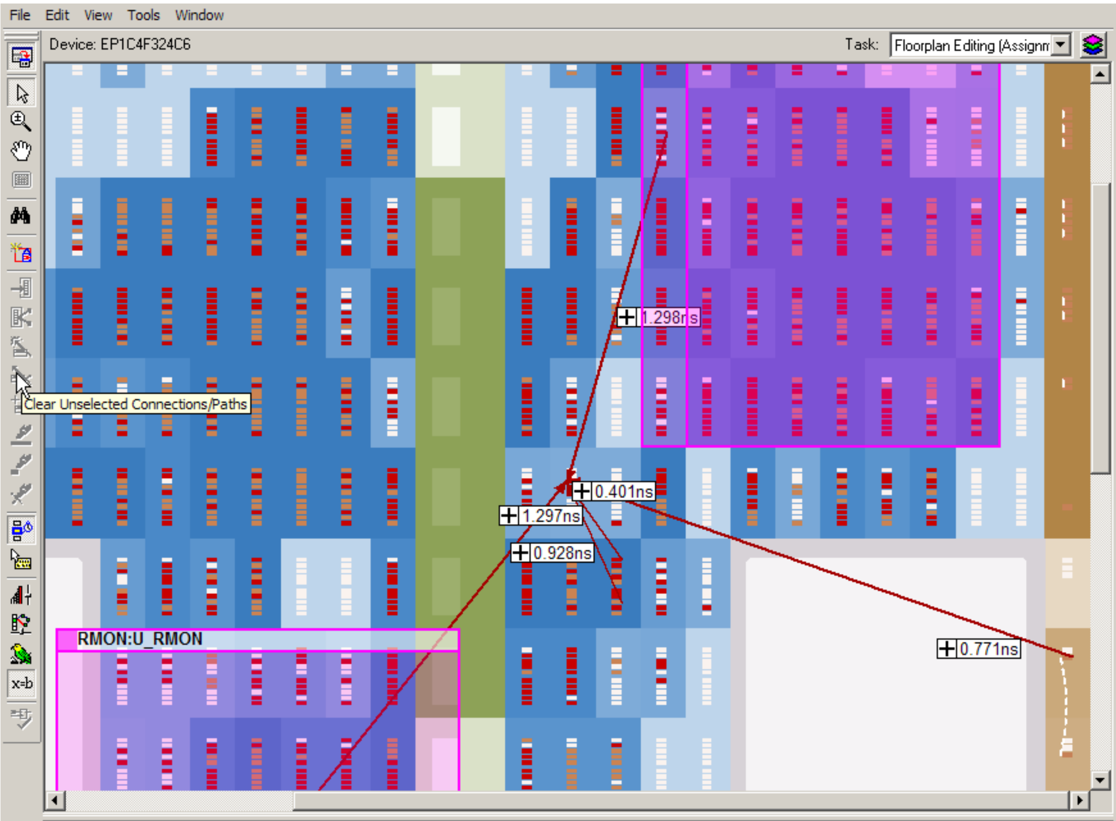
Figure 12–27. Viewing I/O Banks in the Chip Planner



Generating Fan-In and Fan-Out Connections

This feature enables you to view the atoms that fan-in to or fan-out from the selected atom. To remove the connections displayed, use the Clear Unselected Connections/Paths icon in the Chip Planner toolbar. Figure 12–28 shows the fan-in connections for the selected resource.

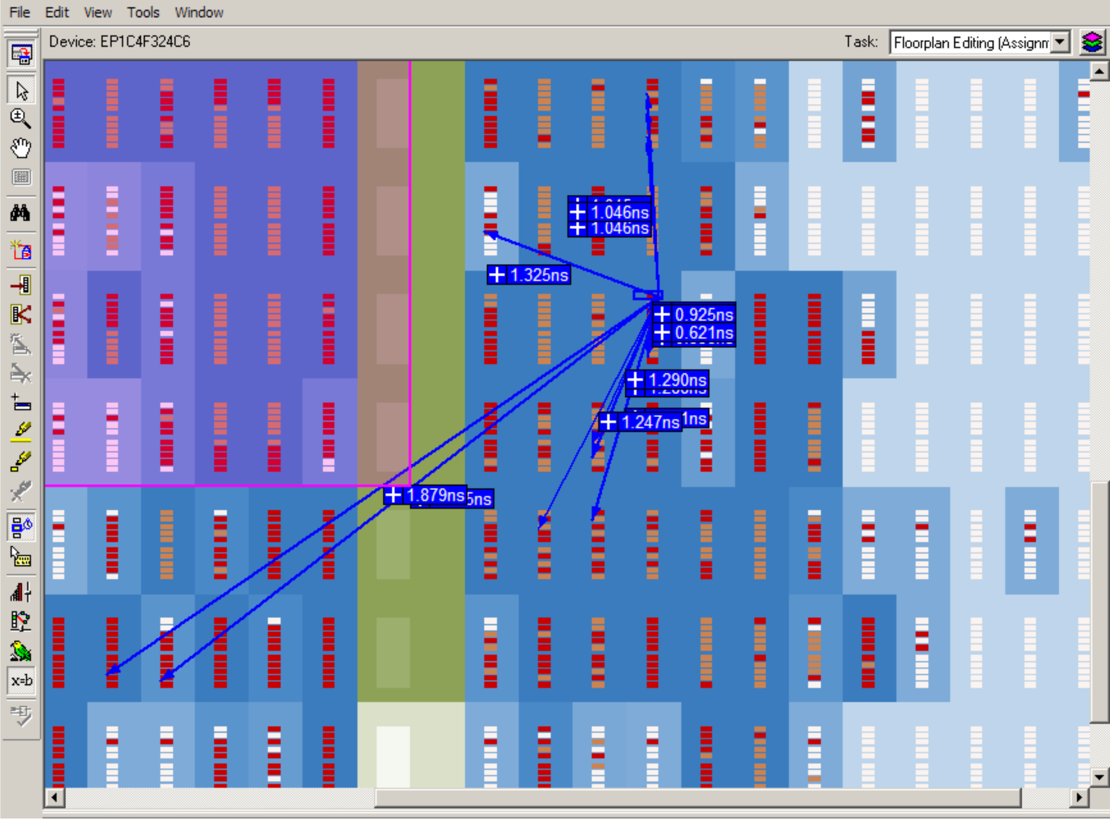
Figure 12–28. Generated Fan-In



Generating Immediate Fan-In and Fan-Out Connections

This feature enables you to view the immediate resource that is the fan-in or fan-out connection for the selected atom. For example, selecting a logic resource and choosing to view the immediate fan-in enables you to see the routing resource that drives the logic resource. You can generate immediate fan-in and fan-outs for all logic resources and routing resources. To remove the connections that are displayed, click the Clear Connections icon in the toolbar. Figure 12–29 shows the immediate fan-out connections for the selected resource.

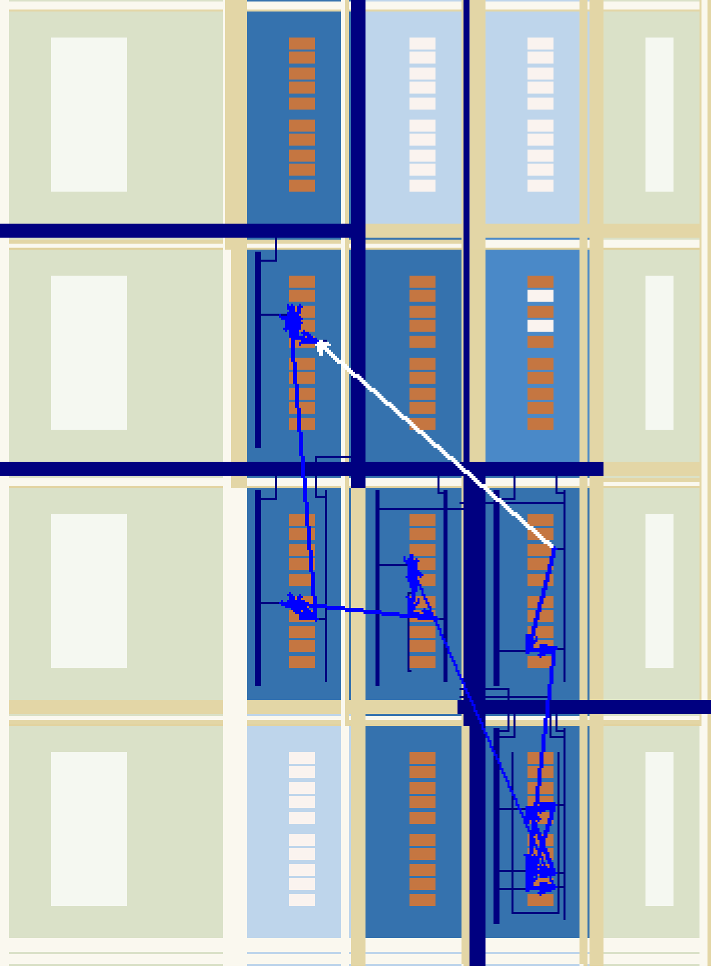
Figure 12–29. Immediate Fan-Out Connection



Highlight Routing

This feature enables you to highlight the routing resources used for a selected path or connection. Figure 12–30 shows the routing resources used between two logic elements.

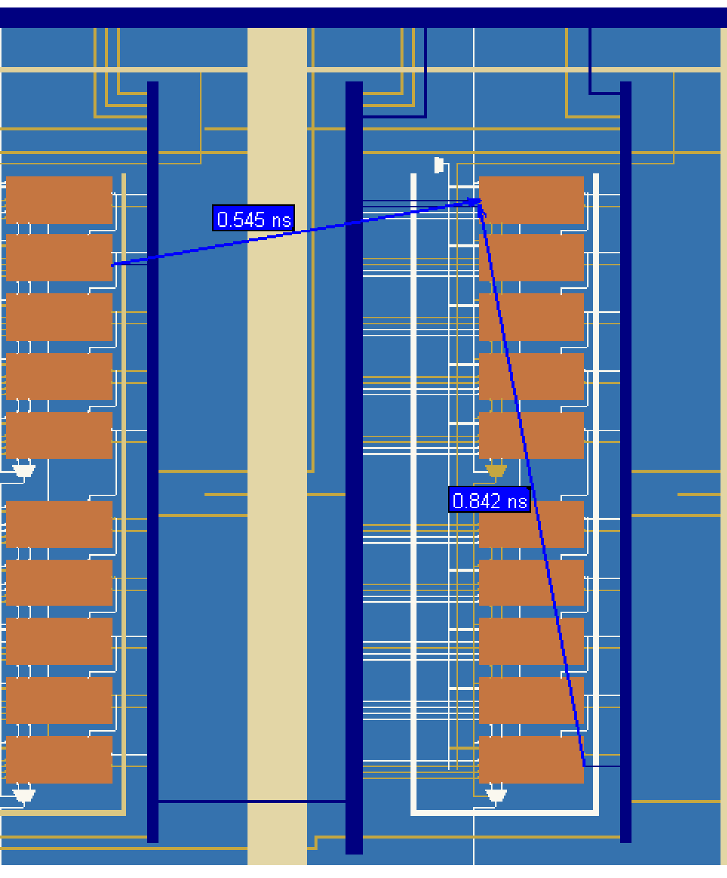
Figure 12–30. Highlight Routing



Show Delays

You can view the timing delays for the highlighted connections when generating connections between elements. For example, you can view the delay between two logic resources or between a logic resource and a routing resource. Figure 12–31 shows the delays between several logic elements.

Figure 12–31. Show Delays



Exploring Paths in the Chip Planner

You can use the Chip Planner to explore paths between logic elements. The following example uses the Chip Planner to traverse paths from the Timing Analysis report.

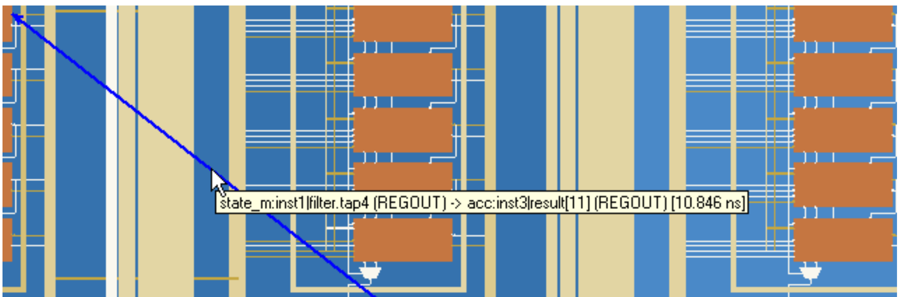
Locate Path from the Timing Analysis Report to the Chip Planner

To locate a path from the Timing Analysis Report to the Chip Planner, perform the following steps:

1. Select the path you want to locate.
2. Right-click the path in the Timing Analysis Report, point to **Locate**, and click **Locate in Chip Planner (Floorplan & Chip Editor)**.

Figure 12–32 shows the path that is displayed in the Chip Planner.

Figure 12–32. Resulting Path

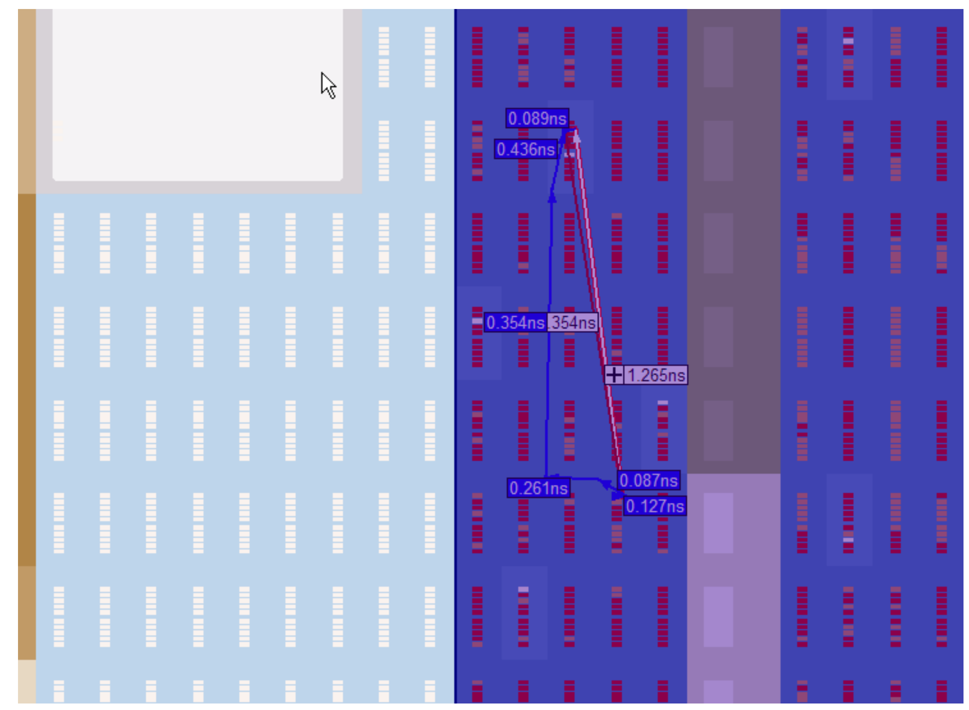


To view the routing resources taken for a path you have located in the Chip Planner, click the Highlight Routing icon in the Chip Planner toolbar, or from the View menu, click **Highlight Routing**.

Analyzing Connections for a Path

To determine the connections between items in the Chip Planner, click the Expand Connections/Paths icon on the toolbar. To add the timing delays between each connection, click the Show Delays icon on the toolbar. Figure 12–33 shows the connections for the selected path that are displayed in the Chip Planner. To see the constituent delays on the selected path, click on the "+" sign next to the path delay displayed in the Chip Planner.

Figure 12–33. Path Analysis

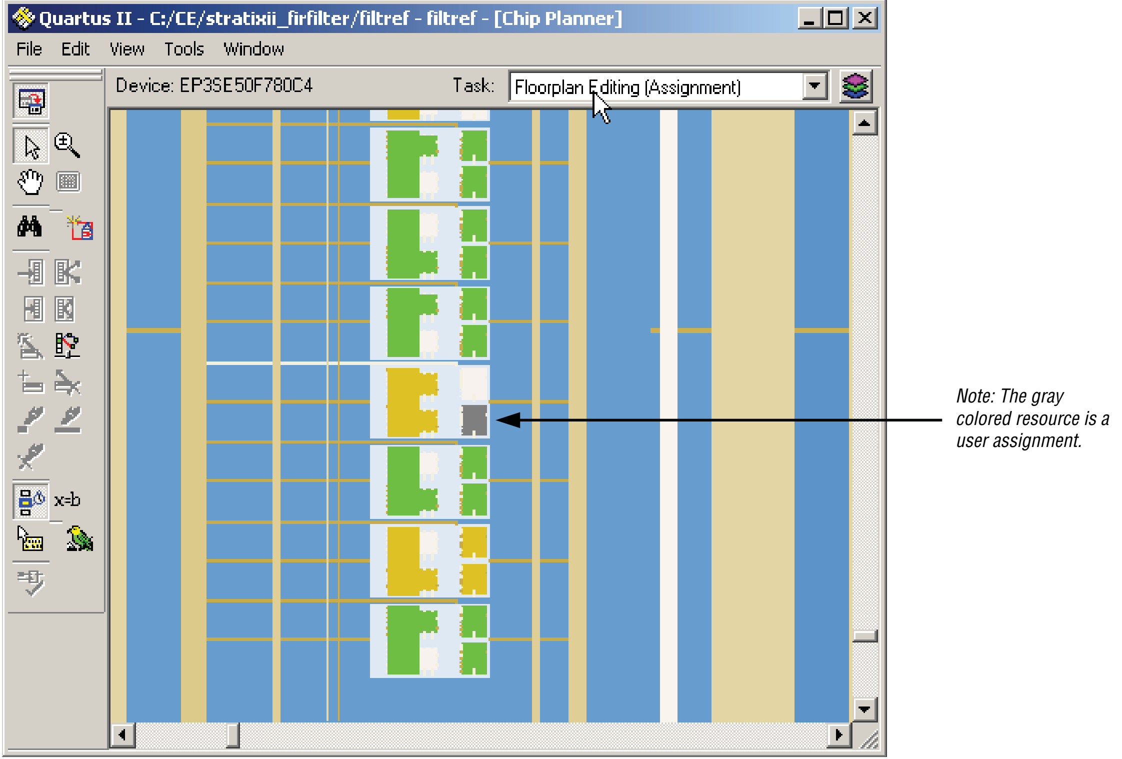


Viewing Assignments in the Chip Planner

Location assignments can be viewed by selecting the appropriate layer set from the tool. To view location assignments in the Chip Planner, select the **Floorplan Editing (Assignment)** task or any custom task with Assignment editing mode. See Figure 12–34.

The Chip Planner shows location assignments graphically, by displaying assigned resources in a particular color (gray, by default). You can create or move an assignment by dragging the selected resource to a new location.

Figure 12–34. Viewing Assignments in the Chip Planner



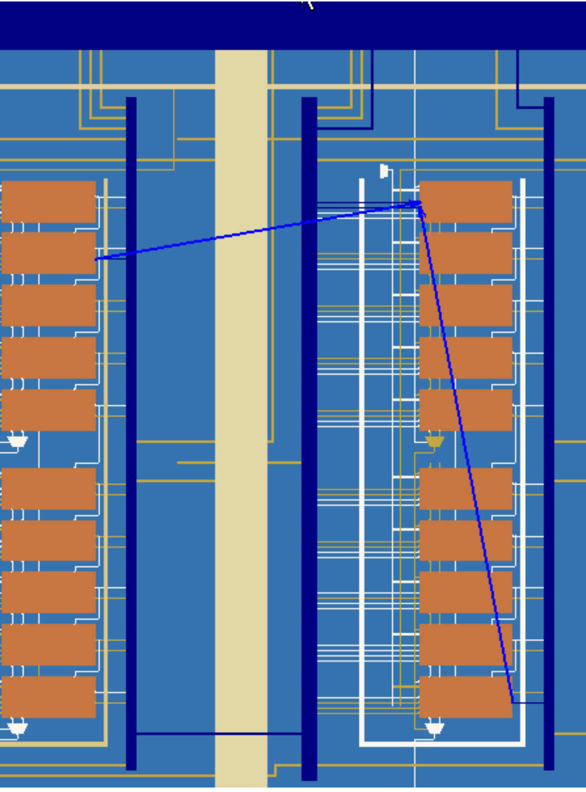
You can make node and pin location assignments and assignments to LogicLock regions and custom regions using the drag-and-drop method in the Chip Planner. The assignments that you create are applied by the Fitter during the next place-and-route operation.

To learn more about working with location assignments, refer to the Quartus II Help.

Viewing Routing Channels for a Path in the Chip Planner

To determine the routing channels between connections, click the Highlight Routing icon on the toolbar. Figure 12–35 shows the routing channels used for the selected path in the Chip Planner.

Figure 12–35. Highlight Routing

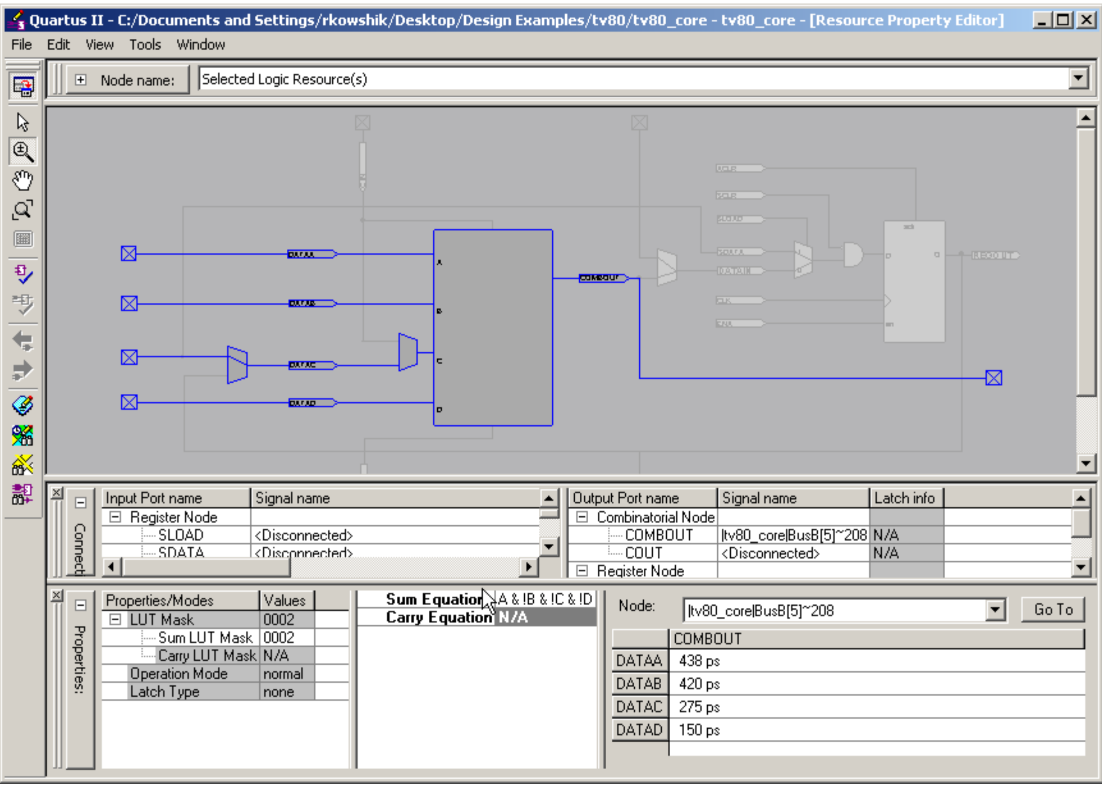


You can view and edit resources in the FPGA using the Resource Property Editor mode of the Chip Planner tool. For more information, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Cell Delay Table

You can view the propagation delay from all inputs to all outputs for any LE in your design. To see the Cell Delay Table for an atom, select the atom in the Chip Planner and right-click. From the pop-up menu, click **Locate** and then click **Locate in Resource Property Editor**. The Resource Property window shows you the atom properties along with the Cell Delay Table, indicating the propagation delay from all inputs to all outputs. Figure 12–36 shows the Cell Delay Table.

Figure 12–36. Cell Delay Table



Timing numbers are displayed only when there is a direct path between the source input port and the destination output port. In cases where there is no path, or the path requires an intermediate buried timing node, the displayed cell delay is given as "N/A."

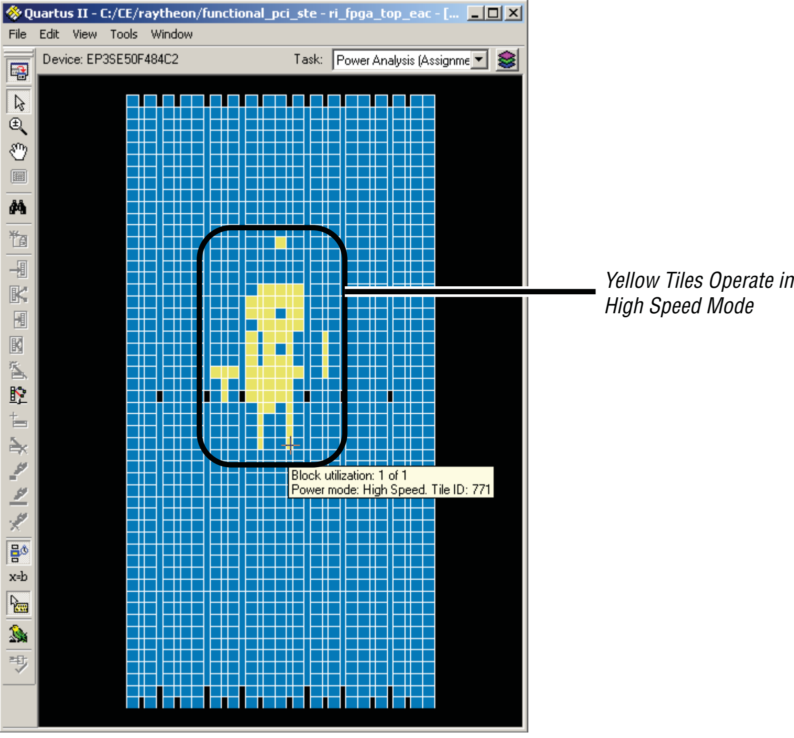
Viewing High-Speed and Low Power Tiles in Stratix III Devices in the Chip Planner

The Chip Planner has a predefined task, **Power Analysis (Assignment)**, which shows the power map of a Stratix III device. Stratix III devices have ALMs that can operate in either high-speed mode or low-power mode. The power mode is set during the fitting process in the Quartus II software. These ALMs are grouped together to form larger blocks, called "tiles."

To learn more about power analyses and optimizations in Stratix III devices, refer to *AN 437: Power Optimization in Stratix III FPGAs*.

When the **Power Analysis (Assignment)** task is selected in the Chip Planner for Stratix III devices, low-power and high-speed tiles are displayed in different colors; yellow tiles operate in a high-speed mode, while blue tiles operate in a low-power mode (see Figure 12–37). In this mode, you can perform all floorplan-related functions for this task; however, you cannot edit any tiles to change the power mode.

Figure 12–37. Viewing High-Speed and Low Power Tiles in a Stratix III Device



Design Analysis Using the Timing Closure Floorplan

For older device families not supported by the Chip Planner, you can perform floorplan analysis using the Timing Closure Floorplan. The APEX, ACEX, FLEX 10K, and MAX 7000 device families are supported only by the Timing Closure Floorplan. This section explains how to use the Timing Closure Floorplan to enhance your FPGA design analysis.

Table 12–1 on page 12–1 lists the device families supported by the Timing Closure Floorplan Editor and the Chip Planner.

The Timing Closure Floorplan Editor allows you to analyze your design visually before and after performing a full design compilation in the Quartus II software. This floorplan editor, used in conjunction with the Quartus II Timing Analyzer, provides a method for performing design analysis.

To start the Timing Closure Floorplan Editor, on the Assignments menu, click **Timing Closure Floorplan**.

If the device in your project is not supported by the Timing Closure Floorplan, the following message appears: `Can't display a floorplan: the current device family is only supported by Chip Planner`.

If your target device is supported by the Timing Closure Floorplan, you can also start the Timing Closure Floorplan tool by right-clicking any of the following sources, pointing to **Locate**, and clicking **Locate in Timing Closure Floorplan**:

- Compilation Report
- Node Finder
- Project Navigator
- RTL source code
- RTL Viewer
- Simulation Report
- Timing Report

Figure 12–38 shows the icons in the Timing Closure Floorplan toolbar.

Figure 12–38. Timing Closure Floorplan Icons

Timing Closure Floorplan Views

The Timing Closure Floorplan Editor provides the following views of your design:

- Field view
- Interior Cells view
- Interior LAB view
- Interior MegaLABs view

Additionally, the following two views are available, which open up the Pin Planner:

- Package Top view
- Package Bottom view

Field View

The Field view provides a color-coded, high-level view of the resources used in the device floorplan. All device resources, such as embedded system blocks (ESBs) and MegaLAB blocks, are outlined.

To view the details of a resource in the Field view, select the resource, right-click, and click **Show Details**. To hide the details, select all the resources, right-click, and click **Hide Details** (Figure 12–39).

Figure 12–39. Show and Hide Details of a Logic Array Block in Field View

Other Views

You can also view your design in the Timing Closure Floorplan Editor with the Interior Cells, Interior LABs, Package Top, and Package Bottom views. Use the View menu to display the various floorplan views. The Interior Cells view provides a detailed view of device resources, including device pins and individual logic elements within a MegaLAB.

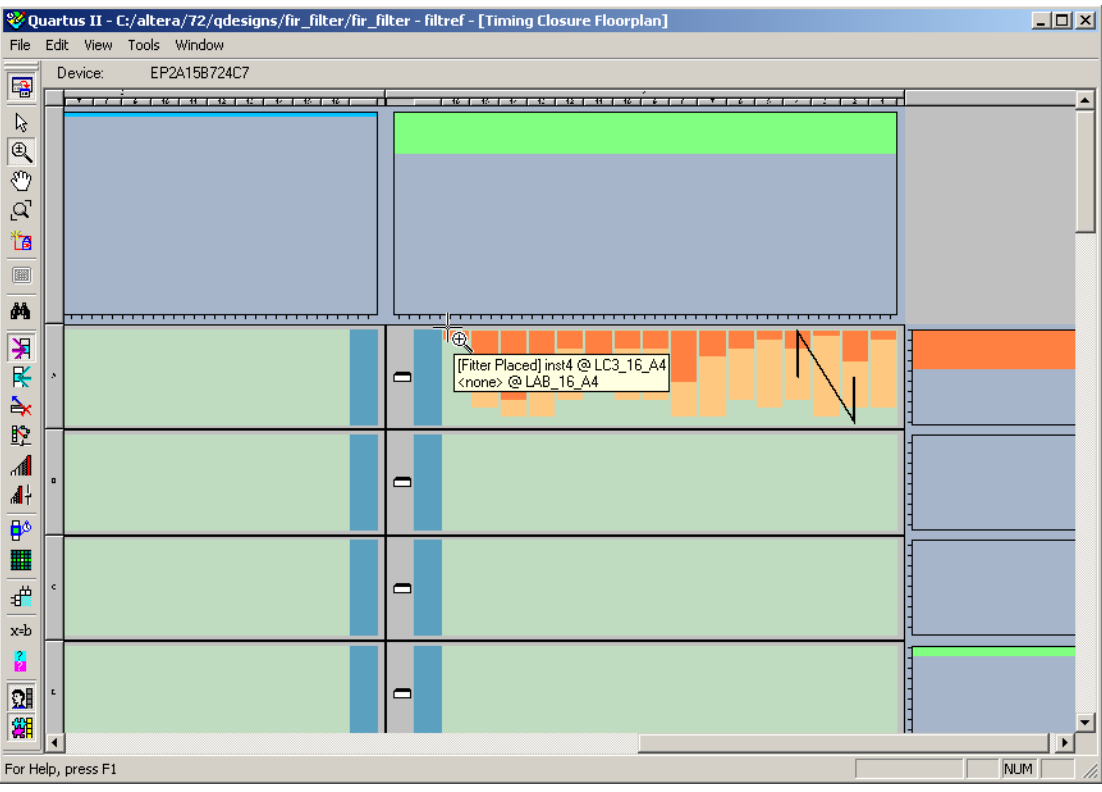
Viewing Assignments

The Timing Closure Floorplan Editor differentiates between user assignments and Fitter placements. User assignments include LogicLock regions and are made by a user.

If the device is changed after a compilation, the user assignment and Fitter placement options cannot be used together. When this situation occurs, the Fitter placement displays the last compilation result and the user assignment displays the floorplan of the newly selected device.

To see the user assignments, click the Show User Assignments icon in the Floorplan Editor toolbar, or, on the View menu, point to **Assignments** and click **Show User Assignments**. To see the Fitter placements, click the Show Fitter Placements icon in the Floorplan Editor toolbar, or, on the View menu, point to **Assignments** and click **Show Fitter Placements**. Figure 12–40 shows the Fitter placements.

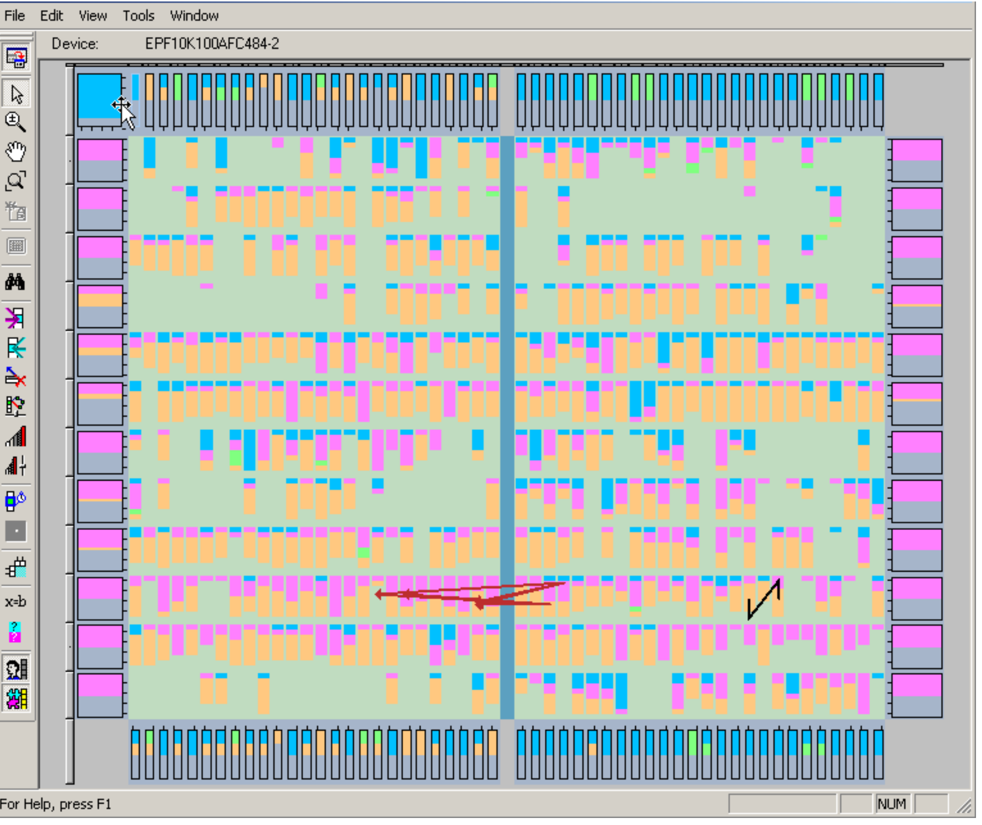
Figure 12–40. Fitter Placements



Viewing Critical Paths

The View Critical Paths feature displays routing paths in the floorplan, as shown in Figure 12–41. The criticality of a path is determined by its slack and is also shown in the timing analysis report.

Figure 12–41. Critical Paths



To view critical paths in the Timing Closure Floorplan, click the Critical Path Settings icon on the toolbar, or, on the View menu, point to **Routing** and click **Critical Path Settings**.

When viewing critical paths, you can specify the clock in the design to be viewed. You can determine which paths to display by specifying the slack threshold in the slack field.

Timing settings must be made and timing analysis performed for paths to be displayed in the floorplan.

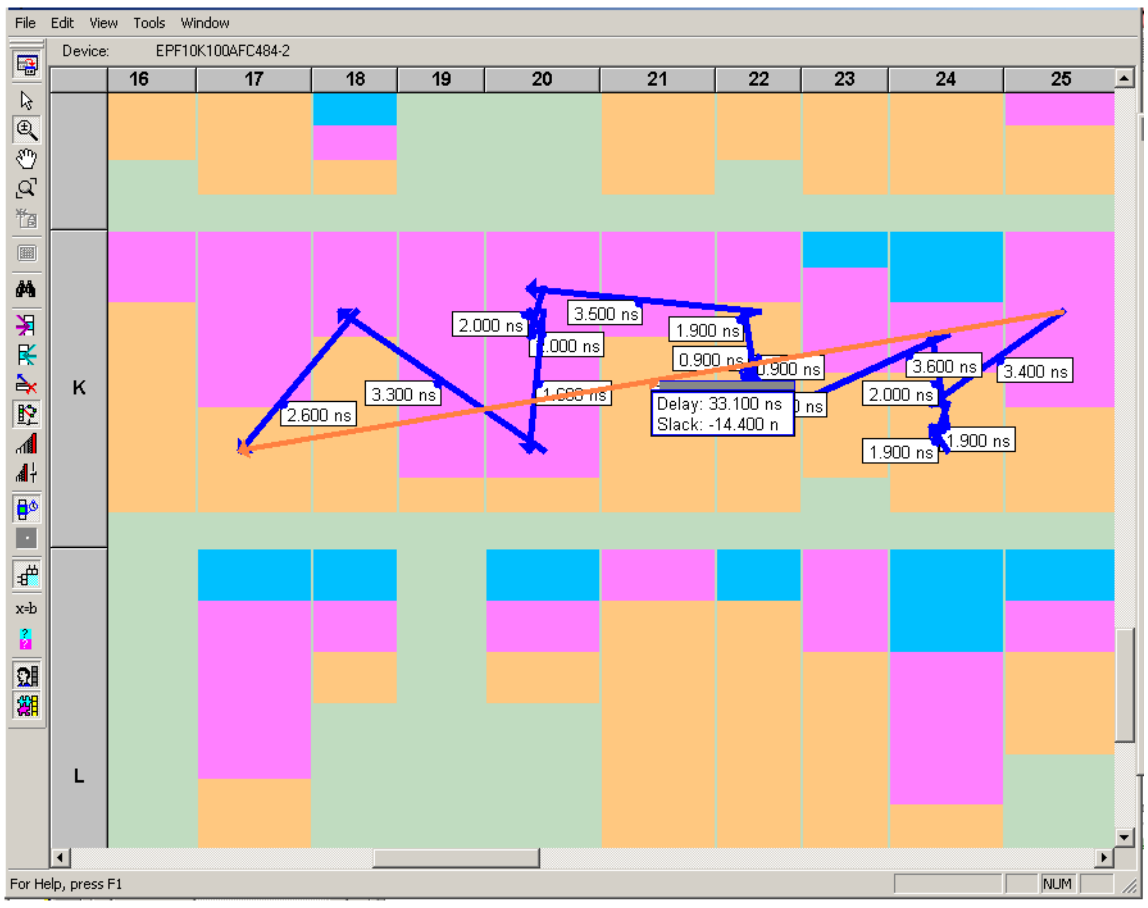
For more information about performing static timing analyses of your design with a timing analyzer, refer to the *Quartus II Classic Timing Analyzer* and the *Quartus II TimeQuest Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*.

Viewing the critical paths is useful for determining the criticality of nodes based on placement. There are a number of ways to view the details of the critical path.

The default view in the Timing Closure Floorplan shows the path with the source and destination registers displayed. You can also view all the combinational nodes along the worst-case path between the source and destination nodes. To view the full path, click on the delay label to select the path, right-click, and select **Show Path Edges**. Figure 12–42 shows the critical path through combinational nodes. To hide the combinational nodes, select the path, right-click, and select **Hide Path Edges**.

You must view the routing delays to select a path.

Figure 12–42. Worst-Case Combinational Path Showing Path Edges



To assign the path to a LogicLock region using the **Paths** dialog box, select the path, right-click, and select **Properties**.

You can determine the maximum routing delay between two nodes within a LogicLock region. To use this feature, on the View menu, point to **Routing** and click **Show Intra-region Delay**. Place your cursor over a Fitter-placed LogicLock region to see the maximum delay.

For more information about making path assignments with the **Paths** dialog box, refer to "Timing Closure Floorplan View" on page 12–48.

After running timing analysis, you can locate timing paths from the timing reports file produced. Right-click on any row in the report file, point to **Locate**, and click **Locate in Timing Closure Floorplan**. The Timing Closure Floorplan window opens with the timing path highlighted.

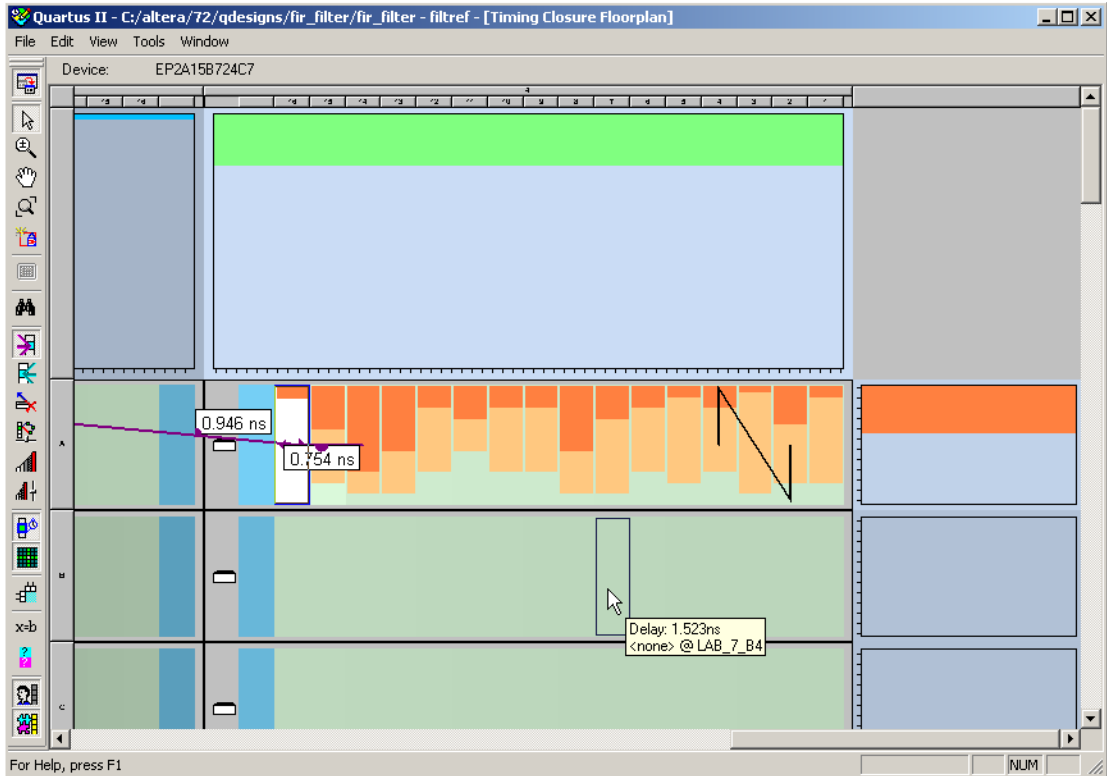
For more information about optimizing your design in the Quartus II software, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*. With the options and tools available in the Timing Closure Floorplan and the techniques described in that chapter, the Quartus II software can help you achieve timing closure in a more time-efficient manner.

Physical Timing Estimates

In the Timing Closure Floorplan Editor, you can select a resource and see the approximate delay to any other resource on the device. After you select a resource, the delay is represented by the color of potential destination resources. The darker the color of the resource, the longer the delay.

You can also obtain an approximation of the delay between two points by selecting a source and holding your cursor over a potential destination resource (Figure 12–43).

Figure 12–43. Delay for Physical Timing Estimate in the Timing Closure Floorplan



The delays represent an estimate based on probable best-case routing. The delay can be greater than what is shown, depending on the availability of routing resources. In general, there is a strong correlation between the probable and actual delay.

To view the physical timing estimates, click the Show Physical Timing Estimate icon, or, on the View menu, point to **Routing** and click **Show Physical Timing Estimates**.

You can use the physical timing estimate information when manually placing logic in a device. This information allows you to place critical nodes and modules closer together, and non-critical or unrelated nodes and modules further apart, reducing the routing congestion between critical and non-critical entities and modules. This placement enables the Quartus II Fitter to meet the timing requirements.

Viewing Routing Congestion

The View Routing Congestion feature allows you to determine the percentage of routing resources used after a compilation. This feature identifies where there is a lack of routing resources.

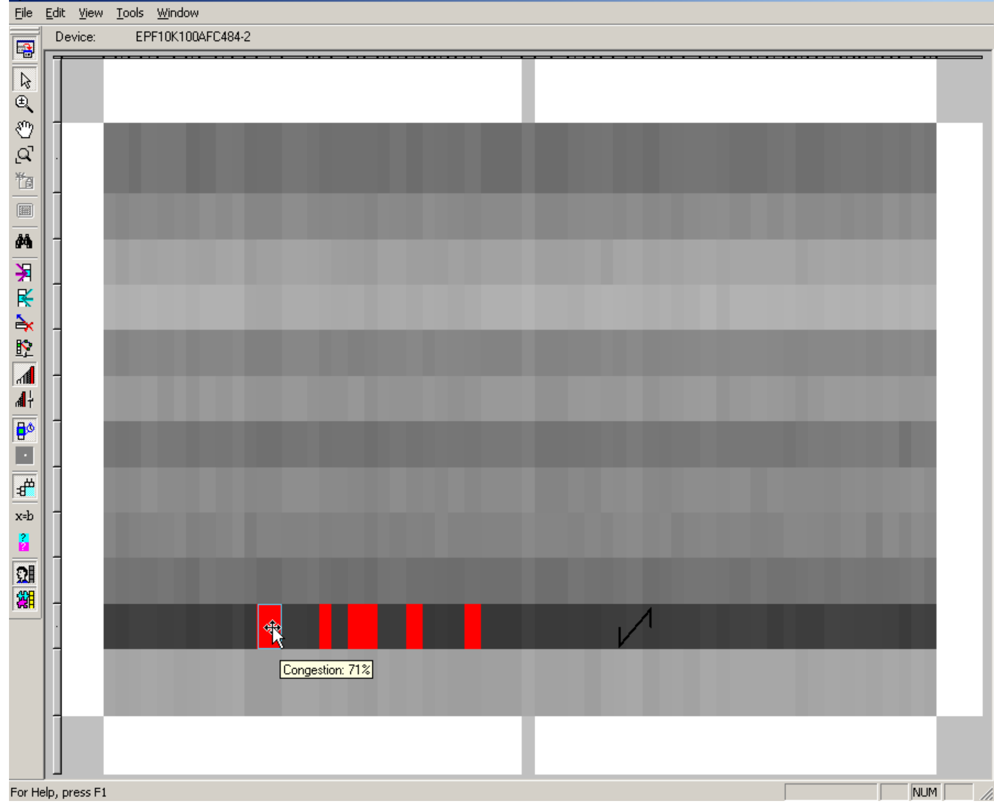
The congestion is shown by the color and shading of logic resources. The darker shading represents a greater routing resource utilization. Logic resources that are red have routing resource utilization greater than the specified threshold.

The routing congestion view is only available from the View menu when you enable the Field view. To view routing congestion in the floorplan, click the Show Routing Congestion icon, or on the View menu, point to **Routing** and click **Show Routing Congestion**. To set the criteria for the critical path you want to view, click the Routing Congestion Settings icon, or on the View menu, point to **Routing** and click **Routing Congestion Settings**.

In the **Routing Congestion Settings** dialog box, you can choose the routing resource (interconnect type) you want to examine and set the congestion threshold. Routing congestion is calculated based on the total resource usage divided by the total available resources.

If you use the routing congestion viewer to determine where there is a lack of routing resources, examine each routing resource individually to determine which ones use close to 100% of the available resources (Figure 12–44). Use this congestion information to evaluate whether you should modify the floorplan, or make changes to the RTL to reduce routing congestion.

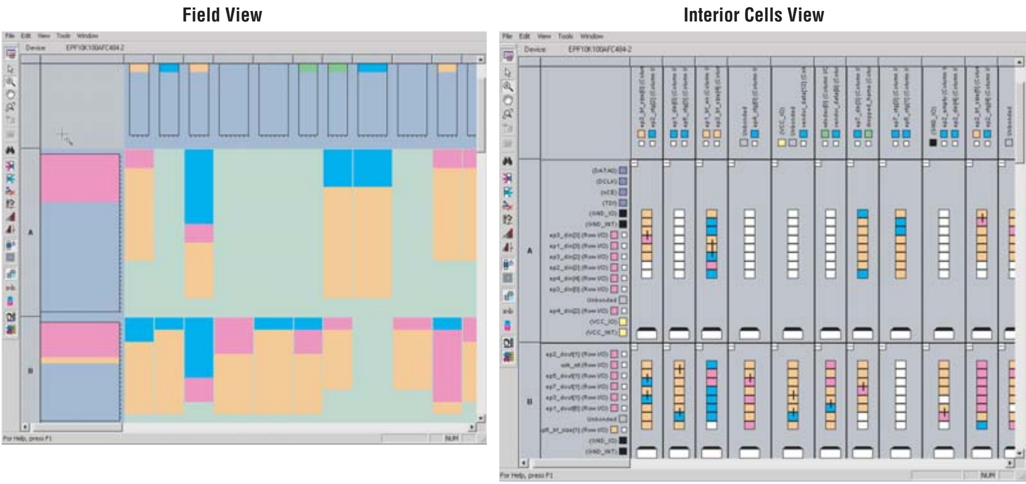
Figure 12–44. Routing Congestion of a Sample Design in a Cyclone Device



Timing Closure Floorplan View

The Timing Closure Floorplan view provides you with current and previous compilation assignments on one screen. You can display device resources in either of two views: the Field view or the Interior Cells view, as shown in Figure 12–45. The Field view provides an uncluttered view of the device floorplan in which all device resources, such as embedded system blocks (ESBs) and MegaLAB blocks, are outlined. The Interior Cells view provides a detailed view of device resources, including individual logic elements within a MegaLAB and device pins.

Figure 12–45. Timing Closure Floorplan Editor



LogicLock Regions in the Timing Closure Floorplan

If you have defined a LogicLock region in a device supported by the Timing Closure Floorplan, then you must assign resources to it using the Timing Closure Floorplan, the LogicLock Regions dialog box, or a Tcl script.

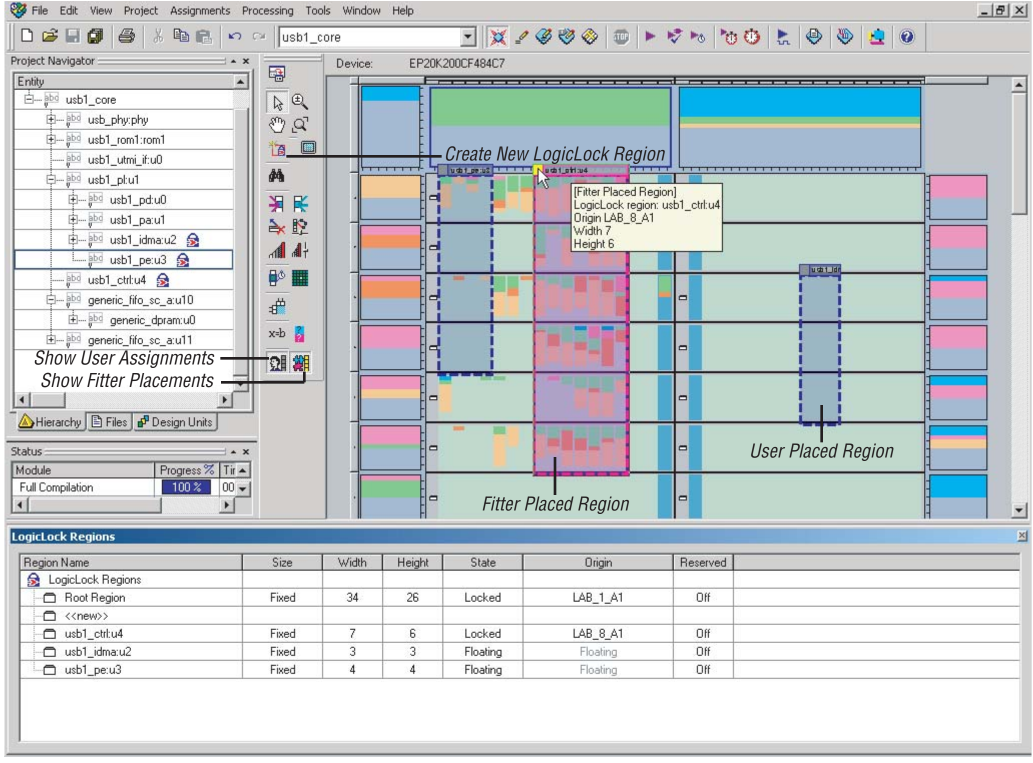
Creating LogicLock Regions in the Timing Closure Floorplan Editor

The Timing Closure Floorplan Editor has toolbar buttons that are used to manipulate LogicLock regions as shown in Figure 12–46. You can use the Create New LogicLock Region icon to draw LogicLock regions in the device floorplan.

The Timing Closure Floorplan Editor displays LogicLock regions when you click the Show User Assignments or Show Fitter Placements icons. The type of region determines its appearance in the floorplan.

The Timing Closure Floorplan Editor differentiates between user assignments and Fitter placements. When the **Show User Assignments** option is turned on in the Timing Closure Floorplan, you can see current assignments made to a LogicLock region. When the **Fitter Placement** option is turned on, you can see the properties of the LogicLock region after the most recent compilation. User-assigned LogicLock regions appear in the Timing Closure Floorplan Editor with a dark blue border. Fitter-placed regions appear in the Timing Closure Floorplan Editor with a magenta border (Figure 12–46).

Figure 12–46. Timing Closure Floorplan Editor Toolbar Buttons



Using Drag and Drop to Place Logic

You can drag selected logic displayed in the **Hierarchy** tab of the Project Navigator, in the Node Finder, or in a schematic design file, and drop it into the Timing Closure Floorplan or the **LogicLock Regions** dialog box. Figure 12–14 on page 12–18 shows logic that has been dragged from the **Hierarchy** tab of the Project Navigator and dropped into a LogicLock region in the Timing Closure Floorplan.

Rubber Banding

On the View menu, click **Routing**, and select **Rubber Banding** to show existing connections between LogicLock regions and nodes during movement of LogicLock regions within the Timing Closure Floorplan Editor.

Show Connection Count

You can determine the number of connections between LogicLock regions by turning on the **Show Connection Count** option.

Analyzing LogicLock Region Connectivity Using the Timing Closure Floorplan

To see how logic in LogicLock regions is connected, view the connectivity between LogicLock regions. This capability is extremely useful when entities are assigned to LogicLock regions. You can also see the fan-in and fan-out of selected LogicLock regions.

To view the connections in the Timing Closure Floorplan, on the View menu, point to **Routing** and click **Show LogicLock Regions Connectivity**. Figure 12–47 shows standard LogicLock region connections.

Figure 12–47. LogicLock Region Connections with Connection Count

File Edit View Tools Window
Device: EP2A15B724C7
usb_tx
19
generic_dpram
usb_rx_r
1
54
usb1_ctrl:u4
For Help, press F1
NUM

As shown in Figure 12–47, the thickness of the connection line indicates how many connections exist between regions. To see the number of connections between regions, on the View menu, point to **Routing** and click **Show Connection Count**.

LogicLock region connectivity is applicable only when the user assignments are enabled in the Timing Closure Floorplan. When you use floating LogicLock regions, the origin of the user-assigned region is not necessarily the same as that of the Fitter-placed region. You can change the origin of your floating LogicLock regions to that of the most recent compilation either in the LogicLock Regions window or by selecting **Back-Annotate Origin and Lock** under **Location** in the **LogicLock Regions Properties** dialog box.

To view the fan-in or fan-out of a LogicLock region in the Timing Closure Floorplan, select the user-assigned LogicLock region while the fan-in or the fan-out option is turned on.

To turn on the **fan-in** option, click the Show Node Fan-In icon, or, on the View menu, point to **Routing** and click **Show Node Fan-In**. To turn on the fan-out option, click the Show Node Fan-Out icon, or on the View menu, point to **Routing** and click **Show Node Fan-Out**. Only the nodes that have user assignments are visible when viewing fan-in or fan-out of LogicLock regions.

Using LogicLock Methodology for Older Device Families

The LogicLock methodology is recommended as an optimization method only for older device families, such as the MAX II and APEX II device families, which do not support incremental compilation. In this methodology, you optimize your design and lock it down one module at a time. With the LogicLock feature, you can create and implement each logic module independently in a hierarchical or team-based design. You can use this LogicLock methodology to optimize and preserve timing, placement, or both for devices that do not support incremental compilation.

Using the LogicLock methodology as an optimization strategy is less effective on newer device families such as those device families in the Cyclone and Stratix series of devices. Altera does not recommend using the LogicLock methodology for designs in such devices, although the feature can be supported on some devices in these series. However, you can use LogicLock regions in conjunction with incremental compilation to create a floorplan and preserve timing results for these devices in the Stratix and Cyclone series of devices.

For more information about hierarchical and team-based design, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

The Quartus II LogicLock Methodology

When you use the LogicLock methodology for older devices, you can place the logic in each netlist file in a fixed or floating region in an Altera device. You can then maintain the placement and, if necessary, the routing of your blocks in the Altera device, thus retaining performance.

If you want to follow this methodology, create a LogicLock region in a supported device and then assign logic to the region. After you optimize the logic placed within the boundaries of a region to achieve the required performance, you must back-annotate the region's contents to lock the logic placement and routing. Locking the placement and routing preserves the performance when you integrate the region with the rest of the design.

Figure 12–48 compares the traditional design flow with the LogicLock design flow.

Figure 12–48. Traditional Design Flow Compared with Quartus II LogicLock Design Flow

Traditional Design Flow

Design → Integrate → Optimize → Verify

LogicLock Design Flow

Design, Optimize & Verify → Integrate → Verify

For more information about block-based design with the LogicLock feature, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Improving Design Performance

LogicLock methodology helps you optimize and preserve performance. You can use the LogicLock regions to place modules, entities, or any group of logic into regions in a device's floorplan. LogicLock assignments can be hierarchical, which allows you to have more control over the placement and performance of each module and group of modules.

In addition to hierarchical blocks, you can apply LogicLock constraints to individual nodes; for example, you can make a wildcard path-based LogicLock assignment on a critical path. This technique is useful if the critical path spans multiple design blocks.

Although LogicLock constraints can improve performance, they can also degrade performance if they are not applied correctly. They can also result in increased resource usage.

LogicLock Restrictions

During the design process, placing restrictions on nodes or entities in the design is often necessary. These restrictions can conflict with the node or entity assignments for a LogicLock region. To avoid conflicts, consider the order of precedence given to constraints by the Quartus II software during fitting. The following assignments have priority over LogicLock region assignments:

- Assignments to device resources and location assignments
- Fast input-register and fast output-register assignments
- Local clock assignments for Stratix devices
- Custom region assignments
- I/O standard assignments

The Quartus II software removes nodes and entities from LogicLock regions if any of these constraints are applied to them.

After a LogicLock region is back-annotated, the Quartus II software can place the region only in areas of the device with exactly the same resources.

You can compare the performance with and without some of the LogicLock back-annotated node assignments without deleting the LogicLock regions when you optimize your design with the LogicLock flow. Select the specific LogicLock region from the LogicLock regions window, right-click and select **Properties**. Go to the **Content Back-annotation** tab and turn on **Disable back-annotated node locations**.

Preserving Timing Results Using the LogicLock Flow

To preserve the timing results for a design module in the Quartus II software, you must preserve the placement and routing information for all the logic in the design module. For older device families, you can use the LogicLock design methodology to back-annotate logic locations within a LogicLock region, which makes assignments to each node in the design.

When preserving logic placement in an Altera device using LogicLock back-annotation, an atom netlist preserves the node names in subblocks of your design. An atom netlist contains design information that fully describes the submodule logic in terms of the device architecture. In the atom netlist, the nodes are fixed as Altera primitives and the node names do not change if the atom netlist does not change. If a node name changes, any placement information associated with that node, such as LogicLock assignments made when back-annotating a region, is invalid and ignored by the compiler.

If all the netlists are contained in one Quartus II project, use the LogicLock flow to back-annotate the logic in each region. If a design region changes, only the netlist associated with the changed region is affected. When you place and route the design using the Quartus II software, the software has to re-fit only the LogicLock region associated with the changed netlist file.

Turn on the **Prevent further netlist optimization** option when back-annotating a region with the **Synthesis Netlist Optimizations** option, the **Physical Synthesis Optimization** option, or both, turned on. This sets the **Netlist Optimizations** option to **Never Allow** for all nodes in the region, preventing node name changes in the top-level design when the region is recompiled.

You must remove previously back-annotated assignments for a modified block if the node names are different in the newly synthesized version. When you recompile with one new netlist file, the placement and assignments for the unchanged netlist files assigned to other LogicLock regions are not affected. Therefore, you can make changes to code in an independent block and not interfere with another designer's changes, even when all the blocks are integrated into the same top-level design.

With the LogicLock design methodology, you can develop and test submodules without affecting other areas of a design.

For the Quartus II software to achieve optimal placement, you must specify timing assignments, including t_{SU}, t_{CO}, and t_{PD}, for all clock signals in the design.

To facilitate the LogicLock design flow, the Timing Closure Floorplan highlights resources that have back-annotated LogicLock regions.

Atom Netlist Design Information

The atom netlist contains design information that fully describes the module's logic in terms of an Altera device architecture. If the design was synthesized using a third-party tool and then brought into the Quartus II software, an atom netlist already exists and all the nodes have assigned names. You do not have to generate another atom netlist. However, if you use any synthesis netlist optimizations or physical synthesis optimizations, you must generate a Verilog Quartus Mapping Netlist File (**.vqm**) using the Quartus II software, because the original atom netlist might have changed as a result of these optimizations.

Turn on the **Prevent further netlist optimization** option when back-annotating a region with the **Synthesis Netlist Optimizations** option, the **Physical Synthesis Optimization** option, or both, turned on. This sets the **Netlist Optimizations** to **Never Allow** for all nodes in the region, avoiding the possibility of a node name change when the region is imported into the top-level design.

If you synthesized the design from a VHDL Design File (**.vhd**), Verilog Design File (**.v**), Text Design File (**.tdf**), or Block Design File (**.bdf**) in the Quartus II software, you must also create an atom netlist to fix the node names. During compilation, the Quartus II software creates a **.vqm** file in the **atom_netlists** subdirectory in the project directory.

If the atom netlist is generated by a third-party synthesis tool and the design has a black box library of parameterized modules (LPM) functions or Altera megafunctions, you must generate a Quartus II **.vqm** file for the black box modules.

For instructions about creating an atom netlist in the Quartus II software, refer to *Saving Synthesis Results in a Verilog Quartus Mapping File* in the Quartus II Help.

Placement Information

The **.qsf** file contains the module's LogicLock constraint information, including clock settings, pin assignments, and relative placement information for back-annotated regions. To maintain performance, you must back-annotate the module.

Routing Information

The **.rcf** file located in your project directory contains the module's LogicLock routing information. To maintain performance, you must back-annotate the module. When you export LogicLock regions with the **Export back-annotated routing** option, the Quartus II software automatically places the **.rcf** file in the **atom_netlists** directory.

Back-Annotating Routing Information

LogicLock regions not only allow you to preserve the placement of logic from one compilation to the next, they also allow you to retain the routing inside the LogicLock regions. With both placement and routing locked, you have an extremely portable design module that can be used many times in a top-level design without requiring further optimization.

If you back-annotate the routing in a LogicLock region with auto size and floating properties, the Quartus II software automatically changes the region properties to **Fixed** and **Locked**. The reason for this property change is that routing resources are generally different in different locations of the Altera device.

Back-annotate routing only when necessary, because doing so can prevent the Quartus II Fitter from finding an optimal fit for your design.

Back-annotate the routing from the Assignments menu, by choosing **Routing** from the **Back-Annotate Assignments** dialog box.

If you are not using an atom netlist, you must turn on the **Save a node-level netlist of the entire design into a persistent source file** option (on the Assignments menu, click **Back-Annotate Assignments**) if back-annotation of routing is selected. Writing out a **.vqm** file causes the Quartus II software to enforce persistent naming of nodes when saving the routing information. The **.vqm** file is then used as the design's source.

Back-annotated routing information is valid only for regions with fixed sizes and locked locations. The Quartus II software ignores the routing information for LogicLock regions you specify as floating and automatically sized.

The **Disable Back-Annotated Node locations** option in the **LogicLock Region Properties** dialog box is not available if the region contains both back-annotated routing and back-annotated nodes.

Back-Annotating LogicLock Regions

To back-annotate the contents of your LogicLock regions, perform the following steps:

1. In the **LogicLock Region Properties** dialog box, click **Back-Annotate Contents**. The **Back-Annotate Assignments** dialog box appears.
2. In the **Back annotation type** list, select **Advanced** and click **OK**.
3. Click **OK**.

If you are using the incremental compilation flow, logic back-annotation is not required. Preserve placement results using the Post-Fit Netlist Type instead of making placement assignments with back-annotation as described in this section.

You can also back-annotate routing within LogicLock regions to preserve performance of the regions. For more information about back-annotating routing, refer to "Back-Annotating Routing Information" on page 12–55.

Back-annotation is not supported for newer device families, such as the Stratix IV GX, Stratix III, and Cyclone III devices.

When you back-annotate a region's contents, all of the design element nodes appear under **Back-annotated nodes** with an assignment to a device resource under **Node Location**; for example, LAB, memory block, or DSP block. Each node's location is the placement of the node after the most recent compilation. If the origin of the region changes, the node's location changes to maintain the same relative placement. This relative placement preserves the performance of the module. If cell assignments are demoted, the nodes are assigned to LABs rather than directly to logic cells. This provides more flexibility to the Fitter, and improves the chances of a fit.

Scripting Support

You can run procedures and create the settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II command-line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The same information is available in the Quartus II Help, and in PDF format in the *Quartus II Scripting Reference Manual*.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For information about all settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Reference Manual*.

Initializing and Uninitializing a LogicLock Region

You must initialize the LogicLock data structures before creating or modifying any LogicLock regions and before executing any of the Tcl commands listed below.

Use the following Tcl command to initialize the LogicLock data structures:

```
initialize_logiclock
```

Use the following Tcl command to uninitialize the LogicLock data structures before closing your project:

```
uninitialize_logiclock
```

Creating or Modifying LogicLock Regions

Use the following Tcl command to create or modify a LogicLock region:

```
set_logiclock -auto_size true -floating true -region \ <my_region-name>
```

In the above example, the region's size is set to auto and the state is set to floating.

If you specify a region name that does not exist in the design, the command creates the region with the specified properties. If you specify the name of an existing region, the command changes all properties you specify and leaves unspecified properties unchanged.

For more information about creating LogicLock regions, refer to the sections "Creating LogicLock Regions" on page 12–7 and "Creating LogicLock Regions with the Chip Planner" on page 12–19.

Obtaining LogicLock Region Properties

Use the following Tcl command to obtain LogicLock region properties. This example returns the height of the region named `my_region`:

```
get_logiclock -region my_region -height
```

Assigning LogicLock Region Content

Use the following Tcl commands to assign or change nodes and entities in a LogicLock region. This example assigns all nodes with names matching `fifo*` to the region named `my_region`.

```
set_logiclock_contents -region my_region -to fifo*
```

You can also make path-based assignments with the following Tcl command:

```
set_logiclock_contents -region my_region -from fifo -to ram*
```

For more information about assigning LogicLock Region Content, refer to "Assigning LogicLock Region Content" on page 12–18.

Prevent Further Netlist Optimization

Use this Tcl code to prevent further netlist optimization for nodes in a back-annotated LogicLock region. In your code, specify the name of your LogicLock region.

```
foreach node [get_logiclock_contents -region \
<region name> -node_locations] {
    set node_name [lindex $node 0]
    set_instance_assignment -name ADV_NETLIST_OPT_ALLOWED \
"NEVER ALLOW" -to $node_name}
```

The `get_logiclock_contents` command is in the `logiclock` package.

Save a Node-Level Netlist for the Entire Design into a Persistent Source File

Make the following assignments to cause the Quartus II Fitter to save a node-level netlist for the entire design into a **.vqm** file:

```
set_global_assignment-name LOGICLOCK_INCREMENTAL_COMPILE_ASSIGNMENT ON
set_global_assignment-name LOGICLOCK_INCREMENTAL_COMPILE_FILE <file
name>
```

Any path specified in the file name is relative to the project directory. For example, specifying **atom_netlists/top.vqm** places **top.vqm** in the **atom_netlists** subdirectory of your project directory.

A **.vqm** file is saved in the directory specified at the completion of a full compilation.

For more information about saving a node-level netlist, refer to "Atom Netlist Design Information" on page 12–55.

This is not supported for designs targeting newer devices such as the Stratix IV, Stratix III, Cyclone III, Arria II GX, or Arria GX.

 © March 2009 Altera Corporation

Setting LogicLock Assignment Priority

Use the following Tcl code to set the priority for a LogicLock region's members. This example reverses the priorities of the LogicLock region in your design.

```
set reverse [list]
for each member [get_logiclock_member_priority] {
    set reverse [insert $reverse 0 $member]
{
set_logiclock_member_priority $reverse
```

For more information about setting the LogicLock assignment priority, refer to "LogicLock Restrictions" on page 12–53.

Assigning Virtual Pins

Use the following Tcl command to turn on the virtual pin setting for a pin called `my_pin`:

```
set_instance_assignment -name VIRTUAL_PIN ON -to my_pin
```

For more information about assigning virtual pins, refer to "Virtual Pins" on page 12–17.

Back-Annotating LogicLock Regions

The Quartus II software provides the back-annotate Tcl package that allows you to back-annotate the contents of a LogicLock region.

```
logiclock_back_annotate [-h | -help] [-long_help]
[-region <region name>] [-from <source name>]
[-to <destination name>] [-exclude_from] [-exclude_to] [-path_exclude
<path_exclude name>]
[-no_delay_chain] [-no_contents] [-lock] [-routing]
[-resource_filter <resource_filter value>] [-no_dont_touch]
[-remove_assignments] [-no_demote_lab] [-no_demote_mac]
[-no_demote_pin] [-no_demote_ram]
```

For example, the following command back-annotates all nodes and routing in the region `one_region`.

```
package require ::quartus::backannotate
logiclock_back_annotate -routing -lock -no_demote_lab -region \
one_region
```

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Conclusion

Design floorplan analysis is a valuable method for achieving timing closure and timing closure optimal performance in highly complex designs. With their analysis capability, the Quartus II Chip Planner and the Timing Closure Floorplan tools help you close timing quickly on your designs. Using these tools together with LogicLock and Incremental Compilation enables you to compile your designs hierarchically, preserving the timing results from individual compilation runs. You can use

LogicLock regions as part of an incremental compilation methodology to improve your productivity. You can also include a module in one or more projects while maintaining performance and reducing development costs and time-to-market. LogicLock region assignments give you complete control over logic and memory placement to improve the performance of non-hierarchical designs as well.

Referenced Documents

This chapter references the following documents:

- *AN 437: Power Optimization in Stratix III FPGAs*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Best Practices for Incremental Compilation Partition and Floorplan Assignments* chapters in volume 1 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Reference Manual*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 12–3 shows the revision history for this chapter.

Table 12–3. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Was chapter 10 in the 8.1.0 release.	Updated for the Quartus II 9.0 software release.

Table 12–3. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
November 2008, v8.1.0	■ Changed page size to 8½" × 11" ■ Removed "Importing LogicLock Regions", "Exporting LogicLock Regions", Importing Back-Annotated Routing in LogicLock Regions", LogicLock Regions Versus Soft LogicLock Regions", and "Exporting Back-Annotated Routing in LogicLock Regions", and removed subsections in "Using LogicLock Methodology for Older Device Families" ■ Updated "Viewing Routing Congestion" on page 12–29 ■ Updated Table 12–2	Updated for the Quartus II 8.1 software release.
May 2008 v8.0.0	■ Updated the following sections: →"Chip Planner Tasks and Layers" →"LogicLock Regions" →"Back-Annotating LogicLock Regions" →"LogicLock Regions in the Timing Closure Floorplan" ■ Added the following sections: →"Reserve LogicLock Region" →"Creating Non-Retangular LogicLock Regions" →"Viewing Available Clock Networks in the Device" ■ Updated Table 10–1 ■ Removed the following sections: →Reserve LogicLock Region Design Analysis Using the Timing Closure Floorplan	Updated for the Quartus II 8.0 software release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

13. Netlist Optimizations and Physical Synthesis

QII52007-9.0.0

Introduction

The Quartus® II software offers physical synthesis optimizations to optimize your design beyond the optimization performed in the normal course of the Quartus II compilation flow. The effect of these options depends on the structure of your design. Physical synthesis optimizations can help improve the performance of your design, regardless of the synthesis tool used.

Netlist optimization options work with your design's atom netlist, which describes a design in terms of Altera®-specific primitives. An atom netlist file can take the form of an Electronic Design Interchange Format (**.edf**) file or a Verilog Quartus Mapping (**.vqm**) file generated by a third-party synthesis tool, or a netlist used internally by the Quartus II software. Physical synthesis optimizations are applied at different stages of the Quartus II compilation flow, either during synthesis, fitting, or both.

This chapter explains how the physical synthesis optimizations in the Quartus II software can modify your design's netlist and help improve your quality of results. This chapter also provides information about preserving your compilation results through back-annotation and writing out a new netlist, and provides guidelines for applying the various options.

Because the node names for primitives in the design can change when you use the physical synthesis option, evaluate whether your design flow requires fixed node names. If you use a verification flow that might require fixed node names, such as the SignalTap® II Embedded Logic Analyzer, formal verification, or the LogicLock based optimization flow (for legacy devices), you must turn off the synthesis netlist optimization and physical synthesis options.

WYSIWYG Primitive Resynthesis

If you use a third-party tool to synthesize your design, use the **Perform WYSIWYG primitive resynthesis** option to apply optimizations to the synthesized netlist.

The **Perform WYSIWYG primitive resynthesis** option directs the Quartus II software to un-map the logic elements (LEs) in an atom netlist to logic gates, and then re-map the gates back to Altera-specific primitives. Third-party synthesis tools generate an atom netlist file that specifies Altera-specific primitives. Atom netlist files can take the form of either an **.edf** or **.vqm** file generated by the third-party synthesis tool. When you turn on this option, the Quartus II software can work on different techniques specific to the device architecture during the re-mapping process. This feature re-maps the design using the **Optimization Technique** specified for your project (**Speed**, **Area**, or **Balanced**).

This option has no effect if you are using Quartus II integrated synthesis to synthesize your design.

To turn on this option, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.

2. In the **Category** list, select **Analysis and Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Select **Perform WYSIWYG Primitive Resynthesis**. Turn on the setting and click **OK**.

If you want to perform WYSIWYG resynthesis on only a portion of your design, you can use the Assignment Editor to assign the **Perform WYSIWYG primitive resynthesis** logic option to a lower-level entity in your design. This option can be used with Arria® II GX, Arria GX, HardCopy® series, Stratix® series, Cyclone® series, MAX® II, or APEX™ series device families.

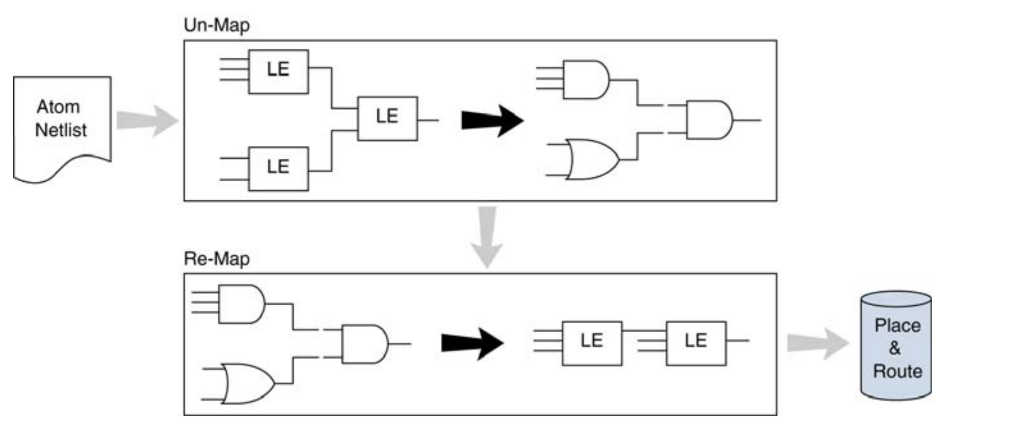
The results of the remapping are dependent on the **Optimization Technique** chosen for your project. To select the **Optimization Technique**, perform the following steps:

1. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
2. Under **Optimization Technique**, select from **Speed**, **Area**, or **Balanced** to specify how the Quartus II technology mapper optimizes the design. The **Balanced** setting is the default for many Altera device families; this setting optimizes the timing critical parts of the design for speed and the rest for area.
3. Click **OK**.

Refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook* for details on the Optimization Technique option.

Figure 13–1 shows the Quartus II software flow for this feature.

Figure 13–1. WYSIWYG Primitive Resynthesis



The **Perform WYSIWYG primitive resynthesis** option is not applicable if you are using Quartus II integrated synthesis to synthesize your design because the design does not contain Altera primitives.

The **Perform WYSIWYG primitive resynthesis** option unmaps and remaps only logic cells, also referred to as LCELL or LE primitives, and regular I/O primitives (which may contain registers). Double data rate (DDR) I/O primitives, memory primitives, digital signal processing (DSP) primitives, and logic cells in carry/cascade chains are not touched. Logic specified in an encrypted **.vqm** file or an **.edf** file, such as third-party intellectual property (IP), is not touched.

Turning on this option can cause drastic changes to the node names in the **.vqm** file or **.edf** file from your third-party synthesis tool, because the primitives in the atom netlist are broken apart and then remapped within the Quartus II software. Registers can be minimized away and duplicates removed, but registers that are not removed have the same name after remapping.

Any nodes or entities that have the **Netlist Optimizations** logic option set to **Never Allow** are not affected during WYSIWYG primitive resynthesis. To apply this logic option, on the Assignments menu, click **Assignment Editor**. This option disables WYSIWYG resynthesis for parts of your design.

Primitive node names are specified during synthesis. When netlist optimizations are applied, node names might change because primitives are created and removed. HDL attributes applied to preserve logic in third-party synthesis tools cannot be maintained because those attributes are not written into the atom netlist read by the Quartus II software.

If you use the Quartus II software to synthesize, you can use the **Preserve Register (preserve)** and **Keep Combinational Logic (keep)** attributes to maintain certain nodes in the design.

For more information about using these attributes during synthesis in the Quartus II software, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook.*

Performing Physical Synthesis Optimizations

Traditionally, the Quartus II design flow has involved separate steps of synthesis and fitting. The synthesis step optimizes the logical structure of a circuit for area, speed, or both. The Fitter then places and routes the logic cells to ensure critical portions of logic are close together and use the fastest possible routing resources. While you are using this push-button flow, the synthesis stage is unable to anticipate the routing delays seen in the Fitter. Because routing delays are a significant part of the typical critical path delay, performing synthesis operations with physical delay knowledge allows the tool to target its timing-driven optimizations at these parts of the design. This tight integration of the fitting and synthesis processes is known as physical synthesis.

The following sections describe the physical synthesis optimizations available in the Quartus II software, and how they can help improve your performance results. Physical synthesis optimization options can be used with Arria II GX, Arria GX, the Stratix and Cyclone series device families, and HardCopy ASIC series.

If you are migrating your design to a HardCopy II device, you can target physical synthesis optimizations to the FPGA architecture in the FPGA-first flow or to the HardCopy II architecture in the HardCopy-first flow. The optimizations are mapped to the other device architecture during the migration process.

You cannot target optimizations to optimize for both device architectures individually because doing so would result in a different post-fitting netlist for each device.

For more information about using physical synthesis with HardCopy devices, refer to the *Quartus II Support of HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook*.

You can choose the physical synthesis optimization options you want for your design during synthesis and fitting in the **Physical Synthesis Optimizations** page. The settings include optimizations for improving performance and fitting in the selected device.

You can also set the effort level for physical synthesis optimizations. Normally, physical synthesis optimizations increase the compilation time. Choose the **Fast** option if you want to reduce the impact on compilation time. When you choose this effort level, the Quartus II software performs limited register retiming operations during fitting. The **Extra** option runs additional algorithms to get the best possible results, with maximum impact on the compilation time.

To optimize the performance, the following options are available:

- **Perform physical synthesis for combinational logic**
- **Perform register retiming**
- **Perform automatic asynchronous signal pipelining**
- **Perform register duplication**

To optimize for better fitting, you can choose from the following options:

- **Perform physical synthesis for combinational logic**
- **Perform logic to memory mapping**

To view and modify the physical synthesis optimization options, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the "+" icon to expand **Compilation Process Settings** and select **Physical Synthesis Optimizations**. The **Physical Synthesis Optimizations** page appears.
3. Specify the options for performing physical synthesis optimizations.

Some physical synthesis options affect only registered logic and some options affect only combinational logic. Select options based on whether you want to keep the registers intact or not. For example, if your verification flow involves formal verification, you might have to keep the registers intact.

You can control the effect of physical synthesis with the **Effort Level** option. The default selection is **Normal**. The **Extra** effort setting uses extra compilation time to try to achieve extra circuit performance, while the **Fast** effort setting uses less compilation time than **Normal** but might not achieve the same gains.

All Physical Synthesis optimizations write results to the **Netlist Optimizations** report. To access this report, on the Processing menu, click **Compilation Report**. In the **Compilation Report** list, click the "+" icon to expand **Fitter** and select **Netlist Optimizations**. This report provides a list of atom netlist files that were modified, created, and deleted during physical synthesis. Similarly, physical synthesis optimizations performed during synthesis write results to the synthesis report. To access this report, on the Processing menu, click **Compilation Report**. In the **Compilation Report** list, select **Analysis & Synthesis**.

Nodes or entities that have the **Netlist Optimizations** logic option set to **Never Allow** are not affected by the physical synthesis algorithms. To access this logic option, on the Assignments menu, click **Assignment Editor**. Use this option to disable physical synthesis optimizations for parts of your design.

Automatic Asynchronous Signal Pipelining

The **Perform automatic asynchronous signal pipelining** option on the **Physical Synthesis Optimizations** page in the **Compilation Process Settings** section of the **Settings** dialog box allows the Quartus II Fitter to perform automatic insertion of pipeline stages for asynchronous clear and asynchronous load signals during fitting when these signals negatively affect performance. You can use this option if asynchronous control signal recovery and removal times are not achieving their requirements.

This option improves performance for designs in which asynchronous signals in very fast clock domains cannot be distributed across the chip fast enough due to long global network delays. This optimization performs automatic pipelining of these signals, while attempting to minimize the total number of registers inserted.

The **Perform automatic asynchronous signal pipelining** option adds registers to nets driving the asynchronous clear or asynchronous load ports of registers. This adds register delays (adds latency) to the reset, adding the same number of register delays for each destination using the reset. Thus, changing the behavior of the signal in the design. Use this only if adding latency to reset signals does not violate any design requirements. This option also prevents the promotion of signals to global routing resources.

The Quartus II software performs automatic asynchronous signal pipelining only if **Enable Recovery/Removal analysis** is enabled. If you use the TimeQuest Timing Analyzer, **Enable Recovery/Removal analysis** is turned on by default. Pipelining is allowed only on asynchronous signals that have the following properties:

- The asynchronous signal is synchronized to a clock (a synchronization register drives the signal)
- The asynchronous signal fans-out only to asynchronous control ports of registers

To use **Enable Recovery/Removal analysis** with the Classic Timing Analyzer, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the “+” icon to expand **Timing Analysis Setting** and select **Classic Timing Analyzer Settings**. The **Classic Timing Analyzer Settings** page appears.
3. Click **More Settings**. The **More Timing Settings** dialog box appears.
4. In the **Name** list, select **Enable Recovery/Removal analysis**. In the **Setting** list, select **On**.
5. Click **OK**.
6. Click **OK**.

The Quartus II software does not perform automatic asynchronous signal pipelining on asynchronous signals that have the **Netlist Optimization** logic option set to **Never Allow**.

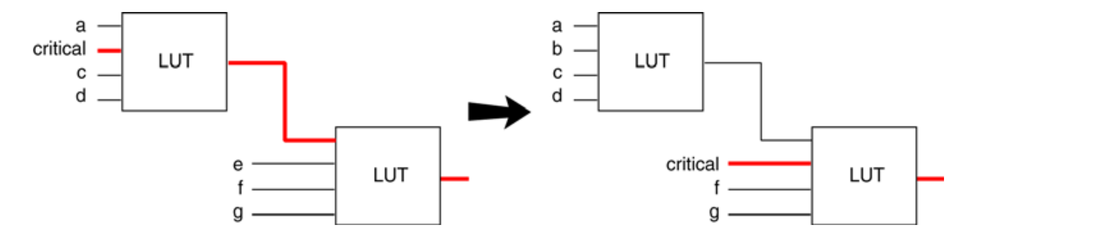
Physical Synthesis for Combinational Logic

To optimize the design and reduce delay along the critical paths, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the "+" icon to expand **Compilation Process Settings** and select **Physical Synthesis Optimizations**. The **Physical Synthesis Optimizations** page appears.
3. Turn on **Perform physical synthesis for combinational logic** by checking the appropriate box.

The software can accomplish this type of optimization by swapping the look-up table (LUT) ports within LEs so that the critical path has fewer layers through which to travel. See Figure 13–2 for an example. This option also allows the duplication of LUTs to enable further optimizations on the critical path.

Figure 13–2. Physical Synthesis for Combinational Logic



In the first case, the critical input feeds through the first LUT to the second LUT. The Quartus II software swaps the critical input to the first LUT with an input feeding the second LUT. This reduces the number of LUTs contained in the critical path. The synthesis information for each LUT is altered to maintain design functionality.

The **Physical synthesis for combinational logic** option affects only combinational logic in the form of LUTs. These transformations might occur during the synthesis stage or the Fitter stage during compilation. The registers contained in the affected logic cells are not modified. Inputs into memory blocks, DSP blocks, and I/O elements (IOEs) are not swapped.

The Quartus II software does not perform combinational optimization on logic cells that have the following properties:

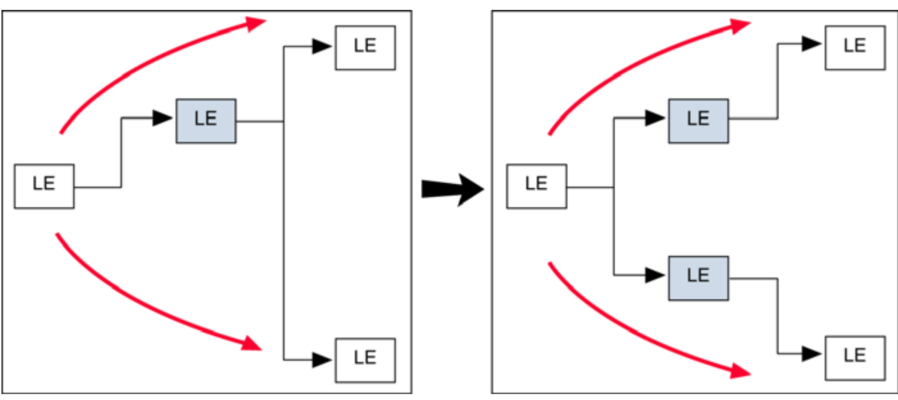
- Are part of a chain
- Drive global signals
- Are constrained to a single logic array block (LAB) location
- Have the **Netlist Optimizations** option set to **Never Allow**

If you consider logic cells with any of these conditions for physical synthesis, you can override these rules by setting the **Netlist Optimizations** logic option to **Always Allow** on a given set of nodes.

Physical Synthesis for Registers—Register Duplication

The **Perform register duplication** option on the **Physical Synthesis Optimizations** page in the **Compilation Process Settings** section of the **Settings** dialog box allows the Quartus II Fitter to duplicate registers based on Fitter placement information. You can also duplicate combinational logic when this option is enabled. A logic cell that fans out to multiple locations can be duplicated to reduce the delay of one path without degrading the delay of another. The new logic cell can be placed closer to critical logic without affecting the other fan-out paths of the original logic cell. Figure 13–3 shows an example of register duplication.

Figure 13–3. Register Duplication



The Quartus II software does not perform register duplication on logic cells that have the following properties:

- Are part of a chain
- Contain registers that drive asynchronous control signals on another register
- Contain registers that drive the clock of another register
- Contain registers that drive global signals
- Contain registers that are constrained to a single LAB location
- Contain registers that are driven by input pins without a t_{SU} constraint
- Contain registers that are driven by a register in another clock domain
- Are considered virtual I/O pins
- Have the **Netlist Optimizations** option set to **Never Allow**

For more information about virtual I/O pins, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook.*

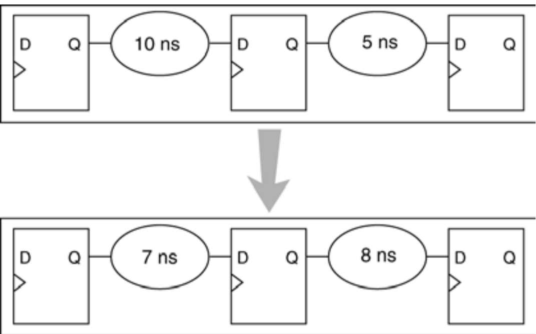
If you want to consider logic cells that meet any of these conditions for physical synthesis, you can override these rules by setting the **Netlist Optimizations** logic option to **Always Allow** on a given set of nodes.

Physical Synthesis for Registers—Register Retiming

The Register Retiming option enables the movement of registers across combinational logic, allowing the Quartus II software to trade off the delay between timing critical paths and non-critical paths. Register Retiming can be done during Quartus II integrated synthesis or during the Fitter stages of design compilation.

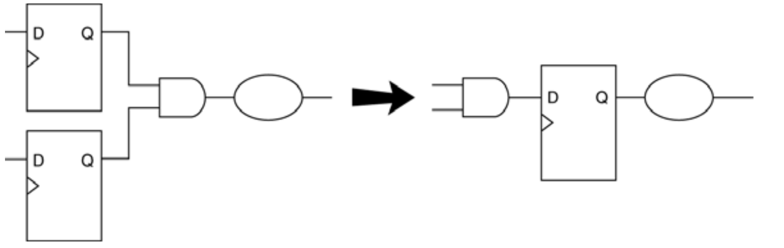
Figure 13–4 shows an example of register retiming in which the 10-ns critical delay is reduced by moving the register relative to the combinational logic.

Figure 13–4. Register Retiming Diagram



Retiming can create multiple registers at the input of a combinational block from a register at the output of a combinational block. In this case, the new registers have the same clock and clock enable. The asynchronous control signals and power-up level are derived from previous registers to provide equivalent functionality. Retiming can also combine multiple registers at the input of a combinational block to a single register (Figure 13–5).

Figure 13–5. Combining Registers with Register Retiming



To move registers across combinational logic to balance timing, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the “+” icon to expand **Compilation Process Settings** and select **Physical Synthesis Optimizations**. The **Physical Synthesis Optimizations** page appears.
3. Specify your preferred option under **Physical synthesis for performance** and **Effort level**.
4. Click **OK**.

If you want to prevent register movement during register retiming, you can set the **Netlist Optimizations** logic option to **Never Allow**. This option can be applied either to individual registers or entities in the design using the Assignment Editor.

In digital circuits, synchronization registers are instantiated on cross clock domain paths to reduce the possibility of metastability. The Quartus II software detects such synchronization registers and they are not moved, even if register retiming is turned on.

The following sets of registers are not moved during register retiming:

- Both registers in a direct connection from input pin-to-register-to-register if both registers have the same clock and the first register does not fan-out to anywhere else. These registers are considered synchronization registers.
- Both registers in a direct connection from register-to-register if both registers have the same clock, the first register does not fan-out to anywhere else, and the first register is fed by another register in a different clock domain (directly or through combinational logic). These registers are considered synchronization registers.

By default, the Quartus II software assumes that a synchronization register chain consists of a set of two registers. If your design has synchronization register chains that are longer than two, you must indicate the number of registers in your synchronization chains, so that those are not affected by when register retiming is enabled. To do this, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Setting** page appears.
3. Click **More Settings**. The **More Analysis & Synthesis Settings** dialog box appears.
4. In the **Name** list, select **Synchronization Register Chain Length** and modify the setting to match the synchronization register length used in your design. If you set a value of 1 for the **Synchronization Register Chain Length**, it means that any registers connected to the first register in a register-to-register connection can be moved during retiming. A value of $n > 1$ means that any registers in a sequence of length 1, 2,… *n* are not moved during register retiming.

In general, the Quartus II software does not perform register retiming on logic cells that have the following properties:

- Are part of a cascade chain
- Contain registers that drive asynchronous control signals on another register
- Contain registers that drive the clock of another register
- Contain registers that drive a register in another clock domain
- Contain registers that are driven by a register in another clock domain
- Contain registers that are constrained to a single LAB location
- Contain registers that are connected to SERDES
- Are considered virtual I/O pins
- Registers that have the **Netlist Optimizations** logic option set to **Never Allow**

For more information about virtual I/O pins, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

If you want to consider logic cells that meet any of these conditions for physical synthesis, you can override these rules by setting the **Netlist Optimizations** logic option to **Always Allow** on a given set of registers.

Preserving Your Physical Synthesis Results

The Quartus II software generates the same results on every compilation for the same source code and settings on a given system. Therefore, it is typically not necessary to take steps to preserve your results from compilation to compilation. When changes are made to the source code or to the settings, you usually get the best results by allowing the software to compile without using previous compilation results or location assignments. In some cases, if you avoid performing analysis and synthesis or `quartus_map`, and run the Fitter or another desired Quartus II executable instead, you can skip the synthesis stage of the compilation.

When you use the Quartus II incremental compilation flow, you can preserve synthesis results for a particular partition of your design by choosing a netlist type of post-synthesis. If you want to preserve fitting results between compilation runs, choose a netlist type of post-fit during incremental compilation. The rest of this section is relevant only for those designs using older devices that do not support incremental compilation.

For information about the incremental compilation design methodology, refer to the *Quartus II Incremental Compilation for Hierarchical and Team Based Design* chapter in volume 1 of the *Quartus II Handbook*.

You can preserve the resulting nodes from physical synthesis in older devices that do not support incremental compilation. Preserving the nodes might be required if you use the LogicLock flow to back-annotate placement and/or import one design into another. Use this only for older devices such as the APEX or ACEX family. If you use older devices (such as the APEX or ACEX series) with LogicLock as an optimization strategy, you might have to use this option to preserve the nodes resulting from netlist optimizations. For newer devices, use incremental compilation to preserve results. When using the incremental compilation flow, you do not have to use this option even if you have LogicLock regions in the design.

To preserve the nodes from Quartus II physical synthesis optimization options for older devices that do not support incremental compilation, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Compilation Process Settings**. The **Compilation Process Settings** page appears.
3. Turn on **Save a node-level netlist of the entire design into a persistent source file**. This setting is not available for Cyclone III, Stratix III, and newer devices.
4. Click **OK**.

The **Save a node-level netlist of the entire design into a persistent source file** option saves your final results as an atom-based netlist in **.vqm** file format. By default, the Quartus II software places the **.vqm** file in the **atom_netlists** directory under the current project directory. To create a different **.vqm** file using different Quartus II settings, in the **Compilation Process Settings** page, change the **File name** setting.

If you use synthesis netlist optimizations (and not physical synthesis optimizations), generating a **.vqm** file is optional. To lock down the location of all logic and device resources in the design with or without a Quartus II-generated **.vqm** file, on the Assignments menu, click **Back-Annotate Assignments** and specify the desired options. You should use back-annotated location assignments unless the design has been finalized. Making any changes to the design invalidates your back-annotated location assignments. If you require changes later, use the new source HDL code as your input files, and remove the back-annotated assignments corresponding to the old code or netlist.

If you create a **.vqm** file to recompile the design, use the new **.vqm** file as the input source file and turn off the synthesis netlist optimizations for the new compilation.

If you use the physical synthesis optimizations and want to lock down the location of all LEs and other device resources in the design with the **Back-Annotate Assignments** command, a **.vqm** file netlist is required. The **.vqm** file preserves the changes that were made to your original netlist. Because the physical synthesis optimizations depend on the placement of the nodes in the design, back-annotating the placement changes the results from physical synthesis. Changing the results means that node names are different, and your back-annotated locations are no longer valid.

You should not use a Quartus II-generated **.vqm** file or back-annotated location assignments with physical synthesis optimizations unless the design has been finalized. Making any changes to the design invalidates your physical synthesis results and back-annotated location assignments. If you require changes later, use the new source HDL code as your input files, and remove the back-annotated assignments corresponding to the Quartus II-generated **.vqm** file.

To back-annotate logic locations for a design that was compiled with physical synthesis optimizations, first create a **.vqm** file. When recompiling the design with the hard logic location assignments, use the new **.vqm** file as the input source file and turn off the physical synthesis optimizations for the new compilation.

If you are importing a **.vqm** file and back-annotated locations into another project that has any **Netlist Optimizations** turned on, it is important to apply the **Never Allow** constraint to make sure node names don’t change; otherwise, the back-annotated location or LogicLock assignments are invalid.

For newer devices, such as the Stratix, Cyclone, or Arria GX series, use incremental compilation to preserve compilation results instead of using logic back-annotation.

Physical Synthesis Options for Fitting

The Quartus II software enables you to use the physical synthesis option to improve the fitting results. To access these settings, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.

2. In the **Category** list, click the "+" icon to expand **Compilation Process Settings** and select **Physical Synthesis Optimizations**. The **Physical Synthesis Optimizations** page appears.
3. Under **Optimize for fitting** (physical synthesis for density), there are two physical synthesis options available to improve fitting your design in the target device: **Physical synthesis for combinational logic** and **Perform logic to memory mapping** (Table 13–1).

Table 13–1. Physical Synthesis Optimizations Options

Option	Function
Physical Synthesis for Combinational Logic	When you select this option, the Fitter detects duplicate combinational logic and optimizes combinational logic to improve the fit.
Perform Logic to Memory Mapping	When you select this option, the Fitter can remap registers and combinational logic in your design into unused memory blocks and achieves a fit.

Applying Netlist Optimization Options

Netlist optimizations options can have various effects on different designs. Designs that are well coded or have already been restructured to balance critical path delays might not see a noticeable difference in performance.

To obtain optimal results when using netlist optimization options, you might have to vary the options applied to find the best results. By default, all options are off. Turning on additional options leads to the largest effect on the node names in the design. Take this into consideration if you are using a LogicLock-based optimization strategy (for older devices such as APEX and ACEX), or verification flow, such as the SignalTap II Embedded Logic Analyzer or formal verification that requires fixed or known node names. Applying all of the physical synthesis options at the **Extra** effort level generally produces the best results for those options, but adds significantly to the compilation time. You can also use the **Physical synthesis effort level** options to decrease the compilation time.

Typically, the WYSIWYG primitive resynthesis does not add much compilation time relative to the overall design compilation time.

To find the best results, you can use the Quartus II Design Space Explorer (DSE) to apply various sets of netlist optimization options.

For more information about using DSE, refer to the *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Scripting Reference Manual* includes the same information in PDF form. For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

You can specify many of the options described in this section either on an instance, or at a global level, or both.

Use the following Tcl command to make a global assignment:

```
set_global_assignment -name <QSF variable name> <value> ↵
```

Use the following Tcl command to make an instance assignment:

```
set_instance_assignment -name <QSF variable name> <value> \
-to <instance name> ↵
```

Synthesis Netlist Optimizations

Table 13–2 lists the Quartus II Settings File (**.qsf**) variable names and applicable values for the settings discussed in "WYSIWYG Primitive Resynthesis" on page 13–1. The **.qsf** file variable name is used in the Tcl assignment to make the setting along with the appropriate value. The **Type** column indicates whether the setting is supported as a global setting, an instance setting, or both.

Table 13–2. Synthesis Netlist Optimizations and Associated Settings

Setting Name	Quartus II Settings File Variable Name	Values	Type
Perform WYSIWYG Primitive Resynthesis	ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP	ON, OFF	Global, Instance
Optimization Technique	*<Device Family Name>*_OPTIMIZATION_TECHNIQUE	AREA, SPEED, BALANCED	Global, Instance
Power-Up Don't Care	ALLOW_POWER_UP_DONT_CARE	ON, OFF	Global
Save a node-level netlist into a persistent source file	LOGICLOCK_INCREMENTAL_COMPILE_ASSIGNMENT	ON, OFF	Global
	LOGICLOCK_INCREMENTAL_COMPILE_FILE	*<filename>*	
Allow Netlist Optimizations	ADV_NETLIST_OPT_ALLOWED	"ALWAYS ALLOW", DEFAULT, "NEVER ALLOW"	Instance

Physical Synthesis Optimizations

Table 13–3 lists the **.qsf** file variable name and applicable values for the settings discussed in "Performing Physical Synthesis Optimizations" on page 13–3. The **.qsf** file variable name is used in the Tcl assignment to make the setting, along with the appropriate value. The **Type** column indicates whether the setting is supported as a global setting, an instance setting, or both.

Table 13–3. Physical Synthesis Optimizations and Associated Settings

Setting Name	Quartus II Settings File Variable Name	Values	Type
Physical Synthesis for Combinational Logic	PHYSICAL_SYNTHESIS_COMBO_LOGIC	ON, OFF	Global
Automatic Asynchronous Signal Pipelining	PHYSICAL_SYNTHESIS_ASYNCHRONOUS_SIGNAL_PIPELINING	ON, OFF	Global
Perform Register Duplication	PHYSICAL_SYNTHESIS_REGISTER_DUPLICATION	ON, OFF	Global
Perform Register Retiming	PHYSICAL_SYNTHESIS_REGISTER_RETIMING	ON, OFF	Global
Power-Up Don't Care	ALLOW_POWER_UP_DONT_CARE	ON, OFF	Global, Instance
Power-Up Level	POWER_UP_LEVEL	HIGH,LOW	Instance
Allow Netlist Optimizations	ADV_NETLIST_OPT_ALLOWED	"ALWAYS ALLOW", DEFAULT, "NEVER ALLOW"	Instance
Save a node-level netlist into a persistent source file	LOGICLOCK_INCREMENTAL_COMPILE_ASSIGNMENT	ON, OFF	Global
	LOGICLOCK_INCREMENTAL_COMPILE_FILE	*<filename>*	

Incremental Compilation

For information about scripting and command line usage for incremental compilation as mentioned in "Preserving Your Physical Synthesis Results" on page 13–10, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Back-Annotating Assignments

You can use the `logiclock_back_annotate` Tcl command to back-annotate resources in your design. This command can back-annotate resources in LogicLock regions, and resources in designs without LogicLock regions.

For more information about back-annotating assignments, refer to "Preserving Your Physical Synthesis Results" on page 13–10.

The following Tcl command back-annotates all registers in your design:

```
logiclock_back_annotate -resource_filter "REGISTER"
```

The `logiclock_back_annotate` command is in the `backannotate` package.

Conclusion

Physical synthesis optimizations restructure and optimize your design netlist. Taking advantage of these Quartus II netlist optimizations can help improve your quality of results.

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Quartus II Support for HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 13–4 shows the revision history for this chapter.

Table 13–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Was chapter 11 in the 8.1.0 release. ■ Updated the “Physical Synthesis for Registers—Register Retiming” and“Physical Synthesis Options for Fitting” ■ Updated “Performing Physical Synthesis Optimizations” ■ Deleted Gate-Level Register Retiming section. ■ Updated the referenced documents	Updated GUI references and procedure steps, and document structure for the Quartus II software 9.0 release.
November 2008 v8.1.0	Changed to 8½” × 11” page size. No change to content.	Updated for the Quartus II 8.1 software release.
May 2008 v8.0.0	Updated “Physical Synthesis Optimizations for Performance on page 11-9 Added Physical Synthesis Options for Fitting on page 11-16	Updated for Quartus II 8.0 version.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

14. Design Space Explorer

QII52008-9.0.0

Introduction

The Quartus® II software includes many advanced optimization algorithms to help you achieve timing closure, optimize area, and reduce dynamic power. The various settings and parameters control the behavior of the algorithms. These options provide complete control over the Quartus II software optimization and power techniques.

Each FPGA design is unique. There is no standard set of options that always results in the best performance or power utilization. Each design requires a unique set of options to achieve optimal performance. This chapter describes Design Space Explorer (DSE), a utility written in Tcl/Tk that automates finding the best set of options for your design. DSE explores the design space of your design by applying various optimization techniques and analyzing the results.

DSE is a valuable tool to use in the late phases of your design cycle. You can take advantage of DSE's capability to automatically sweep multiple options to close timing, minimize area, or reduce power consumption on a design that is nearing completion.

DSE Concepts

This section explains the concepts and terminology used by DSE.

Exploration Space and Exploration Point

Before DSE explores a design, DSE creates an exploration space, which consists of Synthesis and Fitter settings available in the Quartus II software. Each group of settings in an exploration space is referred to as a point. An exploration space contains one or more points. DSE traverses the points in the exploration space to determine optimal settings for your design.

Seed and Seed Sweeping

The Quartus II Fitter uses a seed to specify the starting value that randomly determines the initial placement for the current design. The seed value can be any non-negative integer value. Changing the starting value may or may not produce better fitting. However, varying the value of the seed or seed sweeping allows the Quartus II software to determine an optimal value for the current design.

DSE extends Fitter seed sweeping in exploration spaces by providing a method for sweeping through general compilation and Fitter parameters to find the best options for your design. You can run DSE in various exploration space modes, ranging from an exhaustive try-all-options-and-values mode to a mode that focuses on one parameter.

DSE Exploration

DSE compares all exploration point results with the results of a base compilation, generated from the initial settings that you specify in the original Quartus II project files. As DSE traverses all points in the exploration space, all settings not explicitly modified by DSE default to the base compilation setting. For example, if an exploration point turns on register retiming but does not modify the register packing setting, the register packing setting defaults to the value you specified in the base compilation.

DSE performs the base compilation with the settings you specified in the original Quartus II project. These settings are restored after DSE traverses all points in the exploration space. DSE makes a copy of your base revision and uses this copy for changing the settings required to traverse through all other points in the chosen exploration space. Your base revision is not affected by DSE exploration.

General Description

You can use DSE in either the graphical user interface (GUI) or from a command line. To run DSE with the GUI, either click **Launch Design Space Explorer** on the Tools menu in the Quartus II software, or type the following at the command prompt:

```
quartus_sh --dse ↵
```

To run DSE from a command line, type the following command at the command prompt:

```
quartus_sh --dse -nogui [<options>] ↵
```

You can run DSE with the following options:

```
-archive
-concurrent-compiles [0..6]
-custom-file <filename>
-decision-column <"column name">
-exploration-space <"space">
-ignore-failed-base
-llr-restructuring
-lower-priority
-lsf-queue <queue name>
-nogui
-optimization-goal <"goal"
-report-all-resource-usage
-project <project name>
-revision <revision name>
-run-power
-search-method <"method">
-seeds <seed list>
-skip-base
-slaves <"slave1, slave2, slave3......">
-stop-after-time <dd:hh:mm>
-stop-after-zero-failing-paths
-use-lsf
```

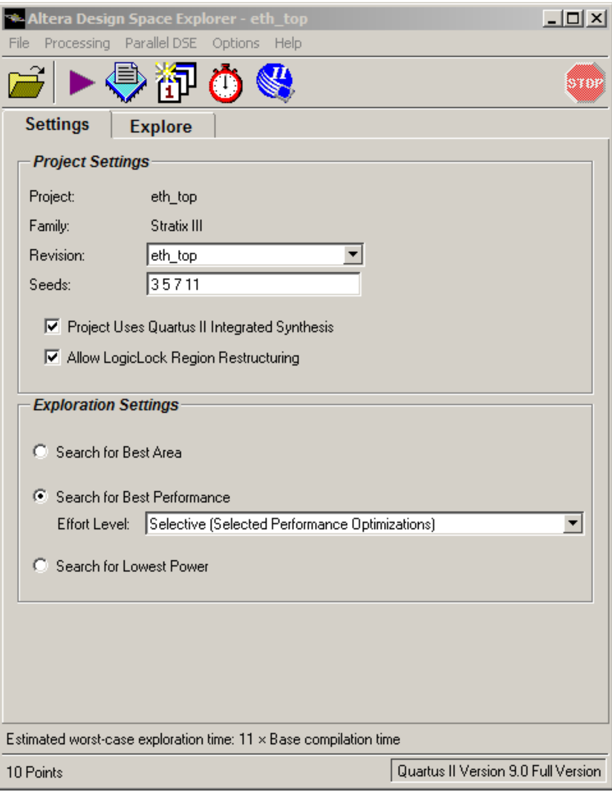
DSE script **dse.tcl** is located in <*Quartus II installation directory*>**/common/tcl/apps/dse** on Windows and Linux operating systems.

To launch DSE from the Quartus II software user interface, on the Tools menu, click **Launch Design Space Explorer**.

For more information about DSE, launch the DSE GUI. On the **Help menu**, click **Contents** or press the F1 key.

Figure 14–1 shows DSE user interface. The **Settings** tab is divided into two sections: **Project Settings** and **Exploration Settings**.

Figure 14–1. DSE User Interface



Timing Analyzer Support

DSE supports both the Quartus II Classic Timing Analyzer and the Quartus II TimeQuest Timing Analyzer. You must set the timing analyzer prior to opening the project in DSE. After the timing analyzer is set, DSE performs the design exploration with the selected timing analyzer.

You can directly launch the TimeQuest Timing Analyzer from DSE if you have set the default timing analyzer to TimeQuest and have specified the timing constraints in an **.sdc** file.

DSE Flow

You can run DSE at any point in the design process. However, Altera recommends that you run DSE late in your design cycle when your focus is on optimizing performance and power. The results gained from different combinations of optimization options early in the design cycle may not persist over large changes in a design.

DSE runs the Quartus II software for every point in the exploration space. The Quartus II software always attempts to achieve all your timing requirements regardless of whether or not you are running DSE. The **Exploration Settings** you choose in DSE will determine the settings to be used for compilation. DSE does not change the behavior of the Quartus II software.

DSE provides a summary of results for all the compilations and flags the best compilation run based on exploration setting you have chosen. Specifying all timing requirements before you use DSE to explore your design is very important to ensure that DSE finds the optimal set of parameters for your design based on design criteria you set in your initial design.

You can change the initial placement configuration used by the Quartus II Fitter by varying the **Fitter See**d value. You can enter seed values in the **Seeds** field of DSE user interface.

To set the seed value in the Quartus II software, on the Assignments menu, click **Settings** and select **Fitter Settings**.

Compilation time increases as DSE exploration spaces become more comprehensive. Increased compilation time results from running several compilations and comparing the generated results with the original base compilation results.

For a typical design, varying only the seed value varies the f_{MAX} within a range of +/-5%. For example, when compiling with three different seeds, one-third of the time f_{MAX} does not improve over the initial compilation, one-third of the time f_{MAX} improves by 5%, and one-third of the time f_{MAX} improves by 5%.

DSE Support for Altera Device Families

DSE support varies across Altera device families. The Stratix® series of devices, the Cyclone® series of devices, and the Arria® GX series of devices can take advantage of all the available DSE optimization methods. The MAX® II series of devices and older families, such as the APEX™ series of devices and the FLEX® series of devices, support a subset of DSE options.

DSE Project Settings

This section provides the following information about DSE project settings:

- Setting up DSE work environment
- Specifying the revision
- Setting the initial seed
- Quartus II integrated synthesis
- Restructuring LogicLock regions

Setting Up the DSE Work Environment

From the DSE GUI, you can open a Quartus II project for a design exploration by clicking **Open Project** on the File menu and then browsing to your project. Clicking on the Quartus II icon in the DSE GUI closes the DSE GUI and opens the project.

Specifying the Revision

You can specify the revision to be explored with the **Revision** field in the DSE GUI. The **Revision** field is populated after the Quartus II project has been opened.

If no revisions were created in the Quartus II project, the default revision, which is the top-level entity, is used. For more information, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

Setting the Initial Seed

To specify the seed that DSE uses for an exploration, specify a non-negative integer value in the **Seeds** box under **Project Settings** on the **Settings** tab. The seed value determines your design's initial placement in a Quartus II compilation.

To specify a range of seeds, type the low end of the range followed by a hyphen, followed by the high end of the range. For example, 2-5 specifies that DSE uses the values 2, 3, 4, and 5 as seeds.

Project Uses Quartus II Integrated Synthesis

Make sure to check the **Project Uses Quartus II Integrated Synthesis** option if you use Quartus II integrated synthesis to synthesize your design. The DSE explores several options that affect, and can help, the synthesis stage of compilation when this option is enabled.

For more information about integrated synthesis options, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Restructuring LogicLock Regions

The **Allow LogicLock Region Restructuring** option allows DSE to modify LogicLock region properties in your design, if any exist. DSE applies the Soft property to LogicLock regions to improve timing. In addition, DSE can remove LogicLock regions that negatively affect the performance of the design.

DSE makes a copy of your base revision and modifies the LogicLock region settings on the new copy to see if the timing improves with the **Allow Logiclock Region Restructuring** option. Your original revision remains intact.

Exploration Settings

Use the **Exploration Settings** list to select the type of exploration to perform: **Search for Best Area**, **Search for Best Performance**, or **Search for Lowest Power**.

The **Search for Best Area** option uses a predefined exploration space that targets device utilization improvements for your design.

The **Search for Best Performance** option uses a predefined exploration space that targets performance improvements for your design. Depending on the device that your design targets, you can select up to five predefined exploration spaces: **Low** (**Seed Sweep**), **Medium** (**Extra Effort Space**), **High (Physical Synthesis Space)**, **Highest** (**Physical Synthesis with Retiming Space**), and **Selective** (**Selected Performance Optimizations**). As you move from **Low** to **Highest**, the number of options explored by DSE increases, which causes compilation time to increase.

The **Search for Lowest Power** option uses a predefined exploration space that targets overall power improvements for your design. When **Search for Lowest Power** is selected, DSE automatically runs the PowerPlay Power Analyzer for each point in the space. You must ensure that the PowerPlay Power Analyzer is configured correctly to ensure accurate results. DSE issues a warning if the confidence level for any power estimate is low.

The **Search for Best Area** option uses a predefined exploration space that targets device utilization improvements for your design.

Using DSE to Search for the Best Performance

This section describes using DSE to search for the best performance in your design.

Effort Level

When you select **Search for Best Performance** under the **Exploration Settings** in the DSE GUI, you can select the effort level you wish to use to compile your design in DSE. The effort levels are **Low** (**Seed Sweep**), **Medium** (**Extra Effort Space**), **High** (**Physical Synthesis Space**), **Highest** (**Physical Synthesis with Retiming Space**) and **Selective** (**Selected Performance Optimizations**). DSE traverses the points in the exploration space, applies the settings to the design, and compares compilation results to determine the best settings for your design based on your chosen effort level. Search time increases proportionally with the breadth of the options being explored. The exploration space search time increases with the number, type, and combination of options DSE explores.

DSE offers the following exploration space types:

- “Seed Sweep”
- “Extra Effort Space”
- “Physical Synthesis Space”
- “Physical Synthesis with Retiming Space”
- “Selective Performance Optimization”

Seed Sweep

Enter the seed values in the **Seeds** field in the DSE user interface. There are no “magic” seeds. The variation between seeds is truly random, any non-negative integer value is as likely to produce good results. DSE defaults to seeds 3, 5, 7, and 11. The **Low (Seed Sweep)** option exploration space does not change your netlist.

The **Seeds** field accepts individual seed values, for example, 2, 3, 4, and 5, or seed ranges, for example, 2-5.

Each seed value you specify requires an additional compilation. For example, if you enter five seeds, the compilation time increases to 5 times the initial (or base) compilation time.

Extra Effort Space

The **Extra Effort Space** effort level adds the **Register Packing** option to the exploration space done by the **Seed Sweep**. The **Extra Effort Space** effort level also increases the Quartus II Fitter effort during placement and routing. However, the **Extra Effort Space** effort level does not change your netlist.

Physical Synthesis Space

The **Physical Synthesis Space** effort level adds physical synthesis options such as register retiming and physical synthesis for combinational logic to the options included in the **Extra Effort Space** effort level. These netlist optimizations move registers in your design. Look-up tables (LUTs) are modified by these options. However, the design behavior is not affected by these options.

For more information about physical synthesis, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

Physical Synthesis with Retiming Space

The **Physical Synthesis with Retiming Space** effort level includes all the options in the **Physical Synthesis Space** effort level, and it explores various Quartus II Integrated Synthesis optimization options and register retiming. Register retiming can move registers in your design.

The **Physical Synthesis with Retiming Space** effort level works only for designs that have been synthesized using Quartus II Integrated Synthesis.

Selective Performance Optimization

The **Selective Performance Optimization** effort level combines a seed sweep with various performance Fitter settings to improve the timing of your design. The seed sweep is performed over a limited number of points in such a way that the base settings are not replicated. This is the recommended option for large designs where other spaces may be too large. Use this exploration space for first-time DSE searches on your designs to evaluate the range of results.

Table 14–1 shows the settings adjusted by each effort level.

Table 14–1. Summaries of Effort Levels (Part 1 of 2) *(Note 1)*

Optimization Options	Effort Levels			
	Seed Sweep	Extra Effort	Physical Synthesis	Retiming
Analysis and Synthesis Settings				
Optimization technique	—	—	✓	✓
Perform WYSIWYG resynthesis	—	—	✓	✓
Fitter Settings				
Fitter seed	✓	✓	✓	✓

Table 14–1. Summaries of Effort Levels (Part 2 of 2) *(Note 1)*

Optimization Options	Effort Levels			
	Seed Sweep	Extra Effort	Physical Synthesis	Retiming
Register packing	—	✓	✓	✓
Increase PowerFit Fitter effort	—	✓	✓	✓
Perform physical synthesis for combinational logic	—	—	✓	✓
Perform register retiming	—	—	—	✓

Note to Table 14–1:

(1) For effort levels that include Quartus II Integrated Synthesis Projects, DSE increases the synthesis effort.

DSE Flow Options

You can control the configuration of DSE with the following options:

- Create a Revision from the DSE Point
- Stop Flow When Zero Failing Paths are Achieved
- Continue Exploration Even If Base Compilation Fails
- Run Quartus II PowerPlay Power Analyzer During Exploration
- Archive All Compilations
- Stop Flow After Time
- Skip Base Analysis and Compilation If Possible

Create a Revision from the DSE Point

After you have performed a design exploration with DSE, a Quartus II revision can be made from any exploration point. This option facilitates the creation of multiple revisions based on the same space point for further optimization within the Quartus II software.

Stop Flow When Zero Failing Paths are Achieved

Instructs DSE to stop exploring the space after it encounters any point, including the base point, that has zero failing paths. DSE uses the failing path count reported in the **All Failing Paths report** column to make this decision.

Continue Exploration Even If Base Compilation Fails

With the **Continue Exploration Even If Base Compilation Fails** option turned on, DSE continues the exploration even when a design compilation error occurs. For example, if timing settings are not applied to your design, a DSE error occurs. To cause DSE to continue with the exploration instead of halting when an error occurs, turn on this option.

Run Quartus II PowerPlay Power Analyzer During Exploration

Turn on **Run Quartus II PowerPlay Power Analyzer During Exploration** to run the Quartus II PowerPlay Analyzer for every exploration performed by DSE. Using this option can help you debug your design and determine trade-offs between power requirements and performance optimization.

Archive All Compilations

Turn on **Archive all Compilations** to create a Quartus II Archive File (**.qar**) for each compilation. These archive files are saved to the **dse** directory in the design's working directory.

The result of each DSE run is saved as a **.qar** file in the **dse** subdirectory under your project directory. Each run is identified by a number. The best result of DSE run is saved with the name **best.qar**.

The **dse** directory also contains a spreadsheet (**results.csv**) that compares the results of all the individual runs in your DSE compilation.

Stop Flow After Time

Turn on Stop Flow After Time to stop further exploration after a specified number of days, hours, and/or minutes.

Exploration time might exceed the specified value because DSE does not stop in the middle of a compilation.

Skip Base Analysis and Compilation If Possible

The **Skip Base Analysis & Compilation if Possible** option allows DSE to skip the Analysis and Elaboration stage or the compilation of the base point if base point compilation results are available from a previous Quartus II compilation.

DSE Configuration File

Many options exist that allow you to customize the behavior of each DSE exploration. For example, you can specify seed values or a list of slave computers to be used for a Parallel DSE run. Each time you close the DSE GUI, it saves these values in a configuration file, **dse.conf**. The next time you launch the DSE GUI, it reads the values from **dse.conf** and restores the previous exploration settings.

Where the **dse.conf** file is stored varies depending on the operating system that launches DSE. Table 14–2 specifies the locations where **dse.conf** files are stored.

Table 14–2. DSE Configuration File Location

OS	File Location (default)	Comment
Windows	**%APPDATA%/Altera/dse.conf**	If the variable %APPDATA% is not defined, the configuration file is saved to **%HOME%\.altera.quartus\dse.conf**
Linux	**~/.altera.quartus/dse.conf**	

Settings specified in the DSE command-line mode are not saved to a **dse.conf** configuration file.

Parallel DSE Information

This section covers the **Parallel DSE** option, which enables you to run an exploration on multiple computers concurrently. This feature increases the processing efficiency of design space exploration. You can access the settings for **Parallel DSE** from the Parallel DSE menu in the DSE GUI.

Computer Load Sharing Using Parallel DSE

DSE uses cluster computing technology to decrease exploration time when you click **Distribute Compilations to Other Machines** (on the Parallel DSE menu). DSE uses multiple client computers to compile points in the specified exploration space.

Parallel DSE functions in one of the following modes:

- **Use LSF Resources—**DSE uses the Platform LSF grid computing technology to distribute exploration space points to a computing network.
- **Use QSlave—**This function uses a Quartus II master process. DSE acts as a master and distributes exploration space points to client computers.

When you use the **Distribute Compilations to Other Machines** option, different exploration points in the exploration space are compiled on different slave client computers at the same time. Concurrent compilations requires a separate license for each instance of the Quartus II software being used to compile the design. Each compilation also might require licenses for any IP cores in the design. Therefore, the number of parallel distributed compilations can be limited to the number of licenses available for the Quartus II software or the IP core used in your design.

Parallel DSE Using LSF Resources

The easiest way to use distributed DSE technology is to submit the compilations to a preconfigured LSF cluster at your local site. For more information about LSF software, refer to www.platform.com, or contact your system administrator. To run Parallel DSE using LSF resources, on the Parallel DSE menu, click **Configure Resources**.

Parallel DSE Using a Quartus II Master Process

Before DSE can use computers in the local area network to compile points in the exploration space, you must create Quartus II software slave instances on the computers that will be used as clients. Type the following command at a command prompt on a client computer:

```
quartus_sh --qslave ↵
```

Repeating this command on several computers creates a cluster of Quartus II software slaves for DSE to use. After you have created a set of Quartus II software slaves on the network, add the names of each slave computer in the **QSlave** tab of the **Configure Resources** dialog box.

To access the **Configure Resources** dialog box, on the Parallel DSE menu, click **Configure Resources**. To add resources, click the **QSlave** tab and click **Add** and type the client name. Click **OK**.

At the start of an exploration, DSE assumes the role of a Quartus II software master process and submits points to the slaves on the list to compile. If the list is empty, DSE issues an error and the search stops.

For more information about running and configuring Quartus II slaves, type the following command at the command prompt:

```
quartus_sh --help=qslave ↵
```

Parallel DSE uses a protocol based on FTP to move files between the master and the slaves. By default, the qslave client listens to port number 1977 for communication with the master. If you are running a firewall on a computer that is running the qslave client, make sure you configure the firewall software such that it allows incoming and outgoing transmission control protocol (TCP) and user datagram protocol (UDP) packets on the port used by qslave.

You must set this configuration in every computer that is used as a slave in a distributed DSE environment.

You can change the default port number used by qslave by typing the following command at a command prompt:

```
quartus_sh --qslave port=<new_port_number> ↵
```

You must use the same version of the Quartus II software to run the slave processes as you use to run DSE. To determine which Quartus II software version you are using to run DSE, select Help and click **About DSE.** Unexpected results can occur if you mix different Quartus II software versions when using the Parallel DSE feature.

When you are using ClearCase revision control software, Parallel DSE compilations launched within a ClearCase view might fail. ClearCase catches system I/O calls that can prevent communication between the DSE master and its slave computers. To avoid this problem, run Parallel DSE outside of the ClearCase environment.

Concurrent Local Compilations

To reduce compilation time, DSE can compile exploration points concurrently. The **Concurrent Local Compilations** option allows you to specify up to six concurrent compilations by choosing an integer value ranging from 0 through 6. You can use this option in conjunction with Parallel DSE. However, your system must have the appropriate resources and licenses to perform concurrent compilations, and distributed processing. Multiprocessor or multicore systems are recommended for concurrent local compilations.

Concurrent Local Compilations require a separate Quartus II software license for each concurrent compilation. For example, if you compile four concurrent compilations, you must have four licenses. Ensure that sufficient licenses are available before you choose a **Concurrent Local Compilations** value and start compilation.

You can use Concurrent compilations and the distributed compiles with other computer options at the same time if you use the QSlave approach for distributing compiles to other computers.

If you use LSF, all the jobs are submitted to the LSF system.

Referenced Documents

This chapter references the following documents:

- *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*
- *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*

Document Revision History

Table 14–3 shows the revision history for this chapter.

Table 14–3. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Was chapter 12 in the 8.1.0 release. ■ Updated Table 14–1 and Table 14–2 . ■ Added the following sections: → "Project Uses Quartus II Integrated Synthesis" on page 14–5 → "Exploration Settings" on page 14–5 → "Effort Level" on page 14–6 ■ Updated the following sections: → "General Description" on page 14–2 → "Setting Up the DSE Work Environment" on page 14–5 → "Seed Sweep" on page 14–6 → "Physical Synthesis Space" on page 14–7 → "Concurrent Local Compilations" on page 14–11 ■ Deleted the following sections: → Ignore SignalTap and SignalProbe Settings → Quartus II Integrated Synthesis → Search fir Best Performance, Search for Best Area Options, or Search for Lowest Power Option	Updated for the Quartus II software version 9.0 release.

Table 14–3. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
November 2008 v8.1.0	Changed to 8½” x 11” page size. No change to content.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	Updated the following sections: ■ “Search for Best Performance, Search for Best Area Options, or Search for Lowest Power Option” on page 12–5 ■ “Using the DSE to Search for the Best Performance” on page 12–6 ■ “Physical Synthesis with Retiming Space” on page 12–7 ■ “Parallel DSE Information” on page 12–10 Deleted the following sections: ■ Advanced Search Options ■ Exploration Space ■ Custom Space ■ Area Optimization Space ■ Change Decision Column ■ Save Exploration Space to File ■ Creating Custom Spaces for DSE	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

Section IV. Engineering Change Management

Programmable logic can accommodate changes to a system specification late in the design cycle. Last-minute design changes, commonly referred to as engineering change orders (ECOs), are small changes to the functionality of a design after the design has been fully compiled. This section describes how the Chip Planner feature in the Quartus® II software supports ECOs by allowing quick and efficient changes to your logic late in the design cycle.

This section includes the following chapter:

- Chapter 15, Engineering Change Management with the Chip Planner

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

15. Engineering Change Management with the Chip Planner

QII52017-9.0.0

Introduction

Programmable logic can accommodate changes to a system specification late in the design cycle. In a typical engineering project development cycle, the specification for the programmable logic portion is likely to change after engineering development begins or while integrating all system elements.

Last-minute design changes, commonly referred to as engineering change orders (ECOs), are small changes to the functionality of a design after the design has been fully compiled. A design is fully compiled when synthesis and place-and-route are completed.

The Chip Planner supports ECOs by allowing quick and efficient changes to your logic late in the design cycle. It provides a visual display of your post place-and-route design mapped to the device architecture of your chosen FPGA, from LAB placement in the device to each mapped Logic Element (LE) or Adaptive Logic Module (ALM). You can analyze your design with the visual display to alter how device resources are mapped to support ECOs.

This chapter addresses the impact that ECOs have on the design cycle, discusses the design flow for performing ECOs, and describes how you can use the Chip Planner to perform ECOs.

In addition to performing ECOs, the Chip Planner allows you to perform detailed analysis on routing congestion, relative resource usage, logic placement, LogicLock™ and customized regions, fan-ins and fan-outs, paths between registers, and delay estimates for paths.

For detailed information about using the Chip Planner for design analysis, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

The Chip Planner does not support the following device families:

- MAX® 7000
- ACEX®
- APEX™ series
- FLEX® series

For floorplan analysis using these devices, use the Timing Closure Floorplan. ECOs are not supported for these device families.

Engineering Change Orders

ECOs are typically performed during the verification stage of a design cycle. When a small change is required on a design, such as modifying a PLL for a different clock frequency or routing a signal out to a pin for analysis, recompilation of the entire design can be time consuming, especially for larger designs. Because several iterations of small design changes can occur during the verification cycle, recompilation times can quickly add up. Furthermore, a full recompilation due to a small design change can result in the loss of previous design optimizations. Performing ECOs, instead of performing a full recompilation on your design, limits the change only to the affected portions of logic.

This section discusses the areas in which ECOs have an impact on a system design and how the Quartus® II software can help you optimize the design in these areas. The following topics are discussed in this section:

- "Performance"
- "Compilation Time" on page 15–3
- "Verification" on page 15–3
- "Documentation" on page 15–3

Performance

Making a small change to the design functionality can result in a loss of previous design optimizations. Typical examples of design optimizations are floorplan optimizations and physical synthesis. Ideally, you should preserve previous design optimizations.

The Chip Planner allows you to perform ECOs directly on the post place-and-route database of your design. Any changes you make are restricted to the affected device resources, so the timing performance of the remaining portions of your design are not affected. The Chip Planner performs design rule checks on all changes to prevent illegal modifications to your design.

Additionally, the Quartus II software offers an incremental compilation feature that preserves the optimizations and placement of your design during recompilation. This feature allows you to create partitions of your design, so that if a change is required after the design is fully placed and optimized, only the affected partition is recompiled to implement the change.

The incremental compilation flow fully supports performing ECOs with the Chip Planner.

When recompiling a project with the Quartus II incremental compilation enabled, the compiler preserves all ECOs performed with the Chip Planner in partitions that have not been modified.

For more information about how to use the incremental compilation feature in the Quartus II software, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

For more information about using the ECO flow in conjunction with incremental compilation, refer to "Using Incremental Compilation in the ECO Flow" on page 15–32.

Compilation Time

In the traditional programmable logic design flow, a small change in the design requires a complete recompilation of the design. A complete recompilation of the design consists of synthesis and place-and-route. Making small changes to the design to reach the final implementation on a board can be a long process. Since the Chip Planner works only on the post place-and-route database, you can implement your design changes in minutes without performing a full compilation.

Verification

After you make a design change, you can verify the impact to your design. To verify that you have not violated timing, you can perform a static timing analysis using the Quartus II Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer after you check and save your netlist changes within the Chip Planner.

For more information about the Quartus II TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*. For more information about the Quartus II Classic Timing Analyzer, refer to the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Additionally, you can perform a gate-level or timing simulation of the ECO-modified design by using the post place-and-route netlist generated by the Quartus II software.

Documentation

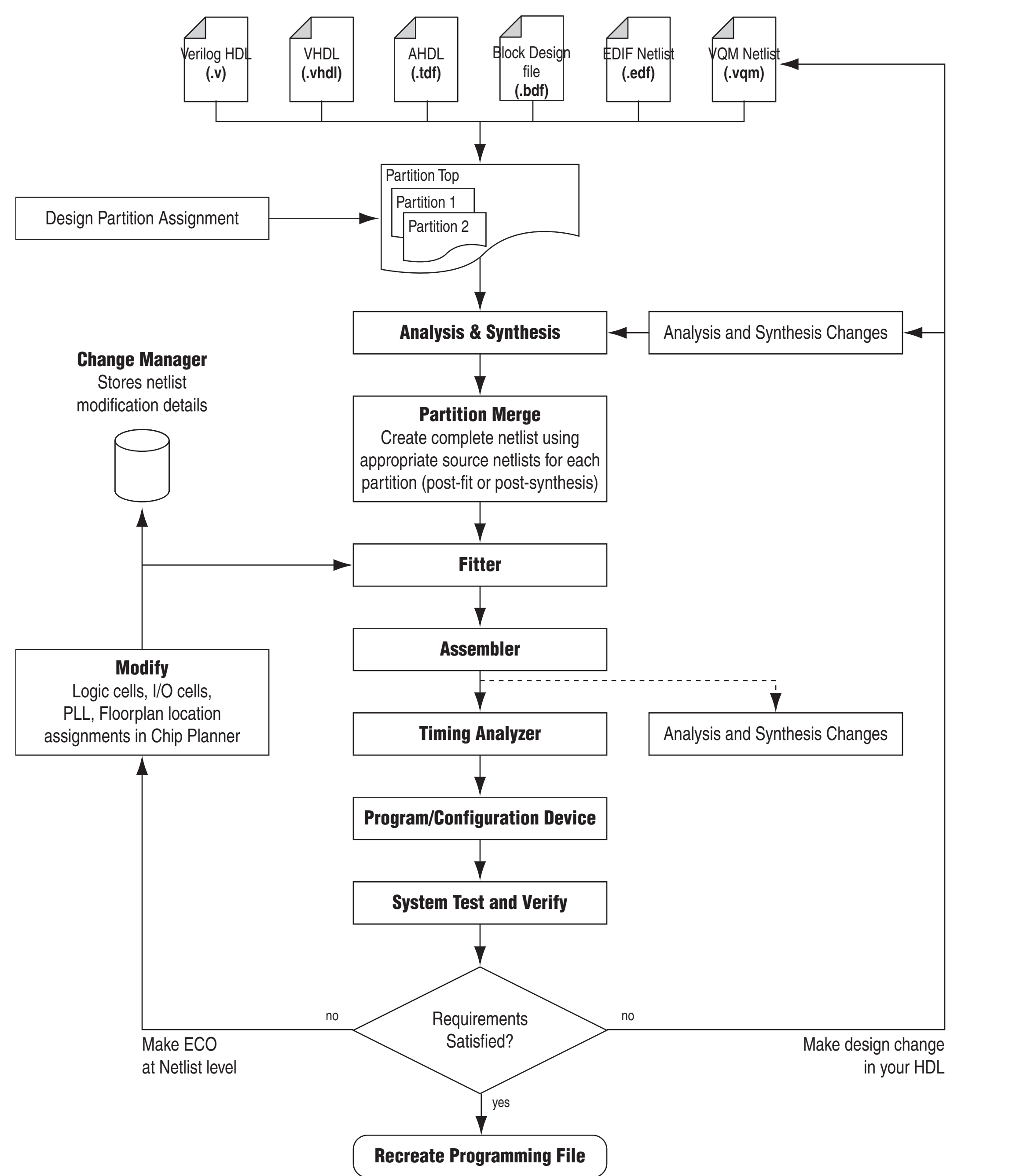
All ECOs made with the Chip Planner are logged in the Change Manager to provide a track record of all changes. By using the Change Manager, you can easily revert back to the original post-fit netlist or you can pick and choose which ECOs you want to have applied.

Additionally, the Quartus II software provides support for multiple compilation revisions of the same project. You can use ECOs made with the Chip Planner in conjunction with revision support to compare several different ECO changes and to provide the ability to revert back to previous project revisions.

ECO Design Flow

Figure 15–1 shows the design flow for performing ECOs.

Figure 15–1. Design Flow to Support ECO Changes



For iterative verification cycles, implementing small design changes at the netlist level can be faster than making an RTL code change. As such, making ECO changes are especially helpful when you debug the design on silicon and require a fast turnaround to generate a programming file for debugging the system.

A typical ECO application occurs when you uncover a problem on the board and isolate the problem to the appropriate nodes or I/O cells on the device. You must be able to correct the functionality quickly and generate a new programming file. Performing small changes using the Chip Planner allows you to modify the post place-and-route netlist directly. This bypasses having to perform synthesis and logic mapping, thus decreasing the turnaround time for programming file generation during the verification cycle. If the change corrects the problem, no modification of the HDL source code is necessary. You can use the Chip Planner to perform the following ECO-related changes to your design:

- Document the changes made with the Change Manager
- Easily recreate the steps taken to produce design changes
- Generate EDA simulation netlists for design verification
- Perform static timing analysis on the design

The Quartus II software can help reduce recompilation time with incremental recompilation for more complex changes that require HDL source code modifications.

The Chip Planner Overview

The Chip Planner provides a visual display of device resources. It shows the arrangement and usage of the resource atoms in the device architecture that you are targeting. Resource atoms are the building blocks for your device, such as ALMs, LEs, PLLs, DSP blocks, memory block, or I/O elements (IOEs).

The Chip Planner also provides an integrated platform for design analysis and for making ECOs to your design after place-and-route. The toolset consists of the Chip Planner (providing a device floorplan view of your mapped design) and two integrated subtools—the Resource Property Editor and the Change Manager.

For analysis, the Chip Planner can show logic placement, LogicLock and custom regions, relative resource usage, detailed routing information, routing congestion, fan-ins and fan-outs, paths between registers, and delay estimates for paths. Additionally, the Chip Planner allows you to create location constraints or resource assignment changes, such as moving or deleting logic cells or I/O atoms using the device floorplan. For ECO changes, the Chip Planner enables you to create, move, or delete logic cells in the post place-and-route netlist for fast programming file generation. Additionally, you can open the Resource Property Editor from the Chip Planner to edit the properties of resource atoms or to edit the connections between them. All changes to resource atoms and connections are logged automatically with the Change Manager.

Opening the Chip Planner

To open the Chip Planner, on the Tools menu, click **Chip Planner**. Alternatively, click the **Chip Planner** icon on the Quartus II software toolbar.

Optionally, the Quartus II software supports cross-probing to open the Chip Planner. To open the Chip Planner by cross-probing, use the shortcut menu in the following tools:

- Compilation Report

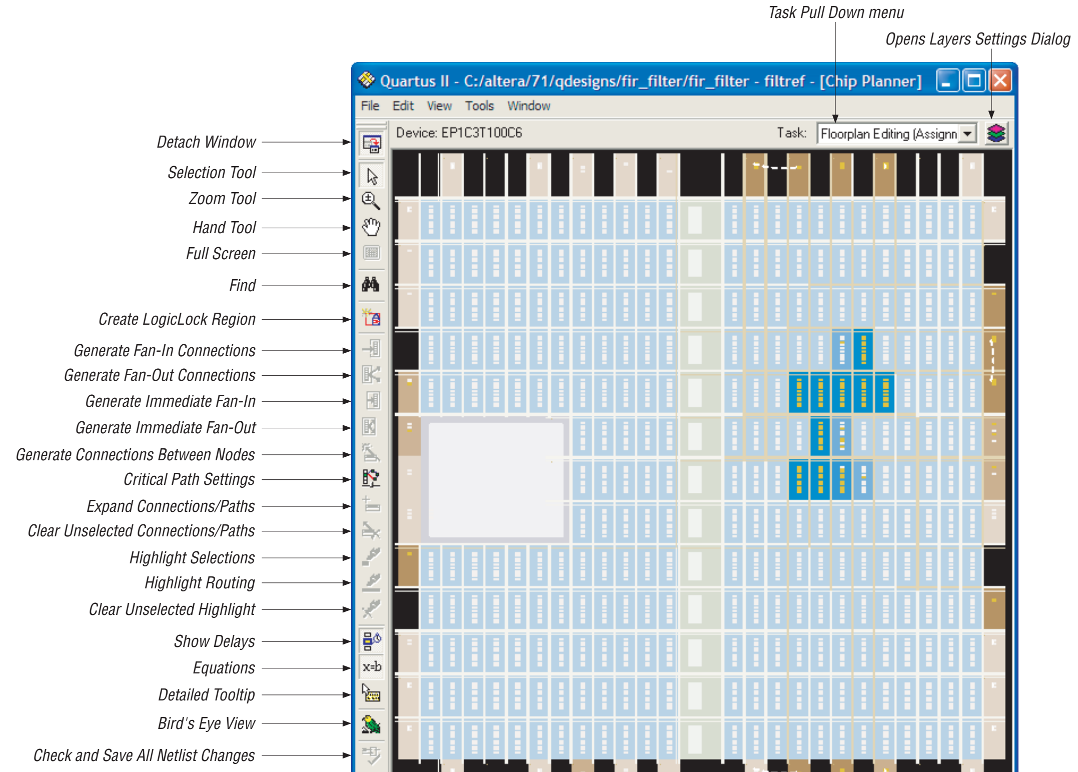
- Project Navigator window
- RTL source code
- Timing Closure Floorplan
- Node Finder
- Simulation Report
- RTL Viewer

For more information about the Timing Closure Floorplan, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

The Chip Planner Toolbar

The Chip Planner gives you design analysis capabilities with a user-friendly GUI. You can perform many functions within the Chip Planner from the menu items or by clicking the icons on the toolbar. Figure 15–2 shows an example of the Chip Planner toolbar and describes the commonly used icons.

Figure 15–2. Chip Planner Toolbar



You can also customize the icons on the Chip Planner toolbar. To customize the icons on the toolbar if the Chip Planner window is attached, on the Tools menu, click **Customize Chip Planner**. If the Chip Planner window is detached, on the Tools menu, click **Customize**.

For more information about using the Chip Planner for analyzing your design, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

The Chip Planner Tasks and Layers

The Chip Planner enables you to set up tasks to quickly implement ECO changes or manipulate assignments for the floorplan of the device. Each task consists of an editing mode and a set of customized layer settings.

The editing modes available in the Chip Planner are the Assignment mode and the ECO mode. Assignment mode enables you to create or manipulate LogicLock regions and make location constraints on the resource atoms used in your design. Assignments made are reflected in the Quartus II Settings File (**.qsf)** and the Assignment Editor. With ECO mode, you can create atoms, delete atoms, and move existing atoms to different locations. The changes made with ECO mode are made in the post place-and-route database. You can analyze your design with both modes.

The layers settings enable you to specify the displayed graphic elements for a given task. You can turn off the display of specific graphic elements to increase the window refresh speed and reduce visual clutter when viewing complex designs. The Background Color Map indicates the relative level of resource usage for different areas of the device. For example, **Routing Utilization** indicates the relative routing utilization and **Physical Timing Estimate** indicates the relative physical timing.

The Chip Planner has predefined tasks that enable you to quickly implement ECO changes or manipulate assignments for the floorplan of the device. The Chip Planner provides the following predefined tasks:

- Post-Compilation Editing (ECO)
- Floorplan Editing (Assignment)
- Partition Display (Assignment)
- Global Clock Network (Assignment)
- Power Analysis (Assignment)—available for Stratix® III, Stratix II, Stratix II GX, Cyclone® III, Cyclone II, and HardCopy® II devices only

You can choose the predefined task by selecting it in the **Task** list located in the upper right corner of the Chip Planner floorplan view.

To customize your own task, click on the Layers icon that is next to the **Task** list to open the **Layers Settings** dialog box.

For more information about assignments and analysis with the Chip Planner, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

For more information about performing ECOs using the ECO mode, refer to "Performing ECOs with the Chip Planner (Floorplan View)" on page 15–12.

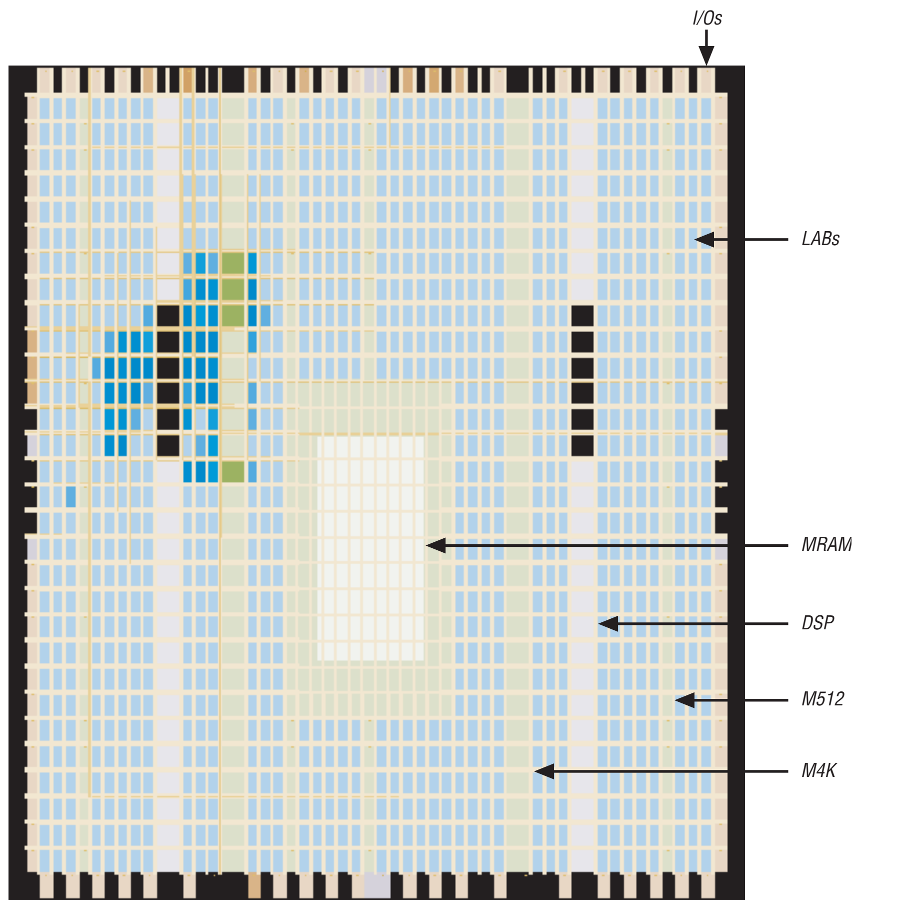
The Chip Planner Floorplan Views

The Chip Planner uses a hierarchical zoom viewer that shows various abstraction levels of the targeted Altera device. As you increase the zoom level, the level of abstraction decreases, thus revealing more detail about your design.

First-Level View

The first zoom level provides a high-level view of the entire device floorplan. This view provides a level of detail similar to the Field View in the Quartus II Timing Closure Floorplan. You can locate and view the placement of any node in your design. Figure 15–3 shows the Chip Planner Floorplan first-level view of a Stratix device.

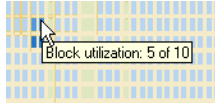
Figure 15–3. The Chip Planner First-Level (Highest) Floorplan View (Stratix Family Device)



Each resource is shown in a different color to help you distinguish between resources. The Chip Planner Floorplan uses a gradient color scheme in which the color becomes darker as the utilization of a resource increases. For example, as more LEs are used in the LAB, the color of the LAB becomes darker.

When you place the mouse pointer over a resource at this level, a tooltip appears that describes, at a high level, the utilization of the resource (Figure 15–4).

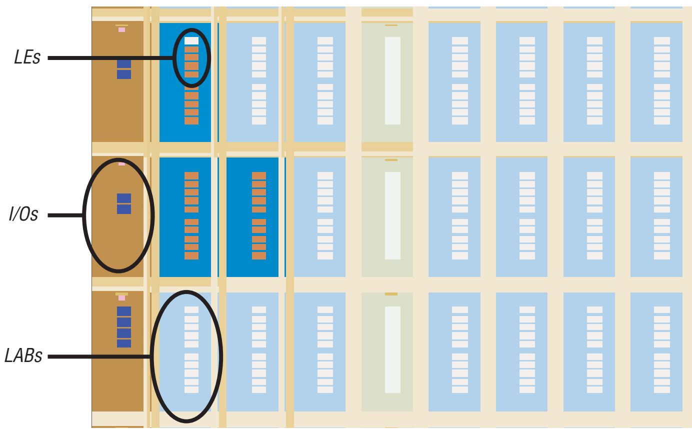
Figure 15–4. Tooltip Message: First-Level View



Second-Level View

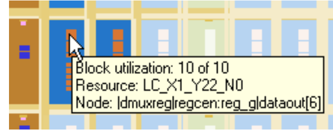
As you zoom in, you see an increase in the level of detail. Figure 15–5 shows the second-level view of the Chip Planner Floorplan for a Stratix device.

Figure 15–5. The Chip Planner Second-Level Floorplan View (Stratix Family Device)



At this level you can see the contents of LABs and I/O banks. You also see the routing channels that are used to connect resources. When you place the mouse pointer over an LE or ALM at this level, a tooltip is displayed (Figure 15–6) that shows the name of the LE/ALM, the location of the LE/ALM, and the number of resources that are used with that LAB. When you place the mouse pointer over an interconnect, the tooltip shows the routing channels that are used by that interconnect.

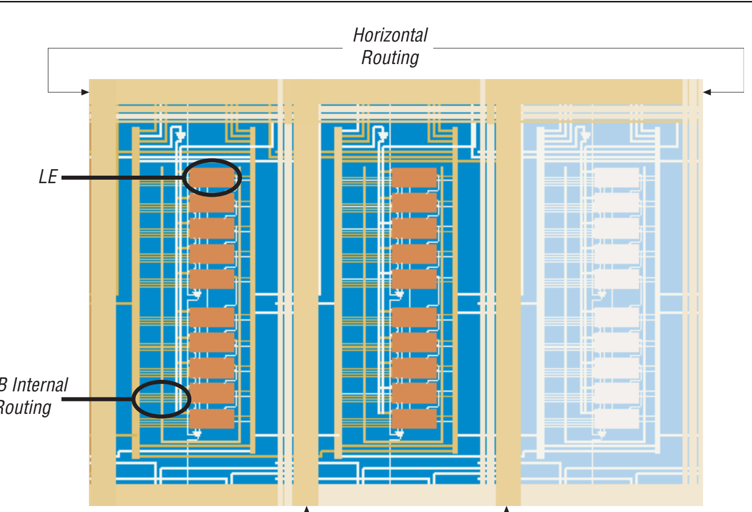
Figure 15–6. Tooltip Message: Second-Level View



Third-Level View

At the third level, which provides the greatest level of detail, you can see each routing resource that is used within a LAB in the FPGA. Figure 15–7 shows the level of detail at the third-level view for a Stratix device.

Figure 15–7. The Chip Planner Third-Level View



The second and third levels of zoom allow you to move LEs, ALMs, and I/Os from one physical location to another. You can move a resource by selecting, dragging, and dropping it into the desired location. At these levels, you can also create new LEs and I/Os.

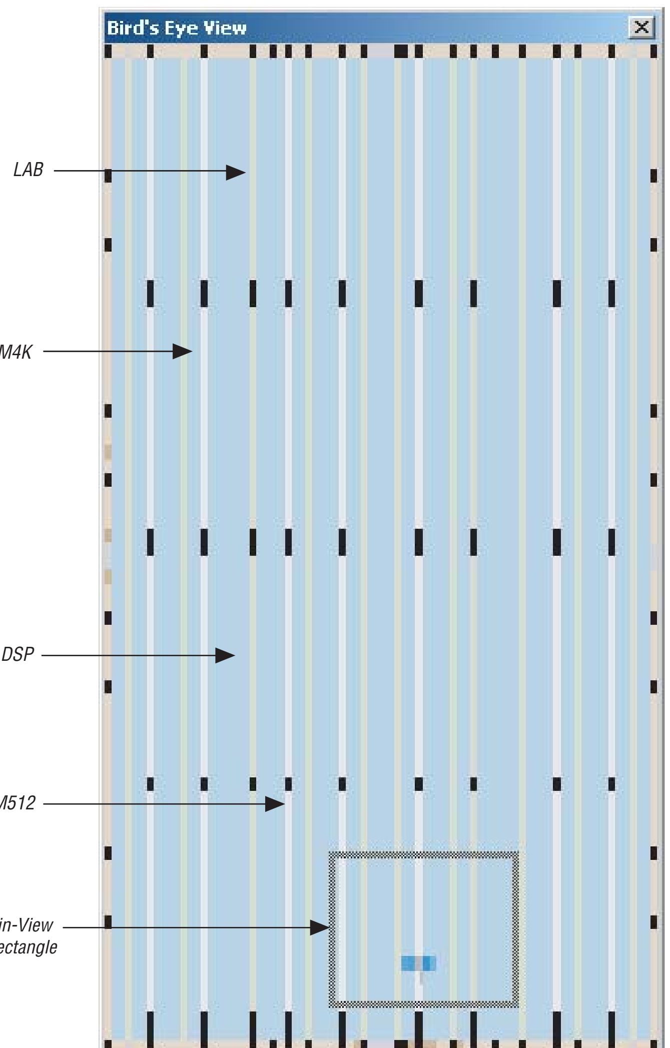
For more information about creating atoms, deleting atoms, or reallocating device atoms, refer to "Performing ECOs with the Chip Planner (Floorplan View)" on page 15–12.

For more information about creating Floorplan Assignments, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Bird's Eye View

The Bird's Eye View (Figure 15–8) displays a high-level picture of resource usage for the entire chip and provides a fast and efficient way to navigate between areas of interest in the Chip Planner.

Figure 15–8. Bird's Eye View



The Bird's Eye View is displayed as a separate window that is linked to the Chip Planner Floorplan. When you select an area of interest in the Bird's Eye View, the Chip Planner Floorplan automatically refreshes to show that region of the device. As you change the size of the main-view rectangle in the Bird's Eye View window, the main Chip Planner Floorplan window also zooms in (or zooms out). You can make the main-view rectangle smaller in the Bird's Eye View to see more detail on the Chip Planner Floorplan window.

The Bird's Eye View is particularly useful when the parts of your design that you are interested in are at opposite ends of the chip and you want to quickly navigate between resource elements without losing your frame of reference.

Performing ECOs with the Chip Planner (Floorplan View)

You can manipulate resource atoms in the Chip Planner when you select the **ECO** editing mode. The following ECO changes can be performed with the Chip Planner Floorplan view:

- "Creating Atoms"
- "Deleting Atoms" on page 15–16
- "Moving Atoms" on page 15–16

To configure the properties of atoms, such as managing the connections between different LEs/ALMs, use the Resource Property Editor.

Refer to "Resource Property Editor" on page 15–16 for details about editing atom resource properties.

To select the **ECO** editing mode in the Chip Planner, perform the following steps with the Chip Planner open:

1. On the View menu, click **Layers Settings**, or click on the Layers icon next to the **Task** list. The **Layers Settings** dialog box appears.
2. Under **Editing Mode**, select **ECO**.

Creating Atoms

While in the **ECO** editing mode, the Chip Planner enables you to easily create atoms by moving the mouse pointer over the desired resource atom, right-clicking, and clicking **Create Atom**. After the atoms are created, the properties can be edited by double-clicking the resource atom, which opens the Resource Property Editor.

The type of atoms that you can create are:

- ALMs (for the appropriate device families)
- LEs (for the appropriate device families)
- IOEs

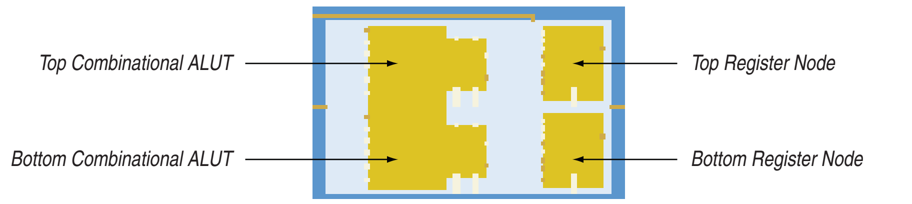
Creating resource atoms is not supported in the **Assignment** editing mode.

For the architectural details and resource atoms supported by your device, refer to the device data sheet.

Creating ALM Atoms

Each ALM has two combinational LUT outputs and two registered outputs. In the Chip Planner, you can divide each ALM into four resource atoms according to the type of output path. Figure 15–9 shows an ALM as shown in the Chip Planner.

Figure 15–9. ALM in the Chip Planner



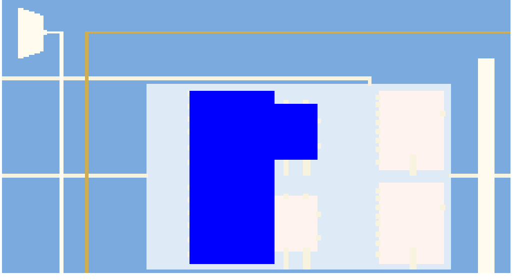
To create a combinational ALM LUT atom, perform the following steps:

1. Right click on the left side of any unused (not shaded) ALM and click **Create Atom**. The **Resource Selection** dialog box appears.
2. In the **Resource Selection** dialog box, select the atom that you wish to create. The lower index number refers to the top combinational node and the higher index refers to the bottom combinational node.
3. Click **OK**. The **Create** <*Altera device*> **LUT Atom** dialog box appears.
4. In the **Atom Name** box, type the name of the resource atom.
5. Under **LUT Mode**, select **Normal**, **Extended**, or **Arithmetic**.
6. If applicable, in the **Partition** list, select the partition that the newly created atom should reside in. The default partition for newly created atoms is the top-level partition.
7. Click **OK**.

For more information about the LUT mode, refer to the data sheet of the appropriate device.

When you have successfully created a combinational output, the combinational element is colored in the Chip Planner. Figure 15–10 shows a combinational ALUT atom.

Figure 15–10. Combinational ALUT Atom



To create a registered ALM atom, perform the following steps:

1. Right click on any ALM register resource and click **Create Atom**. The **Create Register Atom** dialog box appears.
2. In the **Atom Name** box, type the atom name.
3. Click **OK**.

Creating Logic Element Atoms

The Chip Planner shows resource atoms for Stratix, Cyclone, and MAX device families as logic elements. For all other LE-based device families, the Chip Planner shows resource atoms as the combinational output of the logic element LUT and the registered output of the logic element. Figure 15–11 shows an example of an atom resource in the Chip Planner for the Stratix, Cyclone, and MAX devices. Figure 15–12 shows a Cyclone II resource atom in the Chip Planner.

Figure 15–11. Logic Element for Stratix, Cyclone, and MAX Devices in the Chip Planner

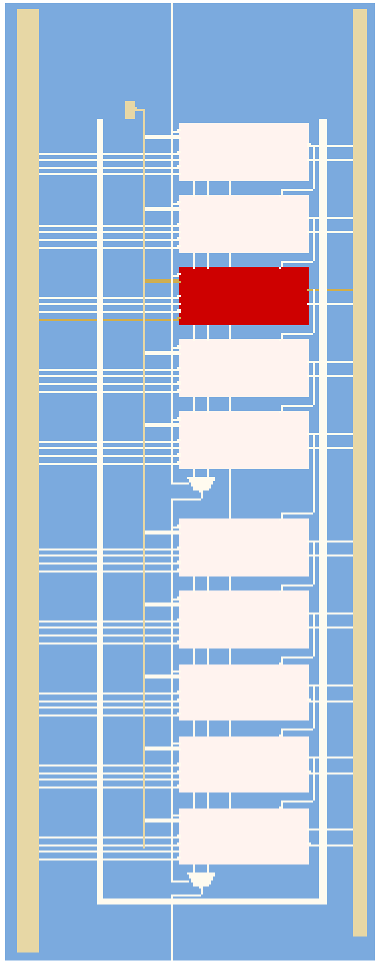
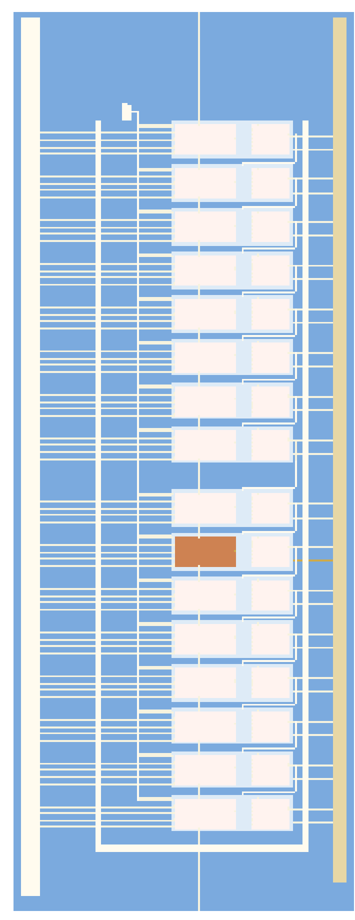


Figure 15–12. Logic Element for a Cyclone II Device in the Chip Planner



To create a logic element resource for Stratix, Cyclone, and MAX device families, perform the following steps:

1. Right-click on any available (unshaded) LE resource and click **Create Atom**. The **Create Logic Cell Atom** dialog box appears.
2. If applicable, in the **Partition** list, select the partition in which the newly created atom should reside. The default partition for newly created atoms is the top-level partition.
3. In the **Atom Name** box, type the atom name.
4. Click **OK**.

To create a combinational resource atom for all other LE-based device families, perform the following steps:

1. Right-click on the left side of an available (unshaded) LE resource and click **Create Atom**. The **Create** <*device family*> **LUT Atom** dialog box appears.
2. If applicable, in the **Partition** list, select the partition in which the newly created atom should reside. The default partition for newly created atoms is the top-level partition.
3. In the **Atom Name** box, type the atom name.

4. Click **OK**.

To create a register resource atom for Cyclone II devices, perform the following steps:

1. Right-click on right side of an available (unshaded) LE resource and click **Create Atom**. The **Create** <*device family*> **Register Atom** dialog box appears.
2. If applicable, in the **Partition** list, select the partition in which the newly created atom should reside. The default partition for newly created atoms is the top-level partition.
3. In the **Atom Name** box, type the atom name.
4. Click **OK**.

Deleting Atoms

To delete a resource atoms, right-click on the desired resource atom in the Chip Planner and click **Delete Atom**.

You can delete a resource only after all of its fan-out connections are removed. Protected resources, such as resources in megafunctions or IP cores, cannot be deleted.

Refer to "Resource Property Editor" on page 15–16 for more details about removing fan-out connections.

Moving Atoms

You can move resource ALMs, LEs, and FPGA I/O atoms by clicking on the desired resource and dragging the selected atom to a free resource atom. Moving nodes as an ECO can only be done in the ECO editing mode. Changes made while in Assignment mode create location constraints on the design and require a recompilation to incorporate the change.

Resource atoms from protected resources, such as resources of megafunction IP cores, cannot be moved.

Check and Save Netlist Changes

After making all ECOs, you can run the Fitter to incorporate the changes by clicking on the Check and Save Netlist Changes icon in the Chip Planner toolbar. The Fitter compiles the ECO changes, performs design rule checks on the design, and generates a programming file.

Resource Property Editor

You can view and edit the following resources with the Resource Property Editor:

- "Logic Element" on page 15–17
- "Adaptive Logic Module" on page 15–19
- "FPGA IOEs" on page 15–21
- "PLL Properties" on page 15–35
- "FPGA RAM Blocks" on page 15–27
- "FPGA DSP Blocks" on page 15–28

Logic Element

An Altera LE contains a four-input LUT, which is a function generator that can implement any function of four variables. In addition, each LE contains a register that is fed by the output of the LUT or by an independent function generated in another LE.

You can use the Resource Property Editor to view and edit any LE in the FPGA. Open the Resource Property Editor for an LE by pointing to **Locate** on the Project menu and clicking **Locate in Resource Property Editor** in one of the following views:

- RTL Viewer
- Technology Map Viewer
- Node Finder
- Chip Planner

For more information about LE architecture for a particular device family, refer to the device family handbook or data sheet.

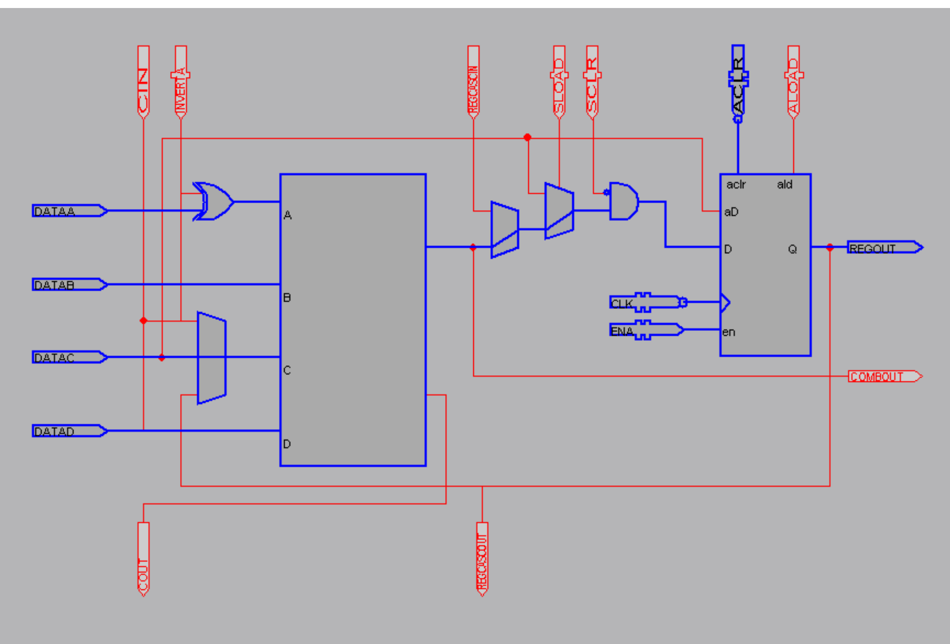
You can use the Resource Property Editor to change the following LE properties:

- Data input to the LUT
- LUT mask or LUT equation

Logic Element Schematic View

Figure 15–13 shows how the LE appears in the Resource Property Editor.

Figure 15–13. Stratix LE Architecture *(Note 1)*, *(2)*



Notes to Figure 15–13:

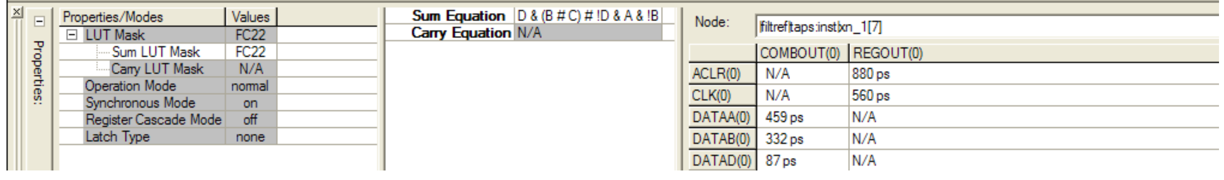
(1) By default, the Quartus II software displays the used resources in blue and the unused in gray. For Figure 15–13, the used resources are in blue and the unused resources are in red.

(2) For more information about the Stratix device's LE Architecture, refer to the *Stratix Device Handbook.*

LE Properties

Figure 15–14 shows an example of the properties that can be viewed for a selected LE in the Resource Property Editor. To view LE properties, on the View menu, click **View Properties** to view these properties.

Figure 15–14. LE Properties



Modes of Operation

LUTs in an LE can operate in either normal or arithmetic mode.

For more information about LE modes of operation, refer to volume 1 of the appropriate device handbook.

When an LE is configured in normal mode, the LUT in the LE can implement a function of four inputs.

When the LE is configured in arithmetic mode, the LUT in the LE is divided into two 3-input LUTs. The first LUT generates the signal that drives the output of the LUT, while the second LUT generates the carry-out signal. The carry-out signal can only drive a carry-in signal of another LE.

Sum and Carry Equations

You can change the logic function implemented by the LUT by changing the sum and carry equations. When the LE is configured in normal mode, you can only change the SUM equation. When the LE is configured in arithmetic mode, you can change both the SUM and the CARRY equations.

The LUT mask is the hexadecimal representation of the LUT equation output. When you change the LUT equation, the Quartus II software automatically changes the LUT mask. Conversely, when you change the LUT mask, the Quartus II software automatically computes the LUT equation.

sload and sclear Signals

Each LE register contains a synchronous load (`sload`) signal and a synchronous clear (`sclr`) signal. You can invert either the `sload` or `sclr` signal feeding into the LE. If the design uses the `sload` signal in an LE, the signal and its inversion state must be the same for all other LEs in the same LAB. For example, if two LEs in a LAB have the `sload` signal connected, both LEs must have the `sload` signal set to the same value. This is also true for the `sclr` signal.

Register Cascade Mode

When register cascade mode is enabled, the cascade-in port feeds the input to the register. The register cascade mode is used most often when the design implements shift registers. You can change the register cascade mode by connecting (or disconnecting) the cascade in the port. However, if you create this port, you must ensure that the source port LE is directly above the destination LE.

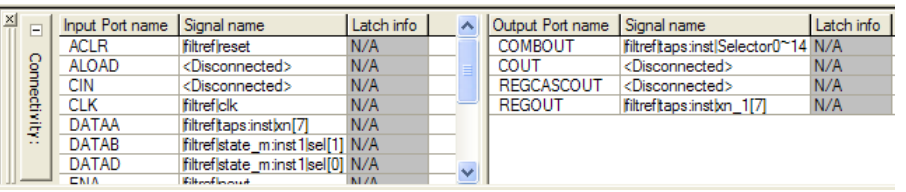
Cell Delay Table

The cell delay table describes the propagation delay from all inputs to all outputs for the selected LE.

LE Connections

On the View menu, click **View Port Connections** to view the connections that feed in and out of an LE. Figure 15–15 shows the LE connections in the Connectivity window.

Figure 15–15. View Connections



Delete an LE

To delete an LE, perform the following steps:

1. Right click the desired LE in the Chip Planner, point to **Locate** and click **Locate in Resource Property Editor**.
2. You must remove all fan-out connections from an LE prior to deletion. To delete fan-out connections, right-click each connected output signal, point to remove, and click **Fanouts**. Select all of the fan-out signals in the dialog box that appear and click **OK**.
3. To delete an atom after all fan-out connections are removed, right-click the atom in the Chip Planner and click **Delete Atom**.

Adaptive Logic Module

Each ALM contains LUT-based resources that can be divided between two adaptive LUTs (ALUTs). With up to eight inputs to the two ALUTs, each ALM can implement various combinations of two functions. This adaptability allows the ALM to be completely backward-compatible with four-input LUT architectures. One ALM can implement any function with up to six inputs and certain seven-input functions. In addition to the adaptive LUT-based resources, each ALM contains two programmable registers, two dedicated full adders, a carry chain, a shared arithmetic chain, and a register chain. The ALM can efficiently implement various arithmetic functions and shift registers using these dedicated resources.

You can implement the following types of functions in a single ALM:

- Two independent 4-input functions
- An independent 5-input function and an independent 3-input function
- A 5-input function and a 4-input function, if they share one input
- Two 5-input functions, if they share two inputs
- An independent 6-input function
- Two 6-input functions, if they share four inputs and share function
- Certain 7-input functions

You can use the Resource Property Editor to change the following ALM properties:

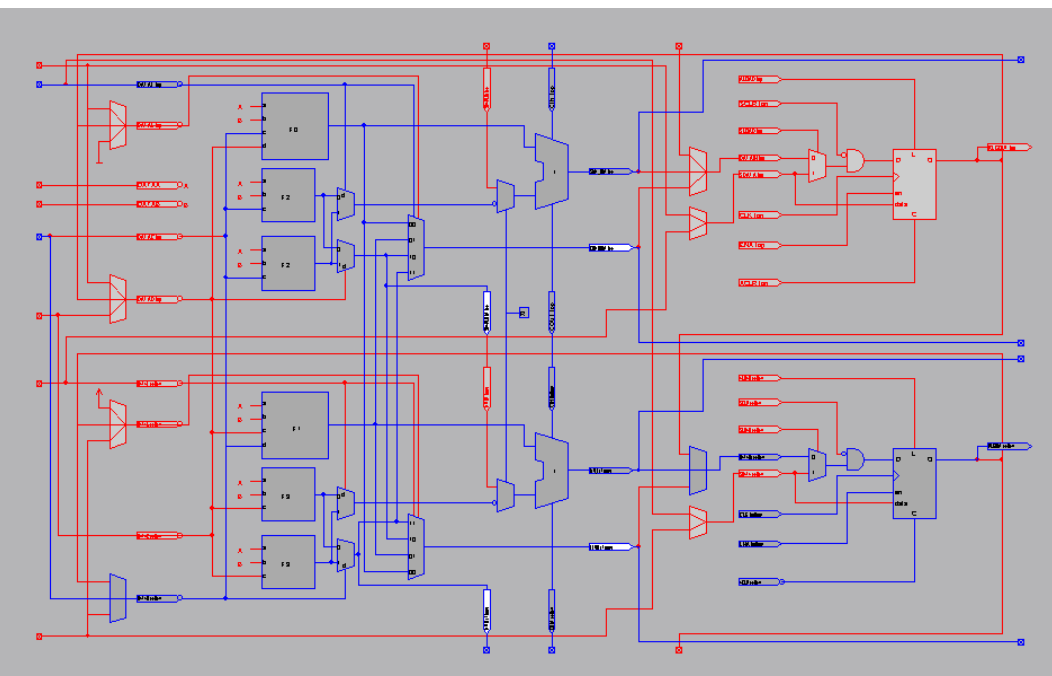
- Data input to the LUT
- LUT mask or LUT equation

ALM Schematic

You can view and edit any ALM atom with the Resource Property Editor by right-clicking the ALM in the RTL Viewer, the Node Finder, or the Chip Planner, and clicking **Locate in Resource Property Editor** (Figure 15–16).

For a detailed description of the ALM, refer to the device handbooks of devices based on an ALM architecture.

Figure 15–16. ALM Schematic *(Note 1)*



Note to Figure 15–16:

(1) By default, the Quartus II software displays the used resources in blue and the unused in gray. For Figure 15–16, the used resources are in blue and the unused resources are in red.

ALM Properties

The properties that you can display for the ALM include an equations table that shows the name and location of each of the two combinational nodes and two register nodes in the ALM, the individual LUT equations for each of the combinational nodes, and the `combout`, `sumout`, `carryout`, and `shareout` equations for each combinational node.

ALM Connections

On the View menu, click **View Port Connections** to view the connections that feed in and out of an ALM.

FPGA IOEs

Altera FPGAs that have high-performance IOEs, including up to six registers, are equipped with support for a number of I/O standards that allow you to run your design at peak speeds. Use the Resource Property Editor to view, change connectivity, and edit the properties of the IOEs. Use the Chip Planner (Floorplan view) to change placement, delete, and create new IOEs.

For a detailed description of the device IOEs, refer to the applicable device handbook.

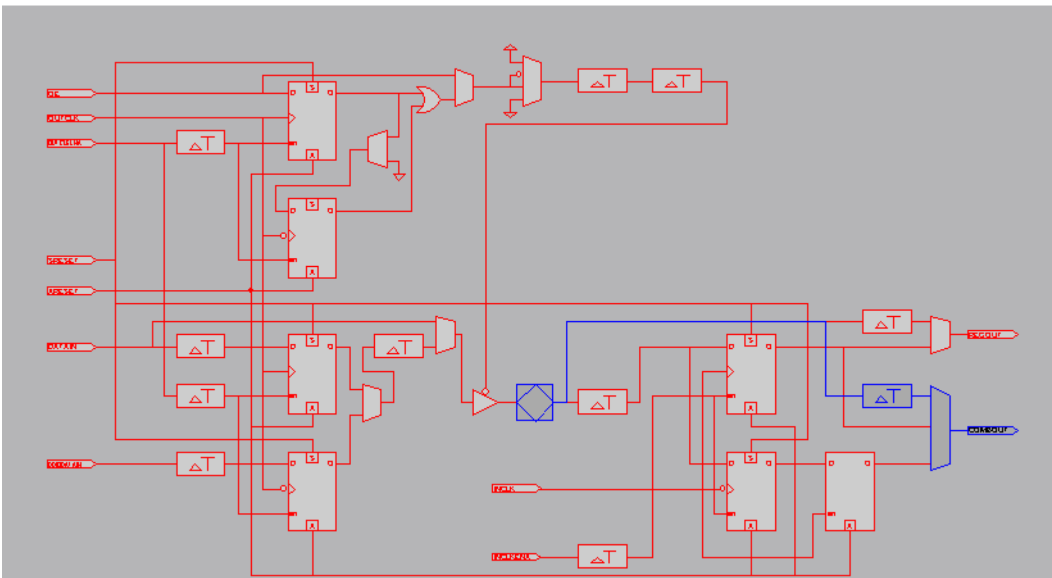
You can change the following I/O properties:

- Delay chain
- Bus hold
- Weak pull up
- Slow slew rate
- I/O standard
- Current strength
- Extend OE disable
- PCI I/O
- Register reset mode
- Register synchronous reset mode
- Register power up
- Register mode

Arria GX, Stratix II, Stratix, and Stratix GX IOEs

The IOEs in Stratix series device families and Arria GX devices contain a bidirectional I/O buffer, six registers, and a latch for a complete bidirectional single data rate or DDR transfer. Figure 15–17 shows the Stratix and Stratix GX IOE structure. The IOE structure contains two input registers (plus a latch), two output registers, and two output enable registers.

Figure 15–17. Stratix and Stratix GX Device IOE and Structure *(Note 1)*, *(2)*



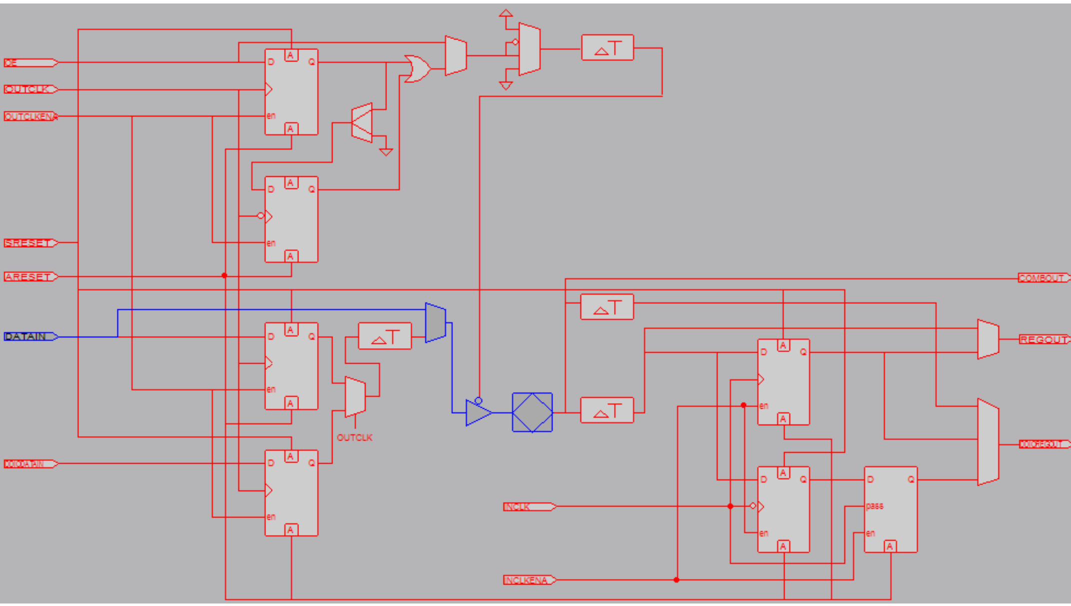
Notes to Figure 15–17:

(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 15–17, the used resources are in blue and the unused resources are in red.

(2) For more information about IOEs in Stratix and Stratix GX devices, refer to the *Stratix Device Handbook* and the *Stratix GX Device Handbook*.

Figure 15–18 shows the Arria GX and Stratix II IOE structures.

Figure 15–18. Arria GX Device and Stratix II IOE and Structure *(Note 1)*, *(2)*, *(3)*



Notes to Figure 15–18:

(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 15–18, the used resources are in blue and the unused resources are in red.

(2) For more information about IOEs in Arria GX and Stratix II devices, refer to the appropriate device handbook.

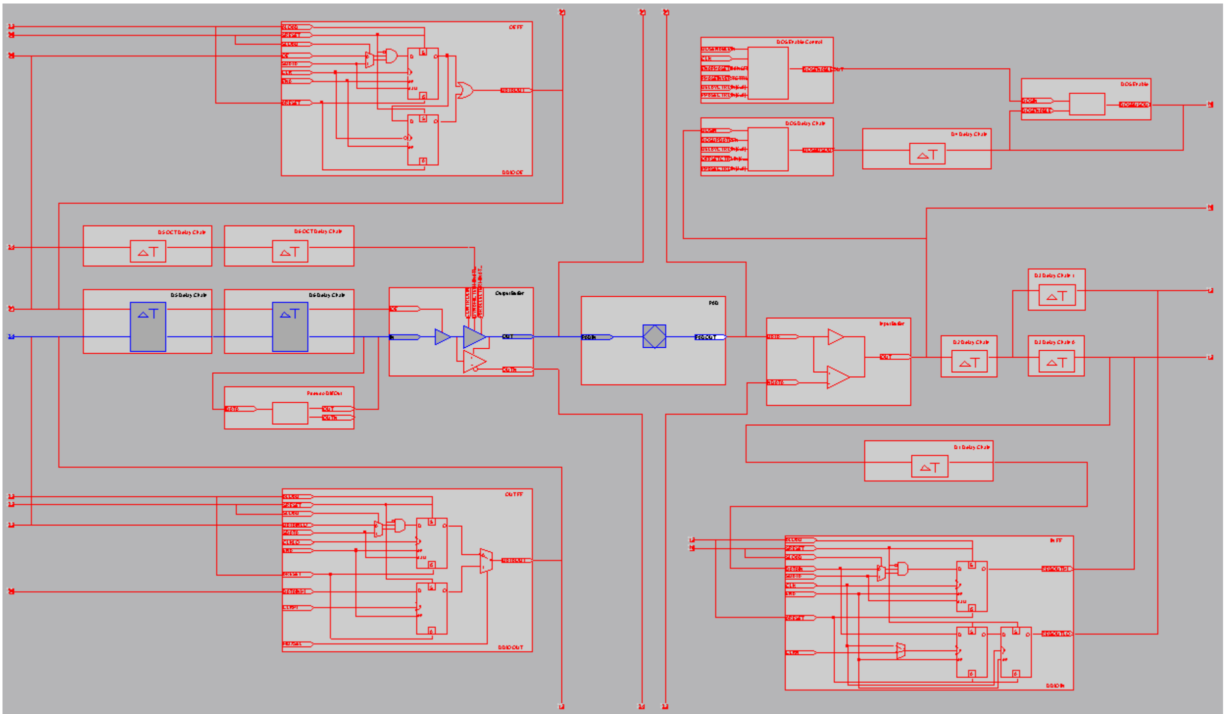
(3) Current IOE shown in a DQS pin. Non-DQS pins do not contain DQS delay circuitry.

Stratix III IOE

The IOE in Stratix III devices contain a bi-directional I/O buffer and I/O registers to support a complete embedded bi-directional single data rate or DDR transfer (shown in Figure 15–19). The I/O registers are composed of the input path for handling data from the pin to the core, the output path for handling data from the core to the pin, and the output enable (OE) path for handling the OE signal for the output buffer. Each path consists of a set of delay elements that allow you to fine-tune the timing characteristics of each path for skew management.

For more information about programmable IOEs in Stratix III devices, refer to *AN 474: Implementing Stratix III Programmable I/O Delay Settings in the Quartus II Software*.

Figure 15–19. Stratix III Device IOE and Structure *(Note 1)*, *(2)*



Notes to Figure 15–19:

(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 15–19, the used resources are in blue and the unused resources are in red.

(2) For more information about IOEs in Stratix III devices, refer to the *Stratix III Device Handbook*. For Stratix IV devices, refer to the *Stratix IV Device Handbook*.

Cyclone II and Cyclone IOEs

The IOEs in Cyclone II and Cyclone devices contain a bidirectional I/O buffer and three registers for complete bidirectional single data-rate transfer. Figure 15–20 shows the Cyclone II and Cyclone IOE structure. The IOE contains one input register, one output register, and one output enable register.

Figure 15–20. Cyclone II and Cyclone Device IOEs and Structure *(Note 1)*, *(2)*

Notes to Figure 15–20:

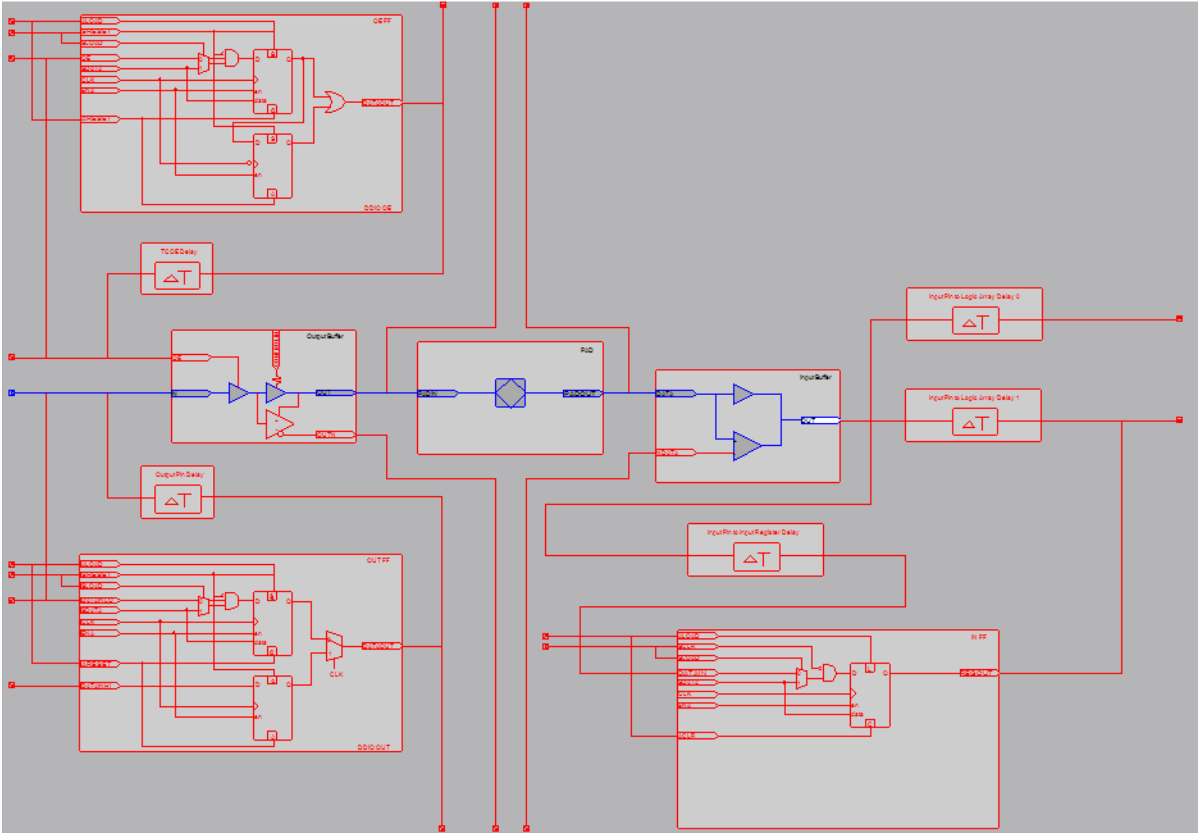
(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 15–20, the used resources are in blue and the unused resources are in red.

(2) For more information about IOEs in Cyclone II and Cyclone devices, refer to the *Cyclone II Device Handbook* and *Cyclone Device Handbook*, respectively.

Cyclone III IOEs

Cyclone III device IOEs contain a bidirectional I/O buffer and five registers for complete embedded bidirectional single-data rate transfer. Figure 15–21 shows the Cyclone III IOE structure. The IOE contains one input register, two output registers, and two output-enable registers. The two output registers and two output-enable registers are utilized for double-data rate (DDR) applications. You can use the input registers for fast setup times and the output registers for fast clock-to-output times. Additionally, you can use the output-enable (OE) registers for fast clock-to-output enable timing. You can use IOEs for input, output, or bidirectional data paths.

Figure 15–21. Cyclone III Device IOEs and Structure *(Note 1)*, *(2)*



Notes to Figure 15–21:

(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 15–21, the used resources are in blue and the unused resources are in red.

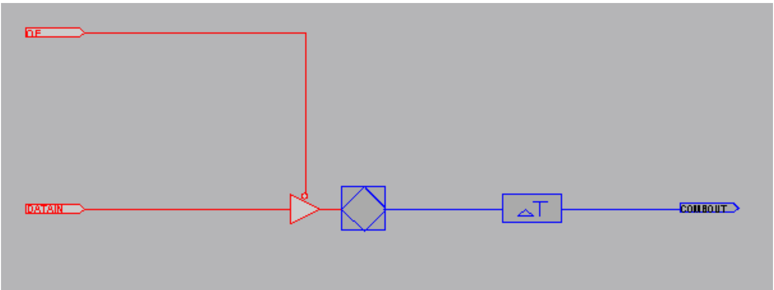
(2) For more information about IOEs in Cyclone III devices, refer to the *Cyclone III Device Handbook*.

© March 2009 Altera Corporation

MAX II IOEs

MAX II device IOEs contain a bidirectional I/O buffer. Figure 15–22 shows the MAX II IOE structure. Registers from adjacent LABs can drive to or be driven from the IOE's bidirectional I/O buffers.

Figure 15–22. MAX II Device IOEs and Structure *(Note 1)*, *(2)*



Notes to Figure 15–22:

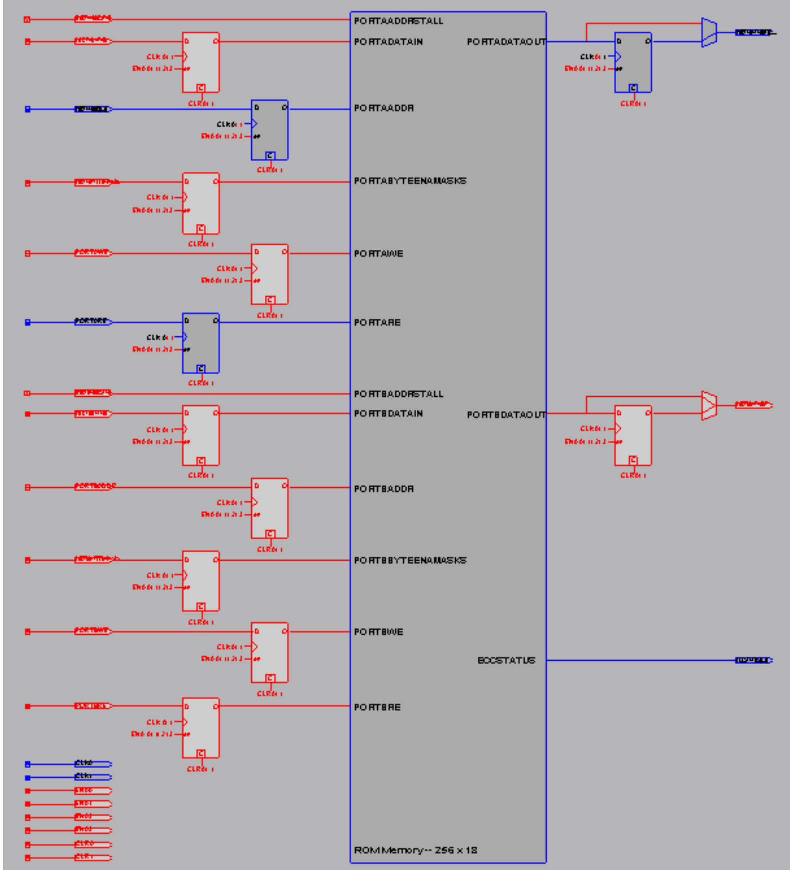
(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 15–22, the used resources are in blue and the unused resources are in red.

(2) For more information about IOEs in MAX II devices, refer to the *MAX II Device Handbook*.

FPGA RAM Blocks

With the Resource Property Editor, you can view the architecture of different RAM blocks in the device, modify the input and output registers from the RAM blocks, and modify the connectivity of the input and output ports. Figure 15–23 shows a M9K RAM view in a Stratix III device.

Figure 15–23. M9k RAM View in a Stratix III Device *(Note 1)*



Note to Figure 15–23:

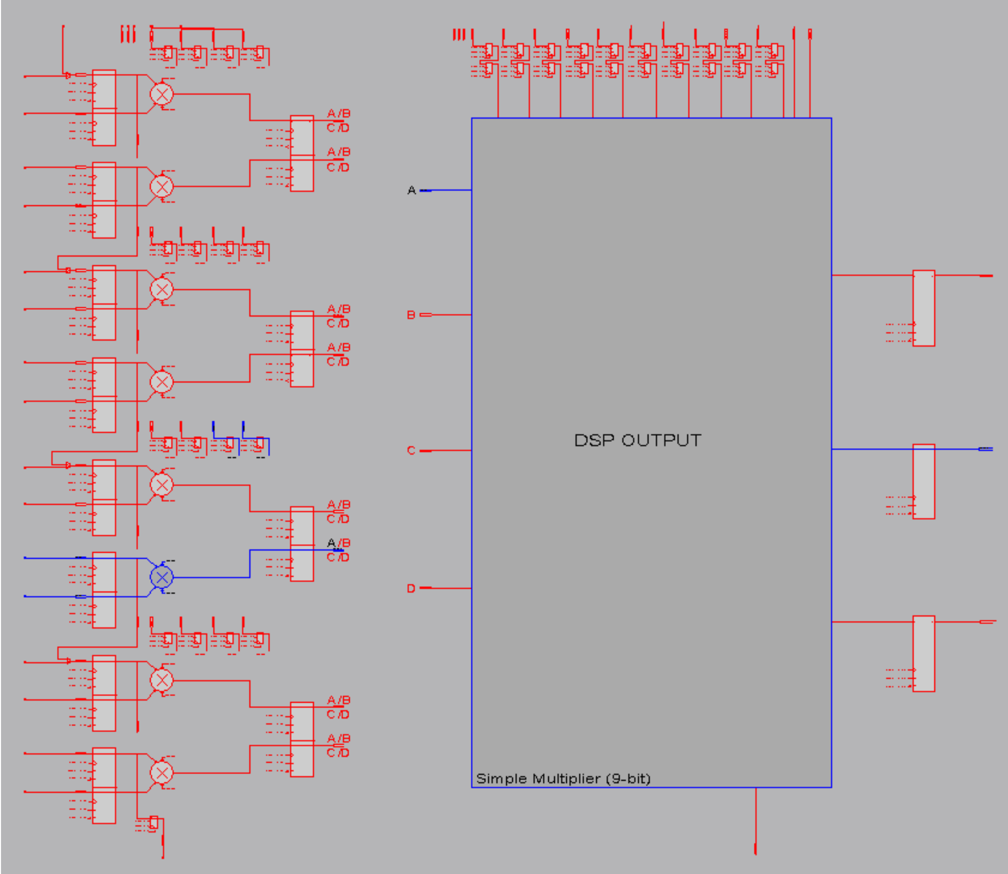
(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 15–23, the used resources are in blue and the unused resources are in red.

FPGA DSP Blocks

Dedicated hardware DSP circuit blocks in Altera devices provide performance benefits for the critical DSP functions in your design. The Resource Property Editor allows you to view the architecture of DSP blocks in the Resource Property Editor for the Stratix and Cyclone series of devices. The Resource Property Editor also allows you to modify the signal connections to and from the DSP blocks and modify the input and output registers from the DSP blocks.

Figure 15–24 shows a view of a DSP architecture in a Stratix III device.

Figure 15–24. DSP Block View in a Stratix III Device *(Note 1)*



Note to Figure 15–24:

(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 15–24, the used resources are in blue and the unused resources are in red.

Change Manager

The Change Manager maintains a record of every change that you perform with the Resource Property Editor. Each row in the Change Manager represents one ECO performed. The changes are numbered sequentially, such that the larger the number, the more recent the change.

More complex changes are marked in the Change Manager with a plus icon. You can expand a complex entry in the Change Manager by clicking the plus icon to reveal all the changes that occurred. An example of a complex change is the creation or deletion of an atom.

Table 15–1 summarizes the information shown by the Change Manager.

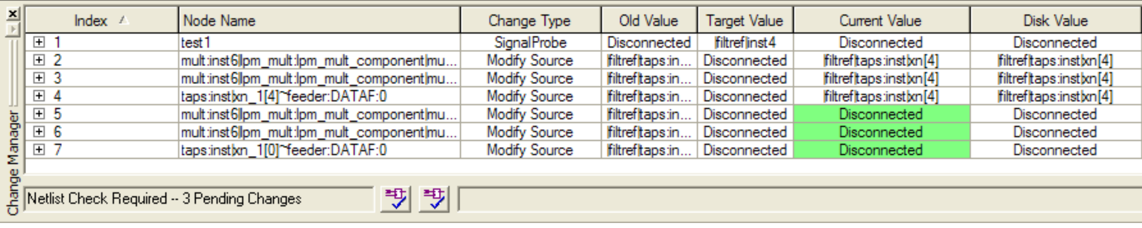
Table 15–1. Change Manager Information

Column Name	Description
Index	Identifies, by a sequential number, change records corresponding to changes made in the Chip Planner or Resource Property Editor. In the case of complex change records, the index column identifies not only the main change, but also any component changes.
Node Name	Uniquely identifies the resource to which a change has been made.
Change Type	Identifies the type of change that has been made to the resource.
Old Value	Lists the value of the resource immediately prior to the change being made.
Target Value	Lists the desired target value (new value) that you have established using the Resource Property Editor, Chip Planner, or SignalProbe.
Current Value	Lists the value of the resource in the netlist that is currently active in memory (as opposed to the value in the netlist saved on disk, which may be different if you have made changes and not yet used the **Check & Save All Netlist Changes** command).
Disk Value	Lists the current value of the resource on disk.
Comment	Lets you add a comment to a change record in the Change Manager. To add a comment to a change record, double-click in the **Comment** field of the record you want to annotate, and type the desired comment.

After you complete all of your design modifications, check the integrity of the netlist by right-clicking in the Change Manager and clicking **Check & Save All Netlist Changes**. If the applied changes successfully pass the netlist check, they are written to disk. If the changes do not pass the netlist check, all changes made since the previous successful netlist check are reversed. Figure 15–25 shows the Change Manager.

Colored indicators in the **Current Value** and **Disk Value** columns indicate the present status of the data in those columns. Green in the **Current Value** column indicates that the change has been recorded. Blue in the **Disk Value** column indicates that the change has successfully passed a **Check & Save Netlist Changes** operation. Choose **Check & Save All Netlist Changes**.

Figure 15–25. Change Manager Results



For more information about SignalProbe pins, refer to the *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*.

Each line in the Change Manager represents a change record. Simple changes appear as a single line. More complex changes, which require that several actions be performed to achieve the change, appear as a single line marked by a plus icon. Click the plus icon to show all the component actions performed as part of the change.

Complex Changes in the Change Manager

Certain types of changes that you make in the Resource Property Editor or the Chip Planner (including creating or deleting atoms and changing connectivity) can appear to be self-contained, but these changes are actually composed of multiple actions. Complex changes are indicated with a plus icon in the **Index** column.

The change record in the Change Manager is a single-line representation of the actual change actions that occurred. You expand the change record to show the component actions that make up the change by clicking the plus icon.

After expanding a change entry in the Change Manager, you can see that creation of an atom consists of three actions:

- The creation of a new logic cell
- The creation of an output port on the newly created logic cell
- The assignment of a location index to the newly created logic cell

You cannot select individual components of a complex change record; if you select any part of a complex change record, the entire complex change record is selected.

For examples of managing changes with the Change Manager, refer to "Example of Managing Changes With the Change Manager" in the Quartus II Help.

Managing SignalProbe Signals

The SignalProbe pins that you create from the **SignalProbe Pins** dialog box are recorded in the Change Manager. After you have created a SignalProbe assignment, you can use the Change Manager to quickly disable SignalProbe assignments by selecting **Revert to Last Saved Netlist** from the right-click menu in the Change Manager.

For more information about SignalProbe pins, refer to the *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*.

Exporting Changes

You can export all your changes to a tool command language (Tcl) script, a Comma Separated Value (**.csv)** file, or a Text **(.txt)** file. The Tcl file enables you to write a script that reapplies changes that were deleted by compilation. You can also write a script that applies to other Quartus II software projects that you create. The **.csv** or **.txt** files provide a list of changes in a tabular format. To export changes, perform the following steps:

1. On the right-click menu, click **Export Changes.**
2. Specify the Tcl file name.
3. Click **OK**.

The resulting Tcl script can also implement similar changes to another Quartus II design.

Using Incremental Compilation in the ECO Flow

Beginning with the Quartus II software version 6.1, the incremental compilation feature is turned on by default. The top-level design is automatically set to a design partition when the incremental compilation feature is on. A design partition during incremental compilation can have different netlist types (netlist types can be set to source HDL, post synthesis, or post-fit). The netlist type indicates whether that partition should be resynthesized or refit during the recompilation. Incremental compilation saves you time and preserves the placement of unchanged partitions in your design if small changes must be made to some partitions late in the design cycle.

For more information about partitions, their netlist types, and Quartus II incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

The behavior of ECOs during an incremental compilation depends on the netlist type of your design partitions. The Quartus II software preserves ECOs if the partition containing the ECO satisfies the following two conditions:

- The netlist type of the affected partition is set to post-fit.
- There are no source code changes in the affected partition that would cause the partition to be resynthesized during recompilation.

If you have ECOs that affect multiple partitions in your design, the Quartus II software preserves your ECOs during recompilation if any of the affected partitions are set to post-fit.

Whenever an ECO affects multiple partitions, all of the affected partitions become linked. All of the higher-level "parent" partitions up to their nearest common parent are also linked. In such cases, the connection between the partitions is actually defined outside of the two partitions immediately affected, so all the partitions must be compiled together. The linked partition inherits the netlist type of the partition with the highest level of preservation. For example, if an ECO is performed on a lower-level partition of a post-fit type and a top-level partition of a post-synthesis type, the two partitions will be linked and will have a post-fit netlist type.

If the partitions are set to use the source code or a post-synthesis netlist, the software issues a warning and the post-fit ECO changes are not included in the new compilation.

For example, if your top-level partition netlist type is set to post-synthesis, and either you have no other lower-level partitions or the lower-level partitions netlist type is also set to post-synthesis, during recompilation, your ECOs are not preserved and a warning message appears in the messages window, indicating that ECO modifications are discarded; however, all of the ECO information is retained in the Change Manager. In this case, you can apply ECOs from the Change Manager and perform the **Check & Save All Netlist Changes** step as described in "ECO Flow without Quartus II Incremental Compilation" on page 15–33.

ECO Flow without Quartus II Incremental Compilation

If you do not use the Quartus II incremental compilation feature and have implemented ECOs, those ECOs are not preserved during recompilation of your design; however, all of the ECOs remain in the Change Manager. To apply an ECO, right click the Change Manager and click **Apply Selected Change**. (If the Change Manager window is not visible at the bottom of your screen, from the View menu, point to **Utility Windows** and click **Change Manager**.)

After applying the selected ECO, perform one of the following steps:

- From the menu within the Change Manager, click **Check & Save All Netlist Changes**.

  or

- From the Processing menu, point to **Start** and click **Start Check & Save All Netlist Changes**.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. The Tcl commands for controlling the Chip Planner are located in the `chip_planner` package of the `quartus_cdb` executable. A comprehensive list of Tcl commands for the Chip Planner can be found in the *Quartus Scripting Reference Manual*.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about all settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Reference Manual*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Common ECO Applications

This section provides examples of the situations in which you might use an ECO to make a post-compilation change to your design. To help build your system quickly, you can use Chip Planner functions to perform the following activities:

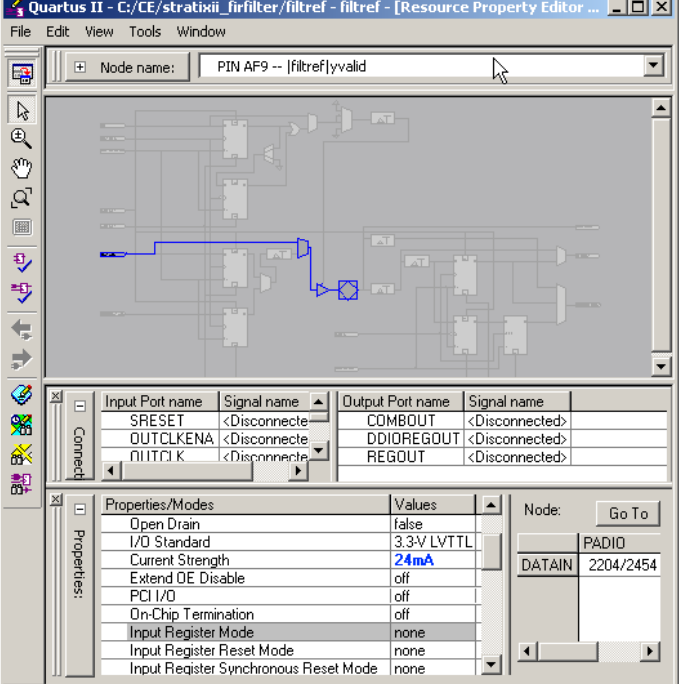
- Adjust the drive strength of an I/O using the Chip Planner
- Modify the PLL properties using the Resource Property Editor (see “Modify the PLL Properties Using the Chip Planner” on page 15–34)
- Modify the connectivity between new resource atoms

Adjust the Drive Strength of an I/O Using the Chip Planner

To adjust the drive strength of an I/O, follow the steps in this section to run the Fitter and assembler to incorporate the ECO changes into the netlist of the design.

1. In the Chip Planner, select the **Post Compilation Editing (ECO)** task.
2. Locate the I/O in the **Resource Property Editor**, as shown in Figure 15–26.

Figure 15–26. I/O in the Resource Property Editor



3. Click the **Current Strength** box for the selected I/O, then click **Edit**.
4. Change the value for the desired current strength.
5. Right-click the ECO change in the Change Manager tool and click **Check & Save All Netlist Changes** to apply the ECO change.

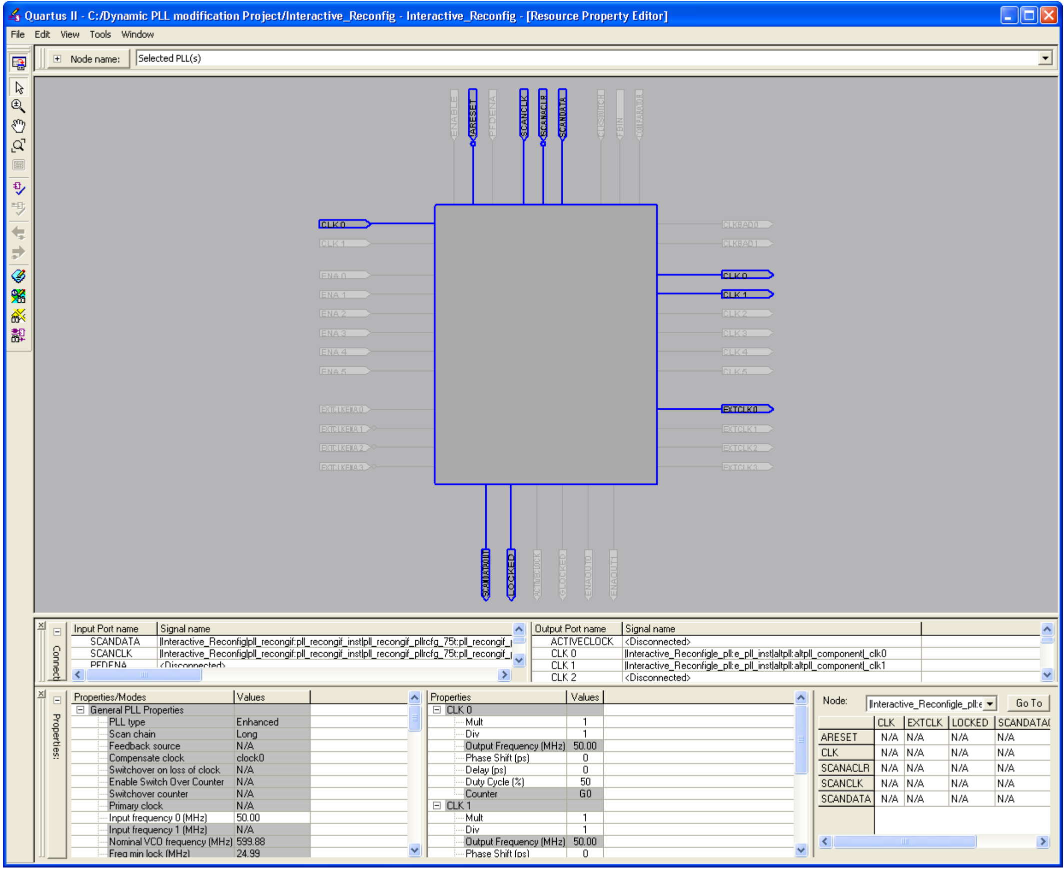
Changing the pin locations of input/output ports can be done using the ECOs flow as well. You can drag and move the signal from an existing pin location to a new location while in the Post Compilation Editing (ECO) task in the Chip Planner. Afterward, you can set **Check & Save All Netlist Changes** to compile the ECO.

Modify the PLL Properties Using the Chip Planner

PLLs are used to modify and generate clock signals to meet design requirements. Additionally, PLLs are used for distributing clock signals to different devices in a design, reducing clock skew between devices, improving I/O timing, and generating internal clock signals.

The Resource Property Editor enables you to view and modify PLL properties to meet your design requirements. Using the Stratix PLL as an example, the rest of this section describes the adjustable PLL properties and the equations as a function of the adjustable PLL properties that govern the PLL output parameters. Figure 15–27 shows a Stratix PLL as shown in the Resource Property Editor.

Figure 15–27. PLL View in a Stratix Device



PLL Properties

The Resource Property Editor enables you to modify PLL options, such as phase shift, output clock frequency, and duty cycle. You can also change the following PLL properties with the Resource Property Editor:

- Input frequency
- M V_{CO} tap
- M initial
- M value
- N value
- M counter delay
- N counter delay
- M2 value
- N2 value
- SS counter
- Charge pump current
- Loop filter resistance

- Loop filter capacitance
- Counter delay
- Counter high
- Counter low
- Counter mode
- Counter initial
- V_{CO} tap

Post-compilation PLL properties can also be viewed in the Compilation Report. To do so, in the Compilation Report, select **Fitter** and then select **Resource Section**.

Adjusting the Duty Cycle

Use Equation 15–1 to adjust the duty cycle of individual output clocks.

Equation 15–1.

$$\text{High \%} = \frac{\text{Counter High}}{(\text{Counter High} + \text{Counter Low})}$$

Adjusting the Phase Shift

Use Equation 15–2 to adjust the phase shift of an output clock of a PLL.

Equation 15–2.

$$\text{Phase Shift} = (\text{Period } V_{CO} \times 0.125 \times \text{Tap } V_{CO}) + (\text{Initial } V_{CO} \times \text{Period } V_{CO})$$

For normal mode Period V_{CO}, Tap V_{CO}, and Initial V_{CO} are governed by the following settings:

$$\text{Tap } V_{CO} = \text{Counter Delay} - \text{M Tap } V_{CO}$$

$$\text{Initial } V_{CO} = \text{Counter Initial} - \text{M Initial}$$

$$\text{Period } V_{CO} = \text{In Clock Period} \times N \div M$$

For external feedback mode, Tap V_{CO}, Initial V_{CO}, and Period V_{CO} are governed by the following settings:

$$\text{Tap } V_{CO} = \text{Counter Delay} - \text{M Tap } V_{CO}$$

$$\text{Initial } V_{CO} = \text{Counter Initial} - \text{M Initial}$$

$$\text{Period } V_{CO} = \frac{\text{In Clock Period} \times N}{(M + \text{Counter High} + \text{Counter Low})}$$

For a detailed description of the settings, refer to the Quartus II Help. For more information about Stratix device PLLs, refer to the *Stratix Architecture* chapter in volume 1 of the *Stratix Device Handbook*. For more information about PLLs in Arria GX, Stratix II, Cyclone II, and Cyclone devices, refer to the appropriate device handbook.

Adjusting the Output Clock Frequency

Use Equation 15–3 to adjust the PLL output clock in normal mode.

Equation 15–3.

$$\text{Output Clock Frequency} = \text{Input Frequency} \bullet \frac{M \text{ value}}{N \text{ Value} + \text{Counter High} + \text{Counter Low}}$$

Use Equation 15–4 to adjust the PLL output clock in external feedback mode.

Equation 15–4.

$$\text{OUTCLK} = \frac{M \text{ value} + \text{External Feedback Counter High} + \text{External Feedback Counter Low}}{N \text{ value} + \text{Counter High} + \text{Counter Low}}$$

Adjusting the Spread Spectrum

Use Equation 15–5 to adjust the spread spectrum for your PLL.

Equation 15–5.

$$\% \text{ spread} = \frac{M_2 N_1}{M_1 N_2}$$

Modify the Connectivity between Resource Atoms

The Chip Planner and Resource Property Editor allow you to create new resource atoms and manipulate the existing connection between resource atoms in the post-fit netlist. This feature is useful for small changes during the debugging stage, such as manually inserting pipeline registers into a combinational path that fails timing, or to route a signal to a spare I/O pin for analysis. Use the following procedure to create a new register in a Cyclone III device and route register output to a spare I/O pin. This example illustrates the mechanics of creating a new resource atom and modifying the connections between resource atoms.

To create new resource atoms and manipulate the existing connection between resource atoms in the post-fit netlist, perform the following steps:

1. Create a new register in the Chip Planner Floorplan.
2. Locate the atom in the Resource Property Editor.
3. Assign a clock signal to the register: Right-click the clock input port for the register, point to **Edit connection,** and click **Other**. Use the Node Finder to assign a clock signal from your design.
4. Tie the SLOAD input port to V_{CC}: Right-click the clock input port for the register, point to **Edit connection**, and click **VCC**.
5. Assign a data signal from your design to the SDATA port.
6. In the connectivity window, under the output port names, copy the port name of the register.
7. In the Chip Planner Floorplan, locate a free I/O resource and create an output buffer.

8. Locate the new I/O atom in the Resource Property Editor.
9. On the input port to the output buffer, right-click, point to **Edit connection**, and click **Other**.
10. In the dialog box that appears, type the output port name of the register you have created.
11. Run the ECO Fitter to apply the changes by clicking the **Check and Save Netlist Changes** button.

A successful ECO connection is subject to the available routing resources. You can view the relative routing utilization by selecting Routing Utilization as the Background Color Map in the **Layers Settings** dialog box. Also, you can view individual routing channel utilization from local, row, and column interconnects using the tooltips created from positioning your mouse pointer over the appropriate resource. Refer to the device data sheet for more information about the architecture of the routing interconnects of your device.

Post ECO Steps

This section describes the operations you can perform after making an ECO change with the Chip Planner.

Performing Static Timing Analysis

After you make an ECO change with the Chip Planner, you must perform static timing analysis of your design with either the Quartus II Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer to ensure that your changes have not adversely affected your design's timing performance.

For example, when you turn on one of the delay chain settings for a specific pin, you change the I/O timing. Therefore, to ensure that all timing requirements are still met, you should perform static timing analysis.

Whenever you change your design using the Chip Planner, Altera also recommends that you perform a gate-level timing simulation on your design with either the Quartus II Simulator or a third-party EDA simulation tool.

For more information about performing a static timing analysis of your design, refer to the *Quartus II Classic Timing Analyzer* or *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the Quartus II *Handbook*.

Conclusion

As the time-to-market pressure mounts, it is more and more important to produce a fully functional design in the shortest amount of time. To address this challenge, Altera developed the Chip Planner in the Quartus II software suite. The Chip Planner enables you to analyze and modify your design floorplan. Also, ECO changes made with the Chip Planner do not require a full recompilation, eliminating the lengthy process of RTL modification, resynthesis, and another place-and-route cycle. In summary, the Chip Planner shortens the verification cycle and brings timing closure to your design in a shorter period of time.

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *AN 474: Implementing Stratix III Programmable I/O Delay Settings in the Quartus II Software*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Cyclone Device Handbook*
- *MAX II Device Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Programmer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*
- *Stratix Architecture* chapter in volume 1 of the *Stratix Device Handbook*
- *Stratix Device Handbook*
- *Stratix III Device Handbook*
- *Stratix IV Device Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 15–2 shows the revision history for this chapter.

Table 15–2. Document Revision History

Date and Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Updated Figure 15–19 ■ Made minor editorial updates ■ Chapter 15 was previously Chapter 13 in the 8.1.0 release.	Updated for the Quartus II software release 9.0.
November 2008 v8.1.0	Corrected preservation attributes for ECOs in the section “Using Incremental Compilation in the ECO Flow” on page 15–32. Minor editorial updates. Changed to 8½” x 11” page size.	Updated for the Quartus II software release 8.1.
May 2008 v8.0.0	■ Updated device support list ■ Modified description for ECO support for block RAMs and DSP blocks ■ Corrected Stratix PLL ECO example ■ Added an application example to show modifying the connectivity between resource atoms	Updated for the Quartus II software release 8.0.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 9.0.

How to Contact Altera

For the most up-to-date information about Altera products, see the following table.

Contact *(Note 1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note:

(1) You can also contact your local Altera sales office or sales representative.

Third-Party Software Product Information

Third-party software products described in this handbook are not Altera products, are licensed by Altera from third parties, and are subject to change without notice. Updates to these third-party software products may not be concurrent with Quartus II software releases. Altera has assumed responsibility for the selection of such third-party software products and its use in the Quartus II 9.0 software release. To the extent that the software products described in this handbook are derived from third-party software, no third party warrants the software, assumes any liability regarding use of the software, or undertakes to furnish you any support or information relating to the software. EXCEPT AS EXPRESSLY SET FORTH IN THE APPLICABLE ALTERA PROGRAM LICENSE SUBSCRIPTION AGREEMENT UNDER WHICH THIS SOFTWARE WAS PROVDED TO YOU, ALTERA AND THIRD-PARTY LICENSORS DISCLAIM ALL WARRANTIES WITH RESPECT TO THE USE OF SUCH THIRD-PARTY SOFTWARE CODE OR DOCUMENTATION IN THE SOFTWARE, INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT. For more information, including the latest available version of specific third-party software products, refer to the documentation for the software in question.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, file names, file name extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: <*file name*>, <*project name*>**.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
(feet)	The feet direct you to more information on a particular topic.

Quartus II Handbook Version 9.0

Volume 3: Verification

101 Innovation Drive
San Jose, CA 95134
www.altera.com

QII5V3-9.0

Copyright © 2009 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

NSAI

I.S. EN ISO 9001

Contents

Chapter Revision Dates xxv

Section I. Simulation

Chapter 1. Quartus II Simulator

Introduction 1-1
Simulation Flow 1-1
Functional Simulation 1-3
Timing Simulation 1-3
Timing Simulation Using Fast Timing Model Simulation 1-4
Waveform Editor 1-4
Creating .vwf Files 1-4
Count Value 1-8
Clock 1-8
Arbitrary Value 1-8
Random Value 1-9
Generating a Testbench 1-9
Grid Size 1-9
Time Bars 1-9
Stretch or Compress a Waveform Interval 1-10
End Time 1-10
Arrange Group or Bus in LSB or MSB Order 1-11
Simulator Settings 1-12
Simulation Verification Options 1-13
Simulation Output Files Options 1-15
Simulation Report 1-15
Simulation Waveform 1-15
Simulating Bidirectional Pin 1-16
Logical Memories Report 1-16
Simulation Coverage Reports 1-17
Comparing Two Waveforms 1-17
Debugging with the Quartus II Simulator 1-18
Breakpoints 1-18
Updating Memory Content 1-18
Last Simulation Vector Outputs 1-19
Conventional Debugging Process 1-19
Accessing Internal Signals for Simulation 1-19
Scripting Support 1-20
Conclusion 1-21
Referenced Documents 1-21
Document Revision History 1-21

Chapter 2. Mentor Graphics ModelSim Support

Introduction 2-1
Background 2-1
Software Compatibility 2-3
Altera Design Flow with ModelSim-Altera or ModelSim Software 2-3
Simulation Libraries 2-4

Pre-Compiled Simulation Libraries in the ModelSim-Altera Software 2-5
RTL Functional Simulation Libraries 2-6
Gate-Level Simulation Libraries 2-6
Simulation Library Files in the Quartus II Software 2-9
RTL Functional Simulation Library Files 2-9
Gate-Level Simulation Library Files 2-10
Simulation Netlist Files 2-12
Generate Post-Synthesis Simulation Netlist Files 2-13
Generate Gate-Level Timing Simulation Netlist Files 2-14
Generating Timing Simulation Netlist Files with Different Timing Models 2-15
Disable Timing Violation on Registers 2-16
Compile Libraries Using the EDA Simulation Library Compiler 2-17
Run the EDA Simulation Library Compiler through the GUI 2-17
Run EDA Simulation Library Compiler In Command Line 2-18
Perform Simulation Using the ModelSim-Altera Software 2-18
Simulating the VHDL Designs through the GUI 2-18
Perform RTL Functional Simulation 2-18
Perform Post-Synthesis Simulation 2-20
Perform Gate-Level Simulation 2-22
Simulating Verilog HDL Designs through the GUI 2-24
Perform RTL Functional Simulation 2-24
Perform Post-Synthesis Simulation 2-26
Perform Gate-Level Simulation 2-28
Simulating the VHDL Designs at the Command Line 2-30
Perform RTL Functional Simulation 2-30
Perform Post-Synthesis Simulation 2-31
Perform Gate-Level Simulation 2-32
Simulating the Verilog HDL Designs at the Command Line 2-34
Perform RTL Functional Simulation 2-34
Perform Post-Synthesis Simulation 2-35
Perform Gate-Level Simulation 2-36
Perform Simulation Using the ModelSim Software 2-38
Simulating the VHDL Designs Using the GUI 2-38
Perform RTL Functional Simulation 2-38
Perform Post-Synthesis Simulation 2-40
Perform Gate-Level Simulation 2-43
Simulating the Verilog HDL Designs Using the GUI 2-45
Perform RTL Functional Simulation 2-45
Perform Post-Synthesis Simulation 2-48
Perform Gate-Level Simulation 2-50
Simulating the VHDL Designs at the Command Line 2-52
Perform RTL Functional Simulation 2-53
Perform Post-Synthesis Simulation 2-55
Perform Gate-Level Simulation 2-56
Simulating the Verilog HDL Designs at the Command Line 2-58
Perform RTL Functional Simulation 2-58
Perform Post-Synthesis Simulation 2-60
Perform Gate-Level Simulation 2-62
Simulating Designs that Include Transceivers 2-64
Stratix GX RTL Functional Simulation 2-64
Perform RTL Functional Simulation for Stratix GX in VHDL 2-64
Perform RTL Functional Simulation for Stratix GX in Verilog HDL 2-65
Stratix GX Gate-Level Timing Simulation 2-65
Perform Gate-Level Timing Simulation for Stratix GX in VHDL 2-65

Perform Gate-Level Timing Simulation for Stratix GX in Verilog HDL 2-66
Stratix II GX RTL Functional Simulation 2-66
Perform RTL Functional Simulation for Stratix II GX in VHDL 2-67
Perform RTL Functional Simulation for Stratix II GX in Verilog HDL 2-68
Stratix II GX Gate-Level Timing Simulation 2-68
Perform Gate-Level Timing Simulation for Stratix II GX in VHDL 2-69
Perform Gate-Level Timing Simulation for Stratix II GX in Verilog HDL 2-69
Transport Delays 2-70
+transport_path_delays 2-70
+transport_int_delays 2-70
Using the NativeLink Feature with ModelSim-Altera or ModelSim Software 2-70
Setting Up NativeLink 2-70
Perform an RTL Simulation Using NativeLink 2-71
Perform a Gate-Level Simulation Using NativeLink 2-73
Setting Up a Testbench 2-75
Creating a Testbench 2-76
Generate Simulation Script from EDA Netlist Writer 2-77
ModelSim Error Message Verification 2-77
Generating a Timing VCD File for PowerPlay 2-78
Viewing a Waveform from a .wlf File 2-78
Scripting Support 2-79
Generating a Post-Synthesis Simulation Netlist for ModelSim 2-79
Tcl Commands 2-79
Command Prompt 2-79
Generating a Gate-Level Timing Simulation Netlist for ModelSim 2-80
Tcl Commands 2-80
Command Line 2-80
Software Licensing and Licensing Setup in ModelSim-Altera Subscription Edition 2-80
LM_LICENSE_FILE Variable 2-81
Conclusion 2-81
Referenced Documents 2-81
Document Revision History 2-82

Chapter 3. Synopsys VCS and VCS-MX Support

Introduction 3-1
Software Requirements 3-1
Using the VCS or VCS-MX Software in the Quartus II Design Flow 3-2
Compile Libraries Using the EDA Simulation Library Compiler 3-4
Using the EDA Simulation Library Compiler GUI 3-4
Verilog HDL example 3-6
VHDL example 3-6
Using the EDA Simulation Library Compiler in the Command Line 3-7
RTL Functional Simulations 3-7
RTL Functional Simulation (Verilog HDL Designs) 3-8
RTL Functional Simulation (VHDL Designs) 3-9
Post-Synthesis Simulation 3-9
Generating a Post-Synthesis Simulation Netlist 3-9
Post-Synthesis Simulations (Verilog HDL) 3-13
Post-Synthesis Simulations (VHDL) 3-13
Gate-Level Timing Simulation 3-14
Generating a Gate-Level Timing Simulation Netlist 3-14
Generating a Timing Netlist with Different Timing Models 3-15
Gate-Level Timing Simulations (Verilog HDL) 3-16
Gate-Level Timing Simulations (VHDL) 3-16

Disable Timing Violation on Registers . . . 3-17
Perform Timing Simulation Using the Post-Synthesis Netlist . . . 3-17
Common VCS and VCS-MX Software Compiler Options . . . 3-17
Using VirSim . . . 3-18
Using DVE . . . 3-18
Debugging Support Command-Line Interface . . . 3-19
Simulating Designs that Include Transceivers . . . 3-19
Stratix GX RTL Functional Simulation . . . 3-19
Compiling Library Files for Functional Stratix GX Simulation in Verilog HDL . . . 3-19
Stratix GX Gate-Level Timing Simulation . . . 3-19
Compiling Library Files for Timing Stratix GX Simulation in Verilog HDL . . . 3-20
Stratix II GX RTL Functional Simulation . . . 3-20
Compiling Library Files for Stratix II GX RTL Functional Simulation in Verilog HDL . . . 3-21
Stratix II GX Gate-Level Timing Simulation . . . 3-21
Compiling Library Files for Stratix II GX Gate-Level Timing Simulation in Verilog HDL . . . 3-22
Transport Delays . . . 3-22
+transport_path_delays . . . 3-22
+transport_int_delays . . . 3-22
Using NativeLink with the VCS or VCS-MX Software . . . 3-22
Setting Up the EDA Tool Executable Location . . . 3-22
Performing an RTL Simulation Using NativeLink . . . 3-23
Performing a Gate-Level Timing Simulation Using NativeLink . . . 3-24
Setting Up a Testbench . . . 3-26
Creating a Testbench . . . 3-27
Generating a Simulation Script from the EDA Netlist Writer . . . 3-27
Generating a Timing .vcd File for PowerPlay . . . 3-28
Viewing a Waveform from a .vpd or .vcd File . . . 3-28
Scripting Support . . . 3-29
Generating a Post-Synthesis Simulation Netlist for VCS . . . 3-29
Tcl Commands . . . 3-29
Command Prompt . . . 3-30
Generating a Gate-Level Timing Simulation Netlist for VCS . . . 3-30
Tcl Commands . . . 3-30
Command Prompt . . . 3-30
Conclusion . . . 3-30
Referenced Documents . . . 3-30
Document Revision History . . . 3-31

Chapter 4. Cadence NC-Sim Support

Introduction . . . 4-1
Software Requirements . . . 4-1
Simulation Flow Overview . . . 4-2
Operation Modes . . . 4-4
Quartus II Software and NC Simulation Flow Overview . . . 4-4
Compile Libraries Using the EDA Simulation Library Compiler . . . 4-5
Run the EDA Simulation Library Compiler through the GUI . . . 4-6
Run EDA Simulation Library Compiler In Command Line . . . 4-6
RTL Functional Simulation . . . 4-7
Create Libraries . . . 4-7
Basic Library Setup . . . 4-7
LPM Functions, Altera Megafunctions, and Altera Primitive Library Setup . . . 4-9
Megafunctions Requiring Atom Libraries . . . 4-10
Compile Source Code and Testbenches . . . 4-11
Compilation in Command-Line Mode . . . 4-11

Compilation in GUI Mode . . . 4-11
Elaborate Your Design . . . 4-12
Elaboration in Command-Line Mode . . . 4-13
Elaboration in GUI Mode . . . 4-13
Add Signals to View . . . 4-13
Adding Signals in Command-Line Mode . . . 4-14
Adding Signals in GUI Mode . . . 4-14
Simulate the Design . . . 4-16
RTL Functional Simulation in Command-Line Mode . . . 4-16
RTL Functional Simulation in GUI Mode . . . 4-17
Post-Synthesis Simulation . . . 4-17
Quartus II Simulation Output Files . . . 4-17
Create Libraries . . . 4-18
Compile Project Files and Libraries . . . 4-18
Elaborate Your Design . . . 4-18
Add Signals to the View . . . 4-18
Simulate Your Design . . . 4-18
Gate-Level Timing Simulation . . . 4-19
Generating a Gate-Level Timing Simulation Netlist . . . 4-19
Generating a Timing Netlist with Different Timing Models . . . 4-19
Disable Timing Violation on Registers . . . 4-21
Perform Timing Simulation Using Post-Synthesis Netlist . . . 4-21
Quartus II Timing Simulation Libraries . . . 4-22
Create Libraries . . . 4-22
Compile the Project Files and Libraries . . . 4-22
Elaborate Your Design . . . 4-22
Compiling the Standard Delay Output File (VHDL Only) in Command-Line Mode . . . 4-23
Compiling the Standard Delay Output File (VHDL Only) in GUI Mode . . . 4-23
Add Signals to View . . . 4-24
Simulate Your Design . . . 4-24
Simulating Designs that Include Transceivers . . . 4-24
Stratix GX RTL Functional Simulation . . . 4-24
Compiling Library Files for Functional Stratix GX Simulation in VHDL . . . 4-25
Compiling Library Files for Functional Stratix GX Simulation in Verilog HDL . . . 4-25
Stratix GX Gate-Level Timing Simulation . . . 4-25
Compiling Library Files for Timing Stratix GX Simulation in VHDL . . . 4-25
Compiling Library Files for Timing Stratix GX Simulation in Verilog HDL . . . 4-26
Stratix II GX RTL Functional Simulation . . . 4-26
Compiling Library Files for Functional Stratix II GX Simulation in VHDL . . . 4-27
Compiling Library Files for Functional Stratix II GX Simulation in Verilog HDL . . . 4-28
Stratix II GX Gate-Level Timing Simulation . . . 4-28
Compiling Library Files for Timing Stratix II GX Simulation in VHDL . . . 4-28
Compiling Library Files for Timing Stratix II GX Simulation in Verilog HDL . . . 4-28
Pulse Reject Delays . . . 4-29
-PULSE_R . . . 4-29
-PULSE_INT_R . . . 4-29
Using the NativeLink Feature with NC-Sim . . . 4-29
Setting Up NativeLink . . . 4-29
Performing an RTL Simulation Using NativeLink . . . 4-30
Performing a Gate-Level Simulation Using NativeLink . . . 4-32
Setting Up a Testbench . . . 4-33
Creating a Testbench . . . 4-34
Generate Simulation Script from EDA Netlist Writer . . . 4-35
Generating a Timing VCD File for PowerPlay . . . 4-35

Viewing a Waveform from a .trn File 4-36
Scripting Support 4-37
Generate NC-Sim Simulation Output Files 4-37
Tcl Commands 4-37
Command Prompt 4-38
Conclusion 4-38
Referenced Documents 4-38
Document Revision History 4-39

Chapter 5. Aldec Active-HDL Support

Introduction 5-1
Software Compatibility 5-1
Using Active-HDL Software in Quartus II Design Flows 5-2
Simulation Libraries 5-3
Simulation Library Files in the Quartus II Software 5-3
RTL Functional Simulation Library Files 5-4
Gate-Level Timing Simulation Library Files 5-4
Simulation Netlist Files 5-6
Generate Post-Synthesis Simulation Netlist Files 5-7
Generate Gate-Level Timing Simulation Netlist Files 5-8
Generating Different Timing Models 5-8
Disable Timing Violation on Registers 5-10
Compile Libraries Using the EDA Simulation Library Compiler 5-10
Run the EDA Simulation Library Compiler through the GUI 5-11
Run EDA Simulation Library Compiler In Command Line 5-11
Perform Simulation Using the Active-HDL Software (GUI Mode) 5-12
Simulating VHDL Designs 5-12
Perform RTL Functional Simulation 5-12
Perform Post-Synthesis Simulation 5-14
Perform Gate-Level Timing Simulation 5-17
Simulating Verilog HDL Designs 5-20
Perform RTL Functional Simulation 5-20
Perform Post-Synthesis Simulation 5-22
Perform Gate-Level Timing Simulation 5-25
Perform Simulation Using the Active-HDL Software (Batch Mode) 5-28
Simulating the VHDL Designs 5-28
Perform RTL Functional Simulation 5-29
Perform Post-Synthesis Simulation 5-30
Perform Gate-Level Timing Simulation 5-32
Simulating the Verilog HDL Designs 5-34
Perform RTL Functional Simulation 5-34
Perform Post-Synthesis Simulation 5-35
Perform Gate-Level Timing Simulation 5-37
Simulating Designs that Include Transceivers 5-39
Stratix GX RTL Functional Simulation 5-39
Performing RTL Functional Simulation for Stratix GX in VHDL 5-39
Performing Functional Simulation for Stratix GX in Verilog HDL 5-40
Stratix GX Gate-Level Timing Simulation 5-40
Performing Timing Simulation for Stratix GX in VHDL 5-40
Performing Gate-Level Timing Simulation for Stratix GX in Verilog HDL 5-41
Stratix II GX RTL Functional Simulation 5-41
Performing RTL Functional Simulation for Stratix II GX in VHDL 5-42
Performing RTL Functional Simulation for Stratix II GX in Verilog HDL 5-43
Stratix II GX Gate-Level Timing Simulation 5-43

Performing Timing Simulation for Stratix II GX in VHDL 5-43
Performing Timing Simulation for Stratix II GX in Verilog HDL 5-44
Transport Delays 5-45
Using the NativeLink Feature in Active-HDL Software 5-45
Setting Up NativeLink 5-45
Performing an RTL Simulation Using NativeLink 5-46
Performing a Gate-Level Timing Simulation Using NativeLink 5-48
Setting Up a Testbench 5-49
Creating a Testbench 5-50
Generate Simulation Script from EDA Netlist Writer 5-51
Generating VCD Files for PowerPlay 5-51
Scripting Support 5-52
Generating a Post-Synthesis Simulation Netlist for Active-HDL 5-52
Tcl Commands 5-52
Command Prompt 5-52
Generating a Gate-Level Timing Simulation Netlist for Active-HDL 5-53
Tcl Commands 5-53
Command Line 5-53
Conclusion 5-53
Referenced Documents 5-53
Document Revision History 5-53

Chapter 6. Simulating Altera IP in Third-Party Simulation Tools

Introduction 6-1
IP Functional Simulation Flow 6-1
Verilog HDL and VHDL IPFS Models 6-2
Instantiate the IP in Your Design 6-3
Perform Simulation 6-3
Simulating Altera IP Using the Quartus II NativeLink Feature 6-4
Set up a Quartus II Project 6-5
Select the Third-Party Simulation Tool 6-5
Specify the Path for the Third-Party Simulator 6-5
Specify the Testbench Settings 6-6
Analyze and Elaborate the Quartus II Project 6-6
Run RTL Functional Simulation 6-6
Simulating Altera IP Without the Quartus II NativeLink Feature 6-7
Design Language Examples 6-9
Verilog HDL Example: Simulating the IPFS Model in the ModelSim Software 6-9
VHDL Example: Simulating the IPFS Model in the ModelSim Software 6-11
NC-VHDL Example: Simulating the IPFS Model in the NC-VHDL Software 6-12
Verilog HDL Example: Simulating Your IPFS Model in VCS 6-13
Single-Step Process 6-13
Two-Step Process (Compilation and Simulation) 6-13
Conclusion 6-13
Referenced Documents 6-13
Document Revision History 6-13

Section II. Timing Analysis

Chapter 7. The Quartus II TimeQuest Timing Analyzer

Introduction 7-1
Getting Started with the Quartus II TimeQuest Timing Analyzer 7-2
Setting Up the Quartus II TimeQuest Timing Analyzer 7-2

Compilation Flow with the Quartus II TimeQuest Timing Analyzer Guidelines . . . 7-2
Running the Quartus II TimeQuest Timing Analyzer . . . 7-4
Directly from the Quartus II Software . . . 7-4
Stand-Alone Mode . . . 7-4
Command-Line Mode . . . 7-4
Timing Analysis Overview . . . 7-6
Clock Analysis . . . 7-10
Clock Setup Check . . . 7-10
Clock Hold Check . . . 7-11
Recovery and Removal . . . 7-13
Multicycle Paths . . . 7-14
Metastability . . . 7-15
Common Clock Path Pessimism . . . 7-16
Clock-As-Data . . . 7-18
The Quartus II TimeQuest Timing Analyzer Flow Guidelines . . . 7-19
Create a Timing Netlist . . . 7-20
Read the Synopsys Design Constraints File . . . 7-20
Update Timing Netlist . . . 7-20
Generate Timing Reports . . . 7-20
Collections . . . 7-21
Application Examples . . . 7-22
SDC Constraint Files . . . 7-22
Fitter and Timing Analysis with SDC Files . . . 7-23
Specifying SDC Files for Place-and-Route . . . 7-23
Specifying SDC Files for Static Timing Analysis . . . 7-23
Synopsys Design Constraints File Precedence . . . 7-24
Clock Specification . . . 7-24
Clocks . . . 7-24
Generated Clocks . . . 7-26
Virtual Clocks . . . 7-28
Multi-Frequency Clocks . . . 7-29
Automatic Clock Detection . . . 7-30
Derive PLL Clocks . . . 7-30
Default Clock Constraints . . . 7-33
Clock Groups . . . 7-33
Clock Effect Characteristics . . . 7-35
Clock Latency . . . 7-35
Clock Uncertainty . . . 7-36
Derive Clock Uncertainty . . . 7-37
Intra-Clock Transfers . . . 7-38
Inter-Clock Transfers . . . 7-38
I/O Interface Clock Transfers . . . 7-38
I/O Specifications . . . 7-39
Input and Output Delay . . . 7-39
Set Input Delay . . . 7-39
Set Output Delay . . . 7-41
Delay and Skew Specifications . . . 7-42
set_net_delay . . . 7-42
set_max_skew . . . 7-43
Timing Exceptions . . . 7-44
Precedence . . . 7-44
False Path . . . 7-44
Minimum Delay . . . 7-45
Maximum Delay . . . 7-46

Multicycle Path . . . 7-47
Annotated Delay . . . 7-49
Application Examples . . . 7-50
Constraint and Exception Removal . . . 7-51
Timing Reports . . . 7-51
report_timing . . . 7-52
report_exceptions . . . 7-55
report_metastability . . . 7-57
report_clock_transfers . . . 7-57
report_clocks . . . 7-58
report_min_pulse_width . . . 7-59
report_net_timing . . . 7-60
report_sdc . . . 7-61
report_ucp . . . 7-61
report_bottleneck . . . 7-62
report_datasheet . . . 7-64
report_rskm . . . 7-64
report_tccs . . . 7-65
report_partitions . . . 7-66
report_path . . . 7-66
report_net_delay . . . 7-68
report_max_skew . . . 7-69
check_timing . . . 7-70
report_clock_fmax_summary . . . 7-72
create_timing_summary . . . 7-73
Timing Analysis Features . . . 7-74
Multi-Corner Analysis . . . 7-74
Advanced I/O Timing and Board Trace Model Assignments . . . 7-76
Wildcard Assignments and Collections . . . 7-76
Resetting a Design . . . 7-78
Cross-Probing . . . 7-78
locate . . . 7-78
The TimeQuest Timing Analyzer GUI . . . 7-79
The Quartus II Software Interface and Options . . . 7-80
View Pane . . . 7-81
View Pane: Splitting . . . 7-81
View Pane: Removing Split Windows . . . 7-82
Tasks Pane . . . 7-83
Opening a Project and Writing a Synopsys Design Constraints File . . . 7-83
Netlist Setup Folder . . . 7-83
Reports Folder . . . 7-84
Macros Folder . . . 7-84
Console Pane . . . 7-85
Report Pane . . . 7-85
Constraints . . . 7-85
Name Finder . . . 7-87
Target Pane . . . 7-88
SDC Editor . . . 7-89
Conclusion . . . 7-89
Referenced Documents . . . 7-89
Document Revision History . . . 7-90

Chapter 8. Best Practices for the Quartus II TimeQuest Timing Analyzer
Introduction . . . 8-1

Clock Requirements 8-1
Base Clocks 8-2
Derived Clocks 8-2
Virtual Clocks 8-2
I/O Requirements 8-4
Input Requirements 8-4
Output Requirements 8-4
Exceptions 8-5
False Paths 8-5
Minimum and Maximum Delays 8-6
Multicycles 8-6
Conclusion 8-7
Referenced Documents 8-7
Document Revision History 8-7

Chapter 9. Switching to the Quartus II TimeQuest Timing Analyzer
Introduction 9-1
Benefits of Switching to the Quartus II TimeQuest Timing Analyzer 9-1
Chapter Contents 9-1
Switching to the Quartus II TimeQuest Timing Analyzer 9-2
Compile Your Design 9-2
Create an .sdc File 9-2
Conversion Utility 9-3
Perform Timing Analysis with the Quartus II TimeQuest Timing Analyzer 9-3
Run the Quartus II TimeQuest Timing Analyzer 9-3
Set the Default Timing Analyzer 9-4
Differences Between Quartus II TimeQuest and Quartus II Classic Timing Analyzers 9-4
Terminology 9-5
Netlist 9-5
Collections 9-6
Constraints 9-6
Constraint Files 9-6
Constraint Entry 9-7
Constraint File Priority 9-8
Constraint Priority 9-10
Ambiguous Constraints 9-10
Clocks 9-11
Related and Unrelated Clocks 9-11
Clock Offset 9-12
Clock Latency 9-13
Offset and Latency Example 9-13
Clock Uncertainty 9-14
Derived and Generated Clocks 9-15
Automatic Clock Detection 9-16
Hold Relationship 9-18
Clock Objects 9-19
Hold Multicycle 9-20
Fitter Behavior 9-22
Fitter Performance 9-22
Reporting 9-22
Paths and Pairs 9-22
Default Reports 9-23
Netlist Names 9-23
Non-Integer Clock Periods 9-24

Other Features . . . 9-24
Scripting API . . . 9-25
Timing Assignment Conversion . . . 9-26
Setup Relationship . . . 9-27
Hold Relationship . . . 9-27
Clock Latency . . . 9-27
Clock Uncertainty . . . 9-28
Inverted Clock . . . 9-28
Not a Clock . . . 9-28
Default Required f_{MAX} Assignment . . . 9-29
Virtual Clock Reference . . . 9-29
Clock Settings . . . 9-30
Multicycle . . . 9-30
Clock Enable Multicycle . . . 9-30
I/O Constraints . . . 9-31
Input and Output Delay . . . 9-31
t_{SU} Requirement . . . 9-32
t_H Requirement . . . 9-34
t_{CO} Requirement . . . 9-36
Minimum t_{CO} Requirement . . . 9-38
t_{PD} Requirement . . . 9-40
Minimum t_{PD} Requirement . . . 9-41
Cut Timing Path . . . 9-41
Maximum Delay . . . 9-41
Minimum Delay . . . 9-42
Maximum Clock Arrival Skew . . . 9-42
Maximum Data Arrival Skew . . . 9-42
Constraining Skew on an Output Bus . . . 9-42
Conversion Utility . . . 9-44
Unsupported Global Assignments . . . 9-44
Recommended Global Assignments . . . 9-45
Clock Conversion . . . 9-46
Instance Assignment Conversion . . . 9-47
PLL Phase Shift Conversion . . . 9-48
t_{CO} Requirement Conversion . . . 9-49
Entity-Specific Assignments . . . 9-50
Paths Between Unrelated Clock Domains . . . 9-50
Unsupported Instance Assignments . . . 9-50
Reviewing Conversion Results . . . 9-51
Warning Messages . . . 9-51
Clocks . . . 9-52
Clock Transfers . . . 9-53
Path Details . . . 9-53
Unconstrained Paths . . . 9-53
Bus Names . . . 9-53
Other . . . 9-53
Re-Running the Conversion Utility . . . 9-54
Notes . . . 9-54
Output Pin Load Assignments . . . 9-54
Constraint Target Types . . . 9-54
DDR Constraints with the DDR Timing Wizard . . . 9-54
HardCopy Stratix Device Handoff . . . 9-55
Unsupported SDC Features . . . 9-55
Constraint Passing . . . 9-55

Optimization 9-55
Clock Network Delay Reporting 9-55
PowerPlay Power Analysis 9-55
Project Management 9-56
Conversion Utility 9-56
t_{PD} and Minimum t_{PD} Requirement Conversion 9-56
Referenced Documents 9-56
Document Revision History 9-57

Chapter 10. Quartus II Classic Timing Analyzer

Introduction 10-1
Timing Analysis Tool Setup 10-2
Static Timing Analysis Overview 10-2
Clock Analysis 10-4
Clock Setup Check 10-4
Clock Hold Check 10-5
Multicycle Paths 10-6
Clock Settings 10-7
Individual Clock Settings 10-8
Default Clock Settings 10-8
Clock Types 10-8
Base Clocks 10-8
Derived Clocks 10-8
Undefined Clocks 10-9
PLL Clocks 10-9
Clock Uncertainty 10-10
Clock Latency 10-11
Timing Exceptions 10-13
Multicycle 10-13
Destination Multicycle Setup Exception 10-13
Destination Multicycle Hold Exception 10-14
Source Multicycle Setup Exception 10-14
Source Multicycle Hold Exception 10-15
Default Hold Multicycle 10-16
Clock Enable Multicycle 10-16
Setup Relationship and Hold Relationship 10-18
Maximum Delay and Minimum Delay 10-19
False Paths 10-20
I/O Analysis 10-21
External Input Delay and Output Delay Assignments 10-21
Input Delay Assignment 10-21
Output Delay Assignment 10-22
Virtual Clocks 10-24
Asynchronous Paths 10-24
Recovery and Removal 10-24
Recovery Report 10-25
Removal Report 10-26
Skew Management 10-28
Maximum Clock Arrival Skew 10-28
Maximum Data Arrival Skew 10-29
Generating Timing Analysis Reports with report_timing 10-30
Other Timing Analyzer Features 10-31
Wildcard Assignments 10-31
Assignment Groups 10-31

Fast Corner Analysis 10-32
Early Timing Estimation 10-33
Timing Constraint Checker 10-33
Latch Analysis 10-34
Timing Analysis Using the Quartus II GUI 10-34
Assignment Editor 10-34
Timing Settings 10-35
Clock Settings Dialog Box 10-35
More Timing Settings Dialog Box 10-36
Timing Reports 10-36
Advanced List Path 10-37
Early Timing Estimate 10-38
Assignment Groups 10-38
Scripting Support 10-38
Creating Clocks 10-39
Base Clocks 10-39
Derived Clocks 10-39
Clock Latency 10-39
Clock Uncertainty 10-39
Cut Timing Paths 10-40
Input Delay Assignment 10-40
Maximum and Minimum Delay 10-40
Maximum Clock Arrival Skew 10-40
Maximum Data Arrival Skew 10-41
Multicycle 10-41
Output Delay Assignment 10-41
Report Timing 10-41
Setup and Hold Relationships 10-42
Assignment Group 10-42
Virtual Clock 10-42
MAX+PLUS II Timing Analysis Methodology 10-43
f_{MAX} Relationships 10-43
Slack 10-43
I/O Timing 10-44
t_{SU} Timing 10-44
t_H Timing 10-45
t_{CO} Timing 10-45
Minimum t_{CO} (min t_{CO}) 10-46
t_{PD} Timing 10-46
Minimum t_{PD} (min t_{PD}) 10-46
The Timing Analyzer Tool 10-46
Conclusion 10-47
Referenced Documents 10-47
Document Revision History 10-48

Chapter 11. Synopsys PrimeTime Support

Introduction 11-1
Quartus II Settings for Generating the PrimeTime Software Files 11-1
Files Generated for the PrimeTime Software Environment 11-2
The Netlist 11-3
The SDO File 11-3
Generating Multiple Operating Conditions with TimeQuest 11-3
The Tcl Script 11-5
Generated File Summary 11-6

Running the PrimeTime Software 11-7
Analyzing Quartus II Projects 11-8
Other pt_shell Commands 11-8
PrimeTime Timing Reports 11-8
Sample of the PrimeTime Software Timing Report 11-8
Comparing Timing Reports from the Quartus II Classic Timing Analyzer and the PrimeTime Software 11-9
Clock Setup Relationship and Slack 11-10
Clock Hold Relationship and Slack 11-13
Input Delay and Output Delay Relationships and Slack 11-16
Static Timing Analyzer Differences 11-18
The Quartus II Classic Timing Analyzer and the PrimeTime Software 11-18
Rise/Fall Support 11-18
Minimum and Maximum Delays 11-18
Recovery/Removal Analysis 11-18
Encrypted Intellectual Property Blocks 11-19
Registered Clock Signals 11-19
Multiple Source and Destination Register Pairs 11-19
Latches 11-20
LVDS I/O 11-20
Clock Latency 11-20
Input and Output Delay Assignments 11-20
Generated Clocks Derived from Generated Clocks 11-20
The Quartus II TimeQuest Timing Analyzer and the PrimeTime Software 11-21
Encrypted Intellectual Property Blocks 11-21
Latches 11-21
LVDS I/O 11-21
The Quartus II TimeQuest Timing Analyzer SDC File and PrimeTime Compatibility 11-21
Clock and Data Paths 11-22
Inverting and Non-Inverting Propagation 11-22
Multiple Rise/Fall Numbers For a Timing Arc 11-22
Virtual Generated Clocks 11-22
Generated Clocks Derived from Generated Clocks 11-22
Conclusion 11-22
Referenced Documents 11-22
Document Revision History 11-23

Section III. Power Estimation and Analysis

Chapter 12. PowerPlay Power Analysis

Introduction 12-1
Quartus II Early Power Estimator File 12-2
PowerPlay Early Power Estimator File Generator Compilation Report 12-3
Types of Power Analyses 12-5
Factors Affecting Power Consumption 12-5
Device Selection 12-5
Environmental Conditions 12-6
Air Flow 12-6
Heat Sink and Thermal Compound 12-6
Ambient Temperature 12-6
Board Thermal Model 12-6
Design Resources 12-6
Number, Type, and Loading of I/O Pins 12-7

Number and Type of Logic Elements, Multiplier Elements, and RAM Blocks 12-7
Number and Type of Global Signals 12-7
Signal Activities 12-7
PowerPlay Power Analyzer Flow 12-8
Operating Conditions 12-8
Signal Activities Data Sources 12-9
Simulation Results 12-10
Using Simulation Files in Modular Design Flows 12-11
Complete Design Simulation 12-12
Modular Design Simulation 12-12
Multiple Simulations on the Same Entity 12-13
Overlapping Simulations 12-13
Partial Simulations 12-14
Node Name Matching Considerations 12-14
Glitch Filtering 12-14
Node and Entity Assignments 12-16
Timing Assignments to Clock Nodes 12-16
Default Toggle Rate Assignment 12-17
Vectorless Estimation 12-17
Using the PowerPlay Power Analyzer 12-17
Common Analysis Flows 12-17
Signal Activities from Full Post-Fit Netlist (Timing) Simulation 12-18
Signal Activities from RTL (Functional) Simulation, Supplemented by Vectorless Estimation . . . .
. . . . 12-18
Signal Activities from Vectorless Estimation, User-Supplied Input Pin Activities 12-18
Signal Activities from User Defaults Only 12-18
Generating a .saf or .vcd File Using the Quartus II Simulator 12-18
Generating a VCD File Using a Third-Party Simulator 12-21
Generating a VCD File from ModelSim Software 12-22
Running the PowerPlay Power Analyzer Using the Quartus II GUI 12-23
PowerPlay Power Analyzer Compilation Report 12-27
Summary 12-27
Settings 12-27
Simulation Files Read 12-27
Operating Conditions Used 12-28
Thermal Power Dissipated by Block 12-28
Thermal Power Dissipation by Block Type (Device Resource Type) 12-28
Thermal Power Dissipation by Hierarchy 12-28
Core Dynamic Thermal Power Dissipation by Clock Domain 12-28
Current Drawn from Voltage Supplies 12-28
Confidence Metric Details 12-29
Signal Activities 12-29
Messages 12-29
Specific Rules for Reporting 12-29
Scripting Support 12-29
Running the PowerPlay Power Analyzer from the Command Line 12-30
Conclusion 12-31
Referenced Documents 12-31
Document Revision History 12-31

Section IV. In-System Design Debugging

Introduction IV-1
On-Chip Debugging Ecosystem IV-1

Analysis Tools for RTL Nodes IV-3
Resource Usage IV-4
Pin Usage IV-5
Usability Enhancements IV-6
Stimulus-Capable Tools IV-8
In-System Sources and Probes IV-8
In-System Memory Content Editor IV-8
Virtual JTAG Interface Megafunction IV-9
Conclusion IV-9

Chapter 13. Quick Design Debugging Using SignalProbe

Introduction 13-1
Debugging Using the SignalProbe Feature 13-1
Reserve the SignalProbe Pins 13-2
Perform a Full Compilation 13-3
Assign a SignalProbe Source 13-3
Add Registers to the Pipeline Path to SignalProbe Pin 13-4
Perform a SignalProbe Compilation 13-5
Analyze the Results of the SignalProbe Compilation 13-5
SignalProbe ECO Flows 13-6
SignalProbe ECO Flow with Quartus II Incremental Compilation 13-6
SignalProbe ECO Flow without Quartus Incremental Compilation 13-7
Common Questions About the SignalProbe Feature 13-8
Why Did I Get the Following Error Message, “Error: There are No Enabled SignalProbes to Process”? 13-8
How Can I Retain My SignalProbe ECOs during Re-Compilation of My Design? 13-8
Why Did My SignalProbe Source Disappear in the Change Manager? 13-8
What is an ECO and Where Can I Find More Information about ECOs? 13-9
How Do I Migrate My Previous SignalProbe Assignments in the Quartus II Software Version 5.1 and Earlier to Version 6.0 and Later? 13-9
What are all the Changes for the SignalProbe Feature between the Quartus II Software Version 5.1 and Earlier, and Version 6.0 and Later? 13-9
Why Can't I Reserve a SignalProbe Pin? 13-10
Scripting Support 13-11
Make a SignalProbe Pin 13-11
Delete a SignalProbe Pin 13-11
Enable a SignalProbe Pin 13-11
Disable a SignalProbe Pin 13-11
Perform a SignalProbe Compilation 13-11
Migrate Previous SignalProbe Pins to the Quartus II Software Versions 6.0 and Later 13-12
Script Example 13-12
Using SignalProbe with the APEX Device Family 13-12
Adding SignalProbe Sources 13-12
Performing a SignalProbe Compilation 13-13
Running SignalProbe with Smart Compilation 13-13
Understanding the Results of a SignalProbe Compilation 13-14
Analyzing SignalProbe Routing Failures 13-15
SignalProbe Scripting Support for APEX Devices 13-15
Reserving SignalProbe Pins 13-16
Adding SignalProbe Sources 13-16
Assigning I/O Standards 13-16
Adding Registers for Pipelining 13-16
Run SignalProbe Automatically 13-17
Run SignalProbe Manually 13-17

Enable or Disable All SignalProbe Routing 13-17
Running SignalProbe with Smart Compilation 13-17
Allow SignalProbe to Modify Fitting Results 13-17
Conclusion 13-18
Referenced Documents 13-18
Document Revision History 13-18

Chapter 14. Design Debugging Using the SignalTap II Embedded Logic Analyzer
Introduction 14-1
Hardware and Software Requirements 14-2
Design Flow Using the SignalTap II Embedded Logic Analyzer 14-4
SignalTap II Embedded Logic Analyzer Task Flow 14-4
Add the SignalTap II Embedded Logic Analyzer to Your Design 14-5
Configure the SignalTap II Embedded Logic Analyzer 14-5
Define Trigger Conditions 14-6
Compile the Design 14-6
Program the Target Device or Devices 14-6
Run the SignalTap II Embedded Logic Analyzer 14-6
View, Analyze, and Use Captured Data 14-6
Add the SignalTap II Embedded Logic Analyzer to Your Design 14-6
Creating and Enabling a SignalTap II File 14-7
Creating a SignalTap II File 14-7
Enabling and Disabling a SignalTap II File for the Current Project 14-8
Embedding Multiple Analyzers in One FPGA 14-9
Monitoring FPGA Resources Used by the SignalTap II Embedded Logic Analyzer 14-9
Using the MegaWizard Plug-In Manager to Create Your Embedded Logic Analyzer 14-10
Creating an HDL Representation Using the MegaWizard Plug-In Manager 14-10
SignalTap II Megafunction Ports 14-13
Instantiating the SignalTap II Embedded Logic Analyzer in Your HDL 14-14
Configure the SignalTap II Embedded Logic Analyzer 14-14
Assigning an Acquisition Clock 14-14
Adding Signals to the SignalTap II File 14-15
Signal Preservation 14-17
Assigning Data Signals Using the Node Finder 14-17
Assigning Data Signals Using the Technology Map Viewer 14-18
Node List Signal Use Options 14-19
Untappable Signals 14-19
Adding Signals with a Plug-In 14-19
Adding Finite State Machine State Encoding Registers 14-20
Modifying and Restoring Mnemonic Tables for State Machines 14-22
Additional Considerations 14-22
Specifying the Sample Depth 14-23
Capturing Data to a Specific RAM Type 14-23
Choosing the Buffer Acquisition Mode 14-23
Non-Segmented Buffer 14-24
Segmented Buffer 14-24
Using the Storage Qualifier Feature 14-25
Input Port Mode 14-27
Transitional Mode 14-28
Conditional Mode 14-29
Start/Stop Mode 14-30
State-Based 14-31
Showing Data Discontinuities 14-31
Disable Storage Qualifier 14-31

Managing Multiple SignalTap II Files and Configurations 14-32
Define Triggers 14-33
Creating Basic Trigger Conditions 14-33
Creating Advanced Trigger Conditions 14-34
Examples of Advanced Triggering Expressions 14-35
Trigger Condition Flow Control 14-36
Sequential Triggering 14-36
Custom State-Based Triggering 14-38
SignalTap II Trigger Flow Description Language 14-42
State Labels 14-42
Boolean_expression 14-43
Action_list 14-43
Resource Manipulation Action 14-43
Buffer Control Action 14-44
State Transition Action 14-44
Using the State-Based Storage Qualifier Feature 14-45
Specifying the Trigger Position 14-48
Creating a Power-Up Trigger 14-49
Enabling a Power-Up Trigger 14-49
Managing and Configuring Power-Up and Runtime Trigger Conditions 14-50
Using External Triggers 14-51
Trigger In 14-51
Trigger Out 14-51
Using the Trigger Out of One Analyzer as the Trigger In of Another Analyzer 14-52
Compile the Design 14-53
Faster Compilations with Quartus II Incremental Compilation 14-53
Enabling Incremental Compilation for Your Design 14-54
Using Incremental Compilation with the SignalTap II Embedded Logic Analyzer 14-55
Preventing Changes Requiring Recompilation 14-57
Timing Preservation with the SignalTap II Embedded Logic Analyzer 14-57
Performance and Resource Considerations 14-57
Program the Target Device or Devices 14-59
Programming a Single Device 14-59
Programming Multiple Devices to Debug Multiple Designs 14-60
Run the SignalTap II Embedded Logic Analyzer 14-60
Running with a Power-Up Trigger 14-62
Running with Runtime Triggers 14-62
Performing a Force Trigger 14-62
Runtime Reconfigurable Options 14-63
SignalTap II Status Messages 14-65
View, Analyze, and Use Captured Data 14-66
Viewing Captured Data 14-66
Capturing Data Using Segmented Buffers 14-67
Creating Mnemonics for Bit Patterns 14-69
Automatic Mnemonics with a Plug-In 14-69
Locating a Node in the Design 14-69
Saving Captured Data 14-70
Converting Captured Data to Other File Formats 14-70
Creating a SignalTap II List File 14-71
Other Features 14-71
Using the SignalTap II MATLAB MEX Function to Capture Data 14-71
Using SignalTap II in a Lab Environment 14-73
Remote Debugging Using the SignalTap II Embedded Logic Analyzer 14-73
Equipment Setup 14-73

Software Setup on the Remote PC 14-73
Software Setup on the Local PC 14-74
SignalTap II Setup on the Local PC 14-75
Using the SignalTap II Embedded Logic Analyzer in Devices with Configuration Bitstream Security 14-75
Backward Compatibility with Previous Versions of Quartus II Software 14-76
SignalTap II Scripting Support 14-76
SignalTap II Command-Line Options 14-76
SignalTap II Tcl Commands 14-78
Design Example: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems 14-79
Custom Triggering Flow Application Examples 14-79
Design Example 1: Specifying a Custom Trigger Position 14-80
Design Example 2: Trigger When triggercond1 Occurs Ten Times between triggercond2 and triggercond3 14-80
Conclusion 14-81
Referenced Documents 14-81
Document Revision History 14-82

Chapter 15. In-System Debugging Using External Logic Analyzers

Introduction 15-1
Choosing a Logic Analyzer 15-1
Required Components 15-2
FPGA Device Support 15-3
Debugging Your Design Using the Logic Analyzer Interface 15-3
Creating an LAI File 15-4
Creating a New Logic Analyzer Interface File 15-5
Opening an Existing External Analyzer Interface File 15-5
Saving the External Analyzer Interface File 15-5
Configuring the Logic Analyzer Interface File Core Parameters 15-5
Mapping the Logic Analyzer Interface File Pins to Available I/O Pins 15-6
Mapping Internal Signals to the Logic Analyzer Interface Banks 15-7
Using the Node Finder 15-7
Enabling the Logic Analyzer Interface Before Compiling Your Quartus II Project 15-8
Compiling Your Quartus II Project 15-8
Programming Your FPGA Using the Logic Analyzer Interface 15-9
Using the Logic Analyzer Interface with Multiple Devices 15-10
Configuring Banks in the Logic Analyzer Interface File 15-10
Acquiring Data on Your Logic Analyzer 15-10
Advanced Features 15-11
Using the Logic Analyzer Interface with Incremental Compilation 15-11
Creating Multiple Logic Analyzer Interface Instances in One FPGA 15-11
Conclusion 15-12
Referenced Documents 15-12
Document Revision History 15-12

Chapter 16. In-System Updating of Memory and Constants

Introduction 16-1
Overview 16-1
Device Megafunction Support 16-2
Using In-System Updating of Memory and Constants with Your Design 16-2
Creating In-System Modifiable Memories and Constants 16-3
Running the In-System Memory Content Editor 16-3
Instance Manager 16-4

Editing Data Displayed in the Hex Editor . . . 16-6
Importing and Exporting Memory Files . . . 16-6
Viewing Memories and Constants in the Hex Editor . . . 16-6
Scripting Support . . . 16-8
Programming the Device Using the In-System Memory Content Editor . . . 16-8
Example: Using the In-System Memory Content Editor with the SignalTap II Embedded Logic Analyzer . . . 16-9
Conclusion . . . 16-9
Referenced Documents . . . 16-9
Document Revision History . . . 16-10

Chapter 17. Design Debugging Using In-System Sources and Probes

Introduction . . . 17-1
Overview . . . 17-1
Hardware and Software Requirements . . . 17-3
Design Flow Using In-System Sources and Probes . . . 17-3
Configuring the altsource_probe Megafunction . . . 17-4
Instantiating the altsource_probe Megafunction . . . 17-6
Compiling the Design . . . 17-6
Running the In-System Sources and Probes Editor . . . 17-7
Programming Your Device Using the JTAG Chain Configuration Window . . . 17-8
Instance Manager . . . 17-8
Sources and Probes Editor Window . . . 17-10
Reading Probe Data . . . 17-10
Writing Data . . . 17-10
Data Organization . . . 17-10
Tcl Support . . . 17-11
Design Example: Dynamic PLL Reconfiguration . . . 17-14
Conclusion . . . 17-17
Referenced Documents . . . 17-17
Document Revision History . . . 17-17

Section V. Formal Verification

Chapter 18. Cadence Encounter Conformal Support

Introduction . . . 18-1
Formal Verification Versus Simulation . . . 18-1
Formal Verification: What You Need to Know . . . 18-2
Formal Verification Design Flow . . . 18-2
Quartus II Integrated Synthesis . . . 18-2
EDA Tool Support for Quartus II Integrated Synthesis . . . 18-3
Synplify Pro . . . 18-3
EDA Tool Support for Synplify Pro . . . 18-4
RTL Coding Guidelines for Quartus II Integrated Synthesis . . . 18-4
Synthesis Directives and Attributes . . . 18-5
Stuck-at Registers . . . 18-6
ROM, LPM_DIVIDE, and Shift Register Inference . . . 18-6
RAM Inference . . . 18-7
Latch Inference . . . 18-7
Combinational Loops . . . 18-7
Finite State Machine Coding Styles . . . 18-8
Black Boxes in the Encounter Conformal Flow . . . 18-8
Tcl Command . . . 18-9

GUI . . . 18-9
Generating the Post-Fit Netlist Output File and the Encounter Conformal Setup Files . . . 18-10
The Quartus II Software Generated Files, Formal Verification Scripts, and Directories . . . 18-14
Understanding the Formal Verification Scripts for Encounter Conformal . . . 18-15
The Encounter Conformal Commands within the Quartus II Software-Generated Scripts . . . 18-15
Comparing Designs Using Encounter Conformal . . . 18-17
Running the Encounter Conformal Software from the GUI . . . 18-17
Running the Encounter Conformal Software From a System Command Prompt . . . 18-18
Known Issues and Limitations . . . 18-19
Black Box Models . . . 18-21
Conformal Dofile/Script Example . . . 18-23
Conclusion . . . 18-25
Referenced Documents . . . 18-25
Document Revision History . . . 18-25

Section VI. Device Programming

Chapter 19. Quartus II Programmer

Introduction . . . 19-1
Programming Flow . . . 19-1
Programming and Configuration Modes . . . 19-4
JTAG Mode . . . 19-4
Passive Serial Mode . . . 19-4
Active Serial Mode . . . 19-5
In-Socket Programming Mode . . . 19-5
Programmer Overview . . . 19-6
Tools Menu . . . 19-9
Hardware Setup . . . 19-11
Hardware Settings . . . 19-11
JTAG Settings . . . 19-11
Device Programming and Configuration . . . 19-12
Single Device Programming and Configuration . . . 19-12
Multi-Device Programming and Configuration . . . 19-13
Bypassing an Altera Device . . . 19-13
Bypassing a Non-Altera Device . . . 19-13
Chain Description File . . . 19-16
Design Security Key Programming . . . 19-16
Optional Programming Files . . . 19-17
Types of Programming and Configuration Files . . . 19-17
Generating Optional Programming Files . . . 19-19
Create Programming Files . . . 19-19
Convert Programming Files . . . 19-19
Generating Optional Programming or Configuration Files During Compilation . . . 19-19
Flash Loaders . . . 19-19
Parallel Flash Loader . . . 19-19
Serial Flash Loader . . . 19-20
JTAG Chain Debugger Tool . . . 19-20
JTAG Chain Integrity . . . 19-21
JTAG Chain Integrity Test . . . 19-22
IDCODE Iteration Test . . . 19-24
JTAG Chain Debugging . . . 19-25
Bypassing Devices in the Chain . . . 19-28
JTAG Chain Log . . . 19-29

Other Programming Tools 19-30
Quartus II Stand-Alone Programmer 19-30
jtagconfig Debugging Tool 19-30
Scripting Support 19-30
Conclusion 19-31
Referenced Documents 19-31
Document Revision History 19-32

Additional Information

About this Handbook Info-1
How to Contact Altera Info-1
Third-Party Software Product Information Info-1
Typographic Conventions Info-2

Chapter Revision Dates

The chapters in this book, *Quartus II Handbook Version 9.0 Volume 3: Verification*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1 Quartus II Simulator
Revised: *March 2009*
Part Number: *QII53017-9.0.0*

Chapter 2 Mentor Graphics ModelSim Support
Revised: *March 2009*
Part Number: *QII53001-9.0.0*

Chapter 3 Synopsys VCS and VCS-MX Support
Revised: *March 2009*
Part Number: *QII53002-9.0.0*

Chapter 4 Cadence NC-Sim Support
Revised: *March 2009*
Part Number: *QII53003-9.0.0*

Chapter 5 Aldec Active-HDL Support
Revised: *March 2009*
Part Number: *QII53023-9.0.0*

Chapter 6 Simulating Altera IP in Third-Party Simulation Tools
Revised: *March 2009*
Part Number: *QII53014-9.0.0*

Chapter 7 The Quartus II TimeQuest Timing Analyzer
Revised: *March 2009*
Part Number: *QII53018-9.0.0*

Chapter 8 Best Practices for the Quartus II TimeQuest Timing Analyzer
Revised: *March 2009*
Part Number: *QII53024-9.0.0*

Chapter 9 Switching to the Quartus II TimeQuest Timing Analyzer
Revised: *March 2009*
Part Number: *QII53019-9.0.0*

Chapter 10 Quartus II Classic Timing Analyzer
Revised: *March 2009*
Part Number: *QII53004-9.0.0*

Chapter 11 Synopsys PrimeTime Support
Revised: *March 2009*
Part Number: *QII53005-9.0.0*

Chapter 12 PowerPlay Power Analysis
Revised: *March 2009*
Part Number: *QII53013-9.0.0*

Chapter 13 Quick Design Debugging Using SignalProbe
Revised: *March 2009*
Part Number: *QII53008-9.0.0*

Chapter 14 Design Debugging Using the SignalTap II Embedded Logic Analyzer
Revised: *March 2009*
Part Number: *QII53009-9.0.0*

Chapter 15 In-System Debugging Using External Logic Analyzers
Revised: *March 2009*
Part Number: *QII53016-9.0.0*

Chapter 16 In-System Updating of Memory and Constants
Revised: *March 2009*
Part Number: *QII53012-9.0.0*

Chapter 17 Design Debugging Using In-System Sources and Probes
Revised: *March 2009*
Part Number: *QII53021-9.0.0*

Chapter 18 Cadence Encounter Conformal Support
Revised: *March 2009*
Part Number: *QII53011-9.0.0*

Chapter 19 Quartus II Programmer
Revised: *March 2009*
Part Number: *QII53022-9.0.0*

Section I. Simulation

As the design complexity of FPGAs continues to rise, verification engineers are finding it increasingly difficult to simulate their system-on-a-programmable-chip (SOPC) designs in a timely manner. The verification process is now the bottleneck in the FPGA design flow. You can perform functional and timing simulation of your design by using the Quartus® II Simulator. The Quartus II software also provides a wide range of features for performing simulation of designs in EDA simulation tools.

This section includes the following chapters:

- Chapter 1, Quartus II Simulator
- Chapter 2, Mentor Graphics ModelSim Support
- Chapter 3, Synopsys VCS and VCS-MX Support
- Chapter 4, Cadence NC-Sim Support
- Chapter 5, Aldec Active-HDL Support
- Chapter 6, Simulating Altera IP in Third-Party Simulation Tools

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

1. Quartus II Simulator

QII53017-9.0.0

Introduction

With today's FPGAs becoming faster and more complex, designers face challenges in validating their designs. Simulation verifies the correctness of the design, reducing board testing and debugging time.

The Altera® Quartus® II Simulator is included in the Quartus® II software to assist designers with design verification. The Quartus II Simulator has a comprehensive set of features that are covered in the following sections:

- "Simulation Flow"
- "Waveform Editor" on page 1–4
- "Simulator Settings" on page 1–12
- "Simulation Report" on page 1–15
- "Debugging with the Quartus II Simulator" on page 1–18
- "Scripting Support" on page 1–20

This chapter describes how to perform different types of simulations with the Quartus II Simulator.

The Quartus II Simulator supports the following device families:

- Arria® GX
- Stratix® III, Stratix II, Stratix, Stratix GX, Stratix II GX
- Cyclone® III, Cyclone II, Cyclone
- HardCopy® II, HardCopy
- MAX® II, MAX 3000A, MAX 7000AE, MAX 7000B, MAX 7000S
- ACEX® 1K
- APEX™ 20KC, APEX 20KE, APEX II
- FLEX® 10K, FLEX 10KA, FLEX 10KE, FLEX 6000

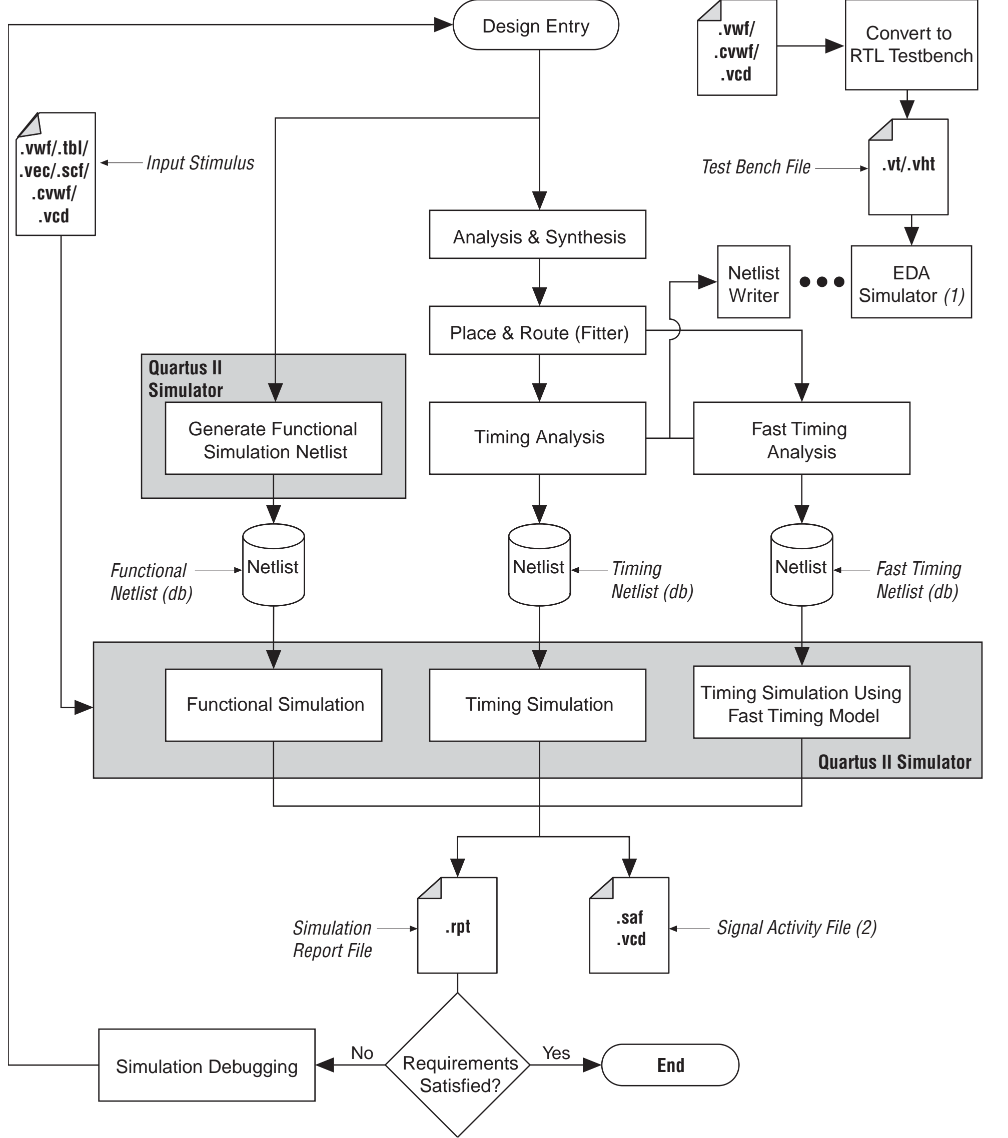
The Quartus II Simulator does not support newer devices introduced after Stratix III and Quartus II software version 8.1 and later. Use the ModelSim-Altera Edition to run simulations on designs targeting device introductions after Stratix III. For more information about the ModelSim-Altera Edition simulator, refer to the *Mentor Graphics ModelSim Support* chapter in volume 3 of the *Quartus II Handbook*.

Simulation Flow

You can perform functional and timing simulations with the Quartus II Simulator. Both types of simulation verify the correctness and behavior of your design. Functional simulations are run at the beginning of the Quartus II design flow and timing simulations are run at the end.

Figure 1–1 shows the Quartus II Simulator flow.

Figure 1–1. Simulation Flow



Notes to Figure 1–1:

(1) For more information about EDA Simulators, refer to the *Simulation* section in volume 3 of the *Quartus II Handbook*.

(2) You can use Signal Activity Files (**.saf**) or Value Change Dump Files (**.vcd**) in the PowerPlay Power Analyzer to check power resources.

As shown in Figure 1–1, your design simulation can happen at the functional level, where your design's logical behavior is verified and no timing information is used in simulation. Timing simulation can happen after your design has been compiled (synthesized and placed and routed) and after you use the timing data of your design's resources. In Timing simulation, your design's logical behavior is verified with the device's worst-case timing models. Timing simulation using the Fast Timing Model is also a type of timing simulation where best-case timing data is used.

To perform functional simulations with the Quartus II Simulator, you must first generate a functional simulation netlist. A functional netlist file is a flattened netlist extracted from the design files that does not contain timing information.

For timing simulations, you must first perform place-and-route and static timing analysis to generate a timing simulation netlist. A timing simulation netlist includes timing delays of each device atom block and the routing delays.

If you want to use third-party EDA simulation tools, you can generate a netlist using EDA Netlist Writer. You can use this netlist with your testbench files in third-party simulation tools.

For more information about third-party simulators, refer to the respective EDA Simulation chapter in the *Simulation* section in volume 3 of the *Quartus II Handbook*.

The Quartus II Simulator supports Functional, Timing, and Timing using Fast Timing Model simulations. The following sections describe how to perform these simulations.

Functional Simulation

To run a functional simulation, perform the following steps:

1. On the Processing menu, click **Generate Functional Simulation Netlist**. This flattens the functional simulation netlist extracted from the design files. The netlist does not contain timing information.
2. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
3. In the **Category** list, select **Simulator Settings**. The **Simulator Settings** page appears.
4. In the **Simulation mode** list, select **Functional**.
5. In the **Simulation input** box, specify the vector source. You must specify the vector file to run the simulation.
6. Click **OK**.
7. On the Processing menu, click **Start Simulation**.

Timing Simulation

To run a timing simulation, perform the following steps:

1. On the Processing menu, click **Start Compilation** or click the **Compilation** button on the toolbar. This flattens the design and generates an internal netlist with timing delay information annotated.
2. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
3. In the **Category** list, select **Simulator Settings**. The **Simulator Settings** page appears.
4. In the **Simulation mode** list, select **Timing**.
5. In the **Simulation input** list, specify the vector source. You must specify the vector file to run the simulation.
6. Click **OK**.

7. On the Processing menu, click **Start Simulation**.

Timing Simulation Using Fast Timing Model Simulation

To run a timing simulation using a fast timing model, perform the following steps:

1. On the Processing menu, point to **Start** and click **Start Analysis and Synthesis**.
2. On the Processing menu, point to **Start** and click **Start Fitter**.

You must perform fast timing analysis before you can perform a timing simulation using the fast timing models.

1. On the Processing menu, point to **Start** and click **Start Classic Timing Analyzer (Fast Timing Model)**.
2. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
3. In the **Category** list, select **Simulator Settings**. The **Simulator Settings** page appears.
4. In the **Simulation mode** list, select **Timing using Fast Timing Model**.
5. In the **Simulation input** box, specify the vector source. You must specify the vector file to run the simulation.
6. Click **OK**.
7. On the Processing menu, click **Start Simulation**.

Waveform Editor

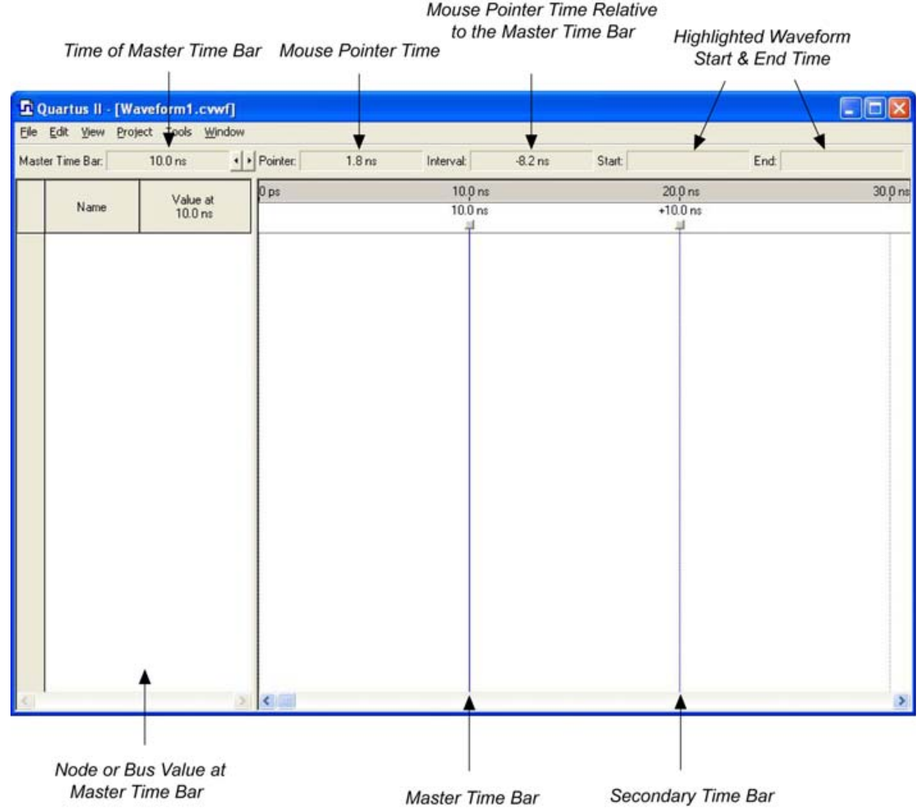
The most common input stimulus for the Quartus II Simulator are Vector Waveform Files (**.vwf** files). You can use the Quartus II Waveform Editor to generate a **.vwf** file.

Creating .vwf Files

To create a **.vwf** file, perform the following steps:

1. On the File menu, click **New**. The **New** dialog box appears.
2. Select **Vector Waveform File**.
3. Click **OK**. A blank Waveform Editor window appears (Figure 1–2).

Figure 1–2. Waveform Editor Window

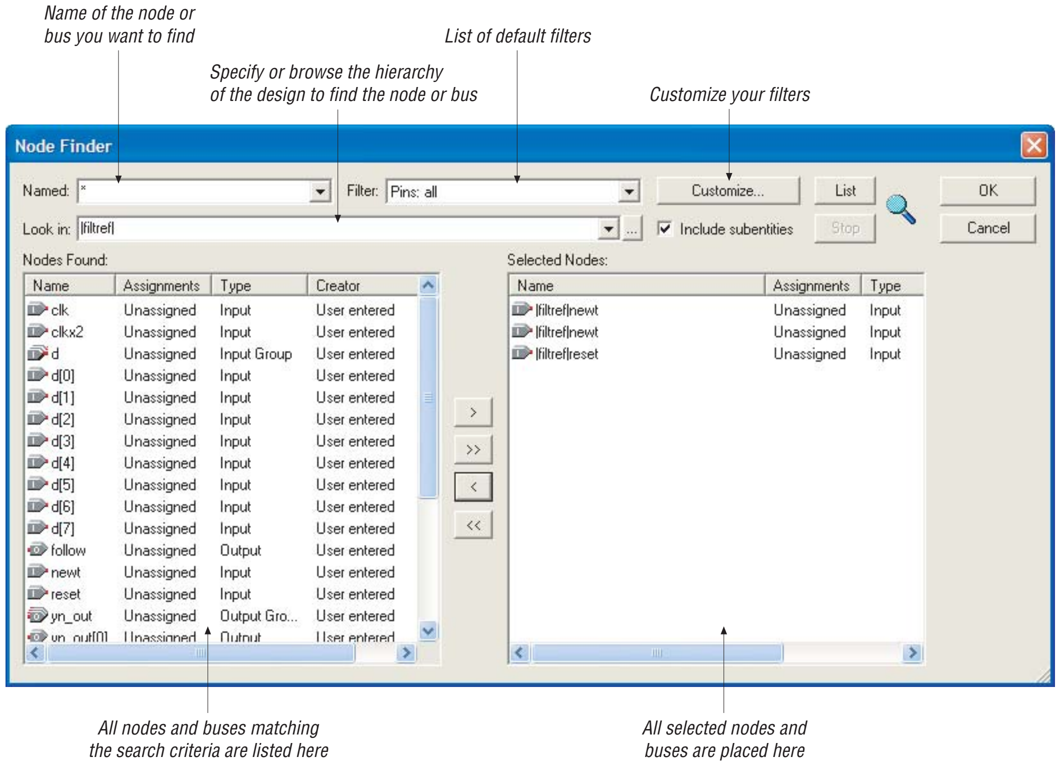


4. Add nodes and buses. To add a node or bus, on the Edit menu, click **Insert** and click **Insert Node or Bus**. The **Insert Node or Bus** dialog box appears. All nodes and buses, as well as the internal signals, are listed under **Name** in the Waveform Editor window.

You can also open the **Insert Node or Bus** dialog box by double-clicking under **Name** in the Waveform Editor.

5. You can customize the type of node or bus you want to add. If you have a large design with many nodes or buses, you may want to use the Node Finder for node or bus selection. To use the Node Finder, click **Node Finder**. The **Node Finder** dialog box appears (Figure 1–3).

Figure 1–3. Node Finder Dialog Box



You can use the Node Finder to find your nodes for simulation among all the nodes and buses in your design. Use the Node Finder to filter and add nodes to your waveform. The Node Finder is equipped with multiple default filter options. By using the correct filter in the Node Finder, you can find the internal node's name and add it to your Vector Waveform File for simulation.

Your node might not appear in the simulation waveform and might be ignored during simulation. This happens because the node has been renamed or synthesized away by the Quartus II software. To prevent this from happening, Altera recommends using the register and pin nodes to simulate your design.

Table 1–1 describes twelve of the Node Finder default filters.

Table 1–1. Filter Options (Part 1 of 2)

Filter	Description
Pins: input	Finds all input pin names in your design file(s).
Pins: output	Finds all output pin names in your design file(s).
Pins: bidirectional	Finds all bidirectional pin names in your design file(s).
Pins: virtual	Finds all virtual pin names.
Pins: all	Finds all pin names in your design file(s).
Registers: pre-synthesis	Finds all user-entered register names contained in the design after design elaboration, but before physical synthesis does any synthesis optimizations.

 © March 2009 Altera Corporation

Table 1–1. Filter Options (Part 2 of 2)

Filter	Description
Registers: post-fitting	Finds all user-entered register names in your design file(s) that survived physical synthesis and fitting.
Design Entry (all names)	Finds all user-entered names in your design file(s).
Post-Compilation	Finds all user-entered and compiler-generated names that do not have location assignments and survived fitting.
SignalTap II: pre-synthesis	Finds all internal device nodes in the pre-synthesis netlist that can be analyzed by the SignalTap® II Logic Analyzer.
SignalTap II: post-fitting	Finds all internal device nodes in the post-fitting netlist that can be analyzed by the SignalTap II Logic Analyzer.
SignalProbe	Finds all SignalProbe device nodes in the post-fitting netlist.

To customize your own filters in the Node Finder, perform the following steps:

a. Click **Customize**. The **Customize Filter** dialog box appears.

b. To configure settings, click **New**. The **New Custom Filter** dialog box appears.

c. In the **Filter name** box, type the name of the custom filter.

d. In the **Copy settings from filter** list, select the filter setting.

e. Click **OK**.

f. You can now customize your filters in the **Customize Filter** dialog box.

6. In the **Look in** box, you can view and edit the current search hierarchy path. You can type the search hierarchy path or you can browse for the hierarchy path by clicking the browse button.

   You can move up the search hierarchy by selecting hierarchical names in the **Select Hierarchy Level** dialog box. This ensures that in a large design with many signals, you can specify which hierarchy you would like to get the node from to reduce the amount of signals displayed.

7. After you have configured the filter and specified the correct hierarchy in the **Node Finder** dialog box, click **List** to display all relevant nodes or buses.

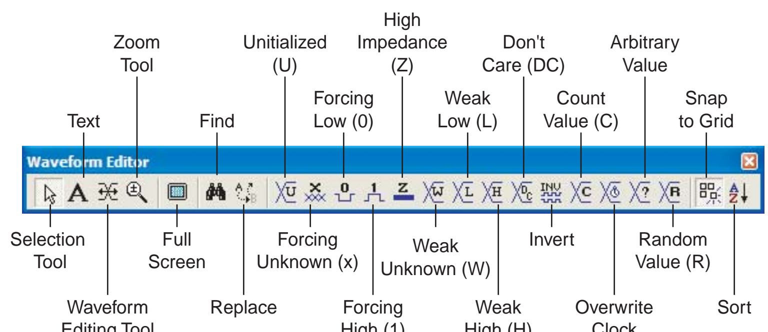
   Select any node(s) or bus(es) from the **Nodes Found** list and click **>** to include it in the waveform, or you can click **>>** to include all nodes and buses displayed in the **Nodes Found** list.

8. Click **OK**.

You can also add nodes to the Waveform Editor by dragging nodes from the Project Navigator, Netlist Viewers, or Block Diagram, and dropping them into the Waveform Editor.

9. Create a waveform for a signal. The Quartus II Waveform Editor toolbar includes some of the most common waveform settings, making waveform vector drawings easier and user friendly. Figure 1–4 shows the options available on the Waveform Editor toolbar.

Figure 1–4. Waveform Editor Toolbar



10. After you edit your waveform, save the waveform. On the File menu, click **Save As**. The **Save As** dialog box appears. Type your file name, specify the file type, and click **Save**.

Instead of using the Node Finder to insert your nodes for your **.vwf** file, you can also drag-and-drop any nodes from the Netlist Viewer to your Simulation Vector Waveform File. For more information about Netlist Viewers, refer to the *Analyzing Designs with the Quartus II Netlist Viewers* chapter in volume 1 of the *Quartus II Handbook*.

Count Value

Count Value applies a count value to a bus to increment the value of the bus by a specified time interval. Instead of manually editing the values for each node, the Count Value feature on the Waveform Editor toolbar automatically creates the counting values for buses. This feature enables you to specify a starting value for a bus, what time interval to increment, and when to stop counting. You can also configure transition occurrences while setting the count type and increment number. When you click on the **Count Value** button in the Waveform Editor toolbar, the **Count Value** dialog box appears. You can also open the **Count Value** dialog box by right-clicking the selected node, pointing to **Value**, and clicking **Count Value**.

Clock

You can use the Clock feature in the Waveform Editor toolbar to automatically generate the clock wave, rather than drawing each clock triggering pulse. To generate a clock signal with the **Clock** dialog box, click the **Overwrite Clock** button on the Waveform Editor toolbar. You can also determine the start and end time of a clock signal, whether to manually configure the period (the offset and the duty cycle), or whether to generate the clock based on a specified clock.

Arbitrary Value

Arbitrary Value allows you to overwrite a node value over the selected waveform, waveform interval, or across one or more nodes or groups. To overwrite a node value, perform the following steps:

1. Select a node or a bus and click the **Arbitrary Value** button on the Waveform Editor toolbar (Figure 1–4). The **Arbitrary Value** dialog box appears.

2. Under **Time range**, specify the start and end time you want to overwrite for the node value.
3. In the **Radix** list, select the radix type.
4. Specify the new value you want overwritten in the **Numeric or named value** box.
5. Click **OK**.

Random Value

Random Value allows you to generate random node values over the selected waveform, waveform interval, or across one or more nodes or groups.

You can generate random node values by every grid interval, every half grid interval, at random intervals, or at fixed intervals.

Generating a Testbench

You can export your **.vwf** file as a VHDL Test Bench File (**.vht**) or Verilog Test Bench File (**.vt**). This is useful when you want to use a vector waveform in different EDA tools. You must run an analysis and elaboration before you can export a waveform vector. To export a waveform vector, have your vector waveform open and perform the following steps:

1. On the File menu, click **Export**. The **Export** dialog box appears.
2. In the **Save as type** list, select **VHDL Test Bench File (*.vht)** or **Verilog Test Bench File (*.vt)**.
3. You can optionally turn on **Add self-checking code to file**. This option adds additional logic to check the results of the output and compares it to the original **.vwf** file.

You must open your project in the Quartus II software before you can export a **.vwf** file.

For more information about using the generated testbench in other EDA tools, refer to the respective EDA simulator chapter in the *Simulation* section in volume 3 of the *Quartus II Handbook*.

Grid Size

When you select portions of your waveform, the selection area snaps to time intervals specified in the **Grid Size** dialog box. You can customize the grid size in the Waveform Editor. You can change the grid size based on the clock settings or by setting the time period. To customize the grid size, on the Edit menu, click **Grid Size**.

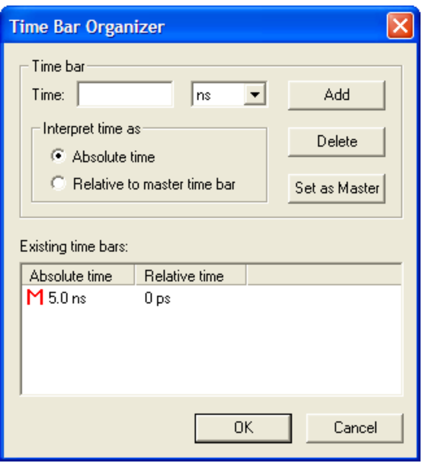
Time Bars

Add time bars in the Waveform Editor to compare edges between different signals. You can also use time bars to jump forward and backward to the next edge transition in the selected signal, and read the logic level of signals by sliding the Time Bar in your waveform. The logic level is displayed in the **Value at** column of the Waveform Editor.

The **Time Bar Organizer** dialog box enables you to create, delete, and edit a time bar, and to create a master time bar. Only one master time bar is allowed per waveform file. To use the Time Bar Organizer, on the Edit menu, point to **Time Bar** and click **Time Bar Organizer**.

Under **Existing time bars**, in the **Absolute time** column, the red **M** indicates the master time bar (Figure 1–5).

Figure 1–5. Time Bar Organizer Dialog Box



Stretch or Compress a Waveform Interval

You can stretch or compress a waveform interval in the Waveform Editor, which enables you to analyze the effects on a waveform. For example, you can check the behavior of your design at high speeds for a short interval by using the compress option to compress the waveform. You can also use this feature to delay the transition of a signal by stretching the waveform.

You have to specify the original start and end time, and the new time for the waveform you want to stretch or compress. If you want to stretch or compress all the nodes or buses, deselect all nodes and buses and set the stretch or compress feature.

To stretch or compress a waveform interval, on the Edit menu, point to **Value** and click **Stretch or Compress Waveform Interval**. The **Stretch or Compress Waveform Interval** dialog box appears.

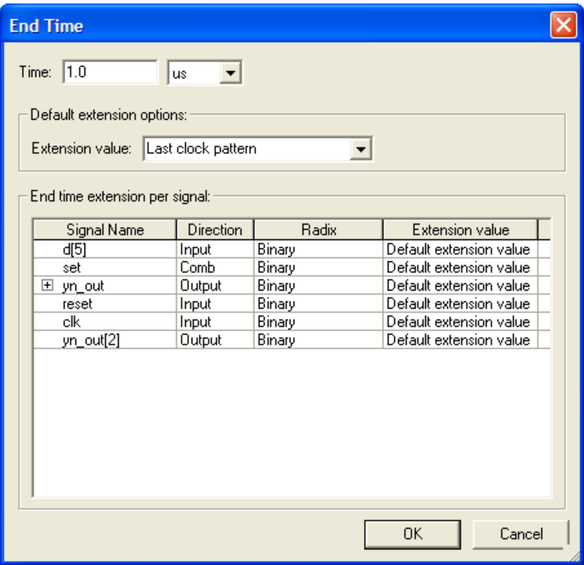
The "To time value" end time specified in the **Stretch or Compress Waveform Interval** dialog box cannot be larger than the "End Time" specified in the **Simulator Settings** page of the **Settings** dialog box. Otherwise, the Quartus II software displays a message indicating the invalid time value.

End Time

The End Time setting enables you to change the end time of the **.vwf** file. The end time represents the maximum length of time in the **.vwf** file. You can specify the end time and your preferred time unit, and have different extension values for different nodes or buses. With the waveform open, specify the end time by performing the following steps:

1. On the Edit menu, click **End Time**. The **End Time** dialog box appears (Figure 1–6).

Figure 1–6. End Time Dialog Box



2. In the **Time** box, specify the end time and select the time unit in the **Time** list.
3. Under **Default extension options**, in the **Extension value** list, select the value.
4. Under **End time extension per signal**, you can select specific extension values for each signal by clicking in the **Extension value** column.

The options in the **End time** dialog box are different settings than those under **Simulation period** in the **Settings** dialog box. Simulation period is the period that the Quartus II software simulates the stimuli. End time is the maximum length of time in the **.vwf** file. For information on the simulation period, refer to Table 1–2 on page 1–12.

Arrange Group or Bus in LSB or MSB Order

You can arrange a group or bus in LSB or MSB order. If you arrange in LSB order, the LSB is on top and the MSB is at the bottom. If you arrange in MSB order, the MSB is on top and the LSB is at bottom.

To arrange a group or bus in LSB or MSB order, perform the following steps:

1. Select the bus that you want to change the LSB or MSB order. You can also select multiple buses in the waveform editor.
2. On the Edit menu, point to **Group and Bus Bit Order** and click either **MSB on top, LSB on Bottom** to change the bus or group in MSB order, or click **LSB on top, MSB on Bottom** to change the bus or group in LSB order.

Simulator Settings

You can enhance your output, reduce debugging time, and provide better coverage before running a simulation. This section covers the different simulation modes supported by the Quartus II Simulator. Additionally, the Quartus II Simulator offers common setup features like glitch filtering, setup and hold violation detection, and simulation coverage.

To set up simulation settings, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulator Settings**. The **Simulator Settings** page appears.

Table 1–2 shows the options in the **Simulator Settings** page.

Table 1–2. Quartus II Simulator Settings (Part 1 of 2)

Settings and Options	Description
Simulation mode *(1)*	**Functional** This simulation mode uses a pre-synthesis compiler database to simulate the logical performance of a project without the timing information. This mode enables you to check the functionality of the design. All nodes and buses are preserved in this simulation because functional simulation is performed before synthesis, partitioning, or fitting. A **.vwf** file is required to perform this simulation mode. **Timing** This simulation mode uses the compiled netlist that includes timing information. With this simulation mode, you can check setup, hold violation, glitches, and simulation coverage. You can remove nodes or buses using the Quartus II Compiler when logic is optimized. This simulation mode uses the worst case timing model. **Timing using Fast Timing Model** This simulation mode is similar to timing simulation but this mode uses the best-case timing model.
Simulation input	You must include the vector file in the **Simulation input** box. You can type the name of the file or use the browse button to open the **Select File** dialog box. In the **Files of type** list, you can select Vector Waveform File (***.vwf**), Compressed Vector Waveform File (***.cvwf**), Value Change Dump File (***.vcd**), Vector Table Output File (***.tbl**), Vector Text File (***.vec**), Simulation Channel File (**.scf**), or **All Files (*.*)**. TBL files contain input vectors and output logic levels in a tabular-format list. You can generate this file using a **.vwf** file. However, if you would like to maintain, view, or update the vectors, **.vwf** files offer better visibility. **.vwf** or TBL file formats are interchangeable. You can generate TBL files from **.vwf** files and vice versa. You can create a **.vwf** with the Waveform Editor. For more information about the Waveform Editor, refer to "Waveform Editor" on page 1–4. The Quartus II software also supports MAX+PLUS® II simulation vector files, such as **.vec** and **.scf** files. A **.cvwf** file is the simplified version, non-readable format of the **.vwf** file format. This file type is in binary format and is generally smaller in file size. You can use **.cvwf** files in the Waveform Editor and simulation. A **.vcd** file is an ASCII file that contains header information, variable definitions, and the value changes for specified variables, or all variables, in a given design. The value changes for a variable are given in scalar or vector format based on the nature of the variable.

Table 1–2. Quartus II Simulator Settings (Part 2 of 2)

Settings and Options	Description
Simulation period	The simulation period determines the length of time that the simulator runs the stimuli with the maximum period being equal to the end time of a **.vwf** file. If the simulation period is configured shorter than the end time, all signals beyond the simulation period are displayed as Unknown (X). Therefore, you can also shorten the simulation period or end the simulation earlier by selecting **End Simulation at** and specifying the time and selecting the time unit. If the simulation period is configured longer than the end time, the simulation will stop at the end time. For information on the end time, refer to "End Time" on page 1–10.
Glitch filtering options	Specifies whether to enable glitch filtering for simulations. You can select one of the following options: **Auto**—The Simulator performs glitch filtering when **.saf** file generation is enabled in the **Simulation Output Files** page of the **Settings** dialog box. **Always**—The Simulator always performs glitch filtering, even if **.saf** file generation is not enabled. **Never**—The Simulator never performs glitch filtering, even if **.saf** file generation is enabled.
More Settings	If you click **More Settings**, the **More Simulator Settings** dialog box appears. The following options are available under **Existing option settings**. **Cell Delay Model Type** Specifies the type of delay model to be used for cell delays: transport or inertial. The default is transport. **Interconnect Delay Model Type** Specifies the type of delay model to be used for interconnect delays: transport or inertial. The default is transport. **Preserve fewer signal transition to reduce memory requirements** This option is effective on lower performance workstations because turning on this option flushes signal transitions from memory to disk for memory optimization.

Note to Table 1–2:

(1) The Quartus II Simulator may flag an error message if zero-time oscillation occurs in your design. Zero-time oscillation occurs when a particular output signal does not achieve a stable output value at a particular fixed time, which may be due to your design containing combinational logic path loops.

Simulation Verification Options

Table 1–3 shows the options in the **Simulation Verification** page.

Table 1–3. Quartus II Simulation Verification

Settings and Options	Description
Check outputs	**Check outputs** checks expected outputs against actual outputs in the simulation report. After turning on **Check outputs**, click the **Waveform Comparison Settings** button. The **Waveform Comparison Settings** dialog box appears. In the **Waveform Comparison Settings** dialog box, you can specify the waveform comparison time frame and the comparison options. You can also set the tolerance level for all the signals by specifying the tolerance limit in the **Default comparison timing tolerance** box. The **Maximum comparison mismatches** box is the amount of mismatches the Quartus II Simulator is allowed to accept before it stops comparing. You can also set the type of transition the comparison should trigger in the **Waveform Comparison Settings** dialog box. You can assign trigger comparisons based on **Input signal transition edges**, **All signal transition edges**, or **Selected Signal transition edges**. To customize the waveform comparison matching rules, you can also click the **Comparison Rules** button. The **Comparison Rules** dialog box appears, allowing you to customize the comparison matching rules.
Setup and hold time violation detection	This option detects setup and hold time violation. Setup time is the period required by a synchronous signal to stabilize before the arrival of a clock edge. Hold time is the time required by a synchronous signal to maintain after the same clock edge. If the **Setup and hold time violation detection** option is turned on, a warning in the Messages window appears if any setup or hold time violation is detected during the simulation. This option is only for Timing and Timing using Fast Timing Model simulation modes.
Glitch detection	Conditions occur when two or more signals toggle simultaneously and can cause glitches or unwanted short pulses. The **Glitch detection** option enables you to detect glitches and specify the time interval that defines a glitch. If two logic level transitions occur in a period shorter than the specified time period, the resulting glitch is detected and reported in the **Processing** tab of the Messages window. If you turn on the **Glitch detection** option, you can specify the acceptable glitch width. A Messages window appears when a pulse is smaller than the specified glitch width that is detected. The **Glitch detection** option is only available for Timing and Timing using Fast Timing Model simulation modes.
Simulation coverage reporting	This option reports the ratio of outputs (coverage) actually simulated to the number of outputs in the netlist and is expressed as a percentage. When you turn on the **Simulation coverage reporting** option, the **Report Settings** button is available. If you click **Report Settings**, the **Report Settings** dialog box appears. The three types of coverage reports you can select from are **Display complete 1/0 value coverage report**, **Display missing 1-value coverage report**, and **Display missing 0-value coverage report**.
Disable setup and hold time violation detection for input registers of bi-directional pins	This option enables you to disable setup and hold time violation detection in input registers of all bidirectional pins in the simulated design during Timing or Timing using Fast Timing Model simulation.

Simulation Output Files Options

Table 1–4 shows the options in the **Simulation Output Files** page.

Table 1–4. Quartus II Simulation Output Files

Setting and Options	Description
Simulation output waveform	Specify the simulation output waveform options. **Automatically add pins to simulation output waveforms** This option automatically adds all outputs that are available in the design to the waveform reports. If your design has large amounts of outputs, turning on this option ensures all outputs are monitored during simulation. **Overwrite simulation input file with simulation results** This option overwrites the vector source file with simulation results. This option is ignored when the **Check outputs** setting is turned on. This option adds the result to the vector file and generally, it can give you more visibility during the debugging process. *(1)* **Group bus channel in simulation results** This option automatically groups bus channels in the output waveform that are shown in the simulation reports. By turning off this option, all output waveforms have a node to represent each bus signal.
Signal activity output for power analysis	When you perform your simulation with the Quartus II Simulator, you can generate a **.saf** file, which is used by the PowerPlay Power Analyzer to assist you with power analysis. *(2)*, *(3)*
VCD output for power analysis	When you perform simulation with the Quartus II Simulator, you can generate a **.vcd** file, which is used by the PowerPlay Power Analyzer to assist you with power analysis. *(2)*, *(3)*

Notes to Table 1–4:

(1) A backup copy of the source vector file is saved under the **db** folder with the name <*project*>**.sim_ori.** <*vector file format type*>.

(2) Instead of using the **.saf** file or Generate **.vcd** file (***.vcd**), you can also save your output waveform as a **.vcd** file to perform power analysis.

(3) For more information about the PowerPlay Power Analyzer, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Simulation Report

Comprehensive reports are shown after the completion of each simulation. These reports are important to ensure designs meet timing and logical correctness. These simulation reports also play an important role in debugging.

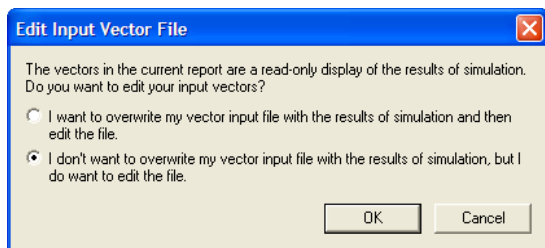
Simulation Waveform

Simulation Waveforms are part of the Simulation report. In this report, the stimuli and the results of the simulation are displayed.

You can export the simulation waveform as a VHDL Test Bench File or a Verilog Test Bench File for use in other EDA tools. You can also save a simulation as a **.vwf** file or Vector Table Output File for use with the Quartus II software.

When you try to edit the Simulation Waveform, the **Edit Input Vector File** dialog box appears, asking whether you would like to edit the vector input file with the results of the simulation or if you would like to overwrite the vector input file with other vector inputs (Figure 1–7).

Figure 1–7. Edit Input Vector File



You can overwrite your simulation input file with the simulation results so that your input vector file is updated with the resulting waveform after a simulation. For more information, refer to the **Overwrite simulation input file with simulation results** option in Table 1–2.

If you do not want to overwrite the simulation input file in every simulation run, perform the following to overwrite simulation input files with simulation results after a simulation:

On the Processing Menu, point to **Simulation Debug** and click **Overwrite Vector Inputs with Simulation Outputs**.

Simulating Bidirectional Pin

A bidirectional pin is represented in the waveform by two channels. One channel represents the input to the bidirectional pin, and the other channel represents the output from the bidirectional pin. You can enter the input channel into the waveform by using the **Node Finder** dialog box. The output channel is automatically created by the Quartus II Simulator and named <*bidir pin name*> `~result`.

Logical Memories Report

The Quartus II software writes out the contents of each memory module after simulation. Therefore, if you use memory cells in your design, you can analyze the contents of the logic memory structures in the device in the Logical Memories Report. The Logical Memories Report displays individual reports for each memory block and contains the data stored in the memory cell used at the end of simulation.

After being simulated, a memory module's contents are stored in the Logical Memories section of the simulation report file.

To view this section, perform the following steps:

1. On the Processing menu, click **Simulation Report**. The Simulation Report window appears.
2. In the report window, click the "+" next to **Logical Memories**.

Simulation Coverage Reports

The **Coverage Summary** report contains the following summary information for the simulation:

- Total toggling coverage as a percentage
- Total nodes checked in the design
- Total output ports checked
- Total output ports with complete 1/0-value coverage
- Total output ports with no 1/0-value coverage
- Total output ports with no 1-value coverage
- Total output ports with no 0-value coverage

The **Complete 1/0-Value Coverage** report lists the following information:

- Node name
- Output port name
- Output port type for output ports that toggle between 1 and 0 during the simulation

The **Missing 0-Value Coverage** report and **Missing 1-Value Coverage** report list the following information:

- Node name
- Output port name
- Output port type for output ports that do not toggle to the designated value

For more information about Simulation Coverage reports, refer to the **Simulation coverage reporting** option in Table 1–2 on page 1–12.

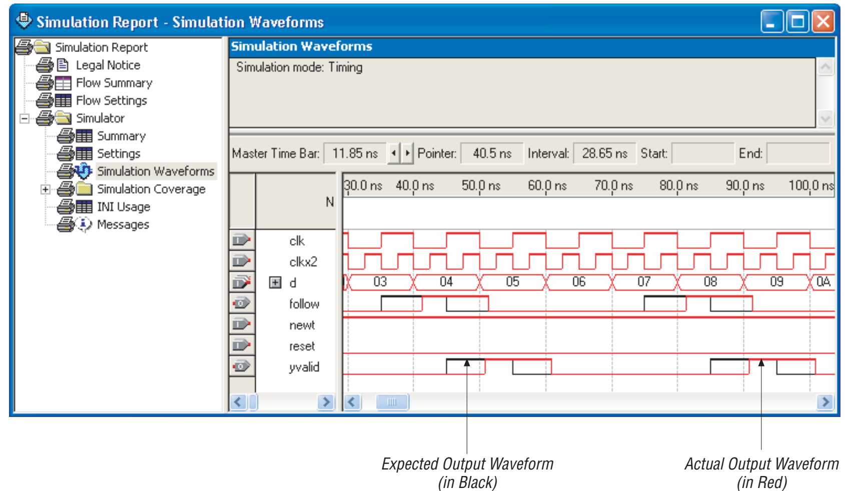
The following are individual reports and their definitions:

- **Complete 1/0 value coverage report**—Displays all the nodes or buses that toggle between 1 and 0 during simulation.
- **Missing 1-value coverage** and **Missing 0-value coverage reports**—Displays all the nodes that do not toggle to the designated value.

Comparing Two Waveforms

You can compare your simulation results against previous simulations using the compare option. To compare two waveforms in the Simulation Report, turn on the **Check outputs** option. For more information about the **Check outputs** option, refer to Table 1–2 on page 1–12. With the **Check outputs** option turned on, the two comparable waveforms are visible in black and red. The black waveforms represent the original output or the expected output, and the red waveforms represent the compared output or the actual output. Figure 1–8 shows an example of expected output waveform versus actual output waveform.

Figure 1–8. Example of Simulation Waveform from the Simulation Report When Check Output is Turned On



Debugging with the Quartus II Simulator

The Quartus II software includes tools to help with simulation debugging. This section covers some debugging tools and their use.

Breakpoints

Inserting breakpoints into the simulation process enables the simulator to break at the desired time or on the desired node or bus condition. You can monitor the activity of nodes or buses during specified times and pinpoint the cause of mismatched signal levels between expected and actual. To use breakpoints, perform the following steps:

1. On the Processing menu, point to **Simulation Debug** and click **Breakpoints**. The **Breakpoints** dialog box appears.
2. In the Equation text box, click **condition**. You can configure the logical conditions of individual nodes or buses, or you can set the time.
3. After you configure the equation conditions, select the action for the Quartus II Simulator. In the **Action** pull-down list, select **Stop**, **Warning Message**, **Error Message**, or **Information Message**. This selection defines the action when the condition is met.
4. You can also enter the text that appears when the Simulator encounters the breakpoint. If you do not make an entry in this box, the Quartus II software displays a default message.

Updating Memory Content

If your design includes memories, when the simulator stops at a breakpoint, you can view and edit the contents of the memories. To view your memories during a breakpoint in the simulation, on the Processing menu, point to **Simulation Debug** and click **Embedded Memory**.

Last Simulation Vector Outputs

The **Last Simulation Vector Outputs** command opens the Output Simulation Waveforms report generated by the last simulation. To use this command, on the Processing menu, point to **Simulation Debug** and click **Last Simulation Vector Outputs**.

You can open the current input vectors that you defined in the **Simulator Settings** dialog box with the **Current Vector Inputs** command. To use this command, on the Processing menu, point to **Simulation Debug** and click **Current Vector Inputs**. Lastly, you can overwrite the vector source file with the simulation outputs that open the resulting file.

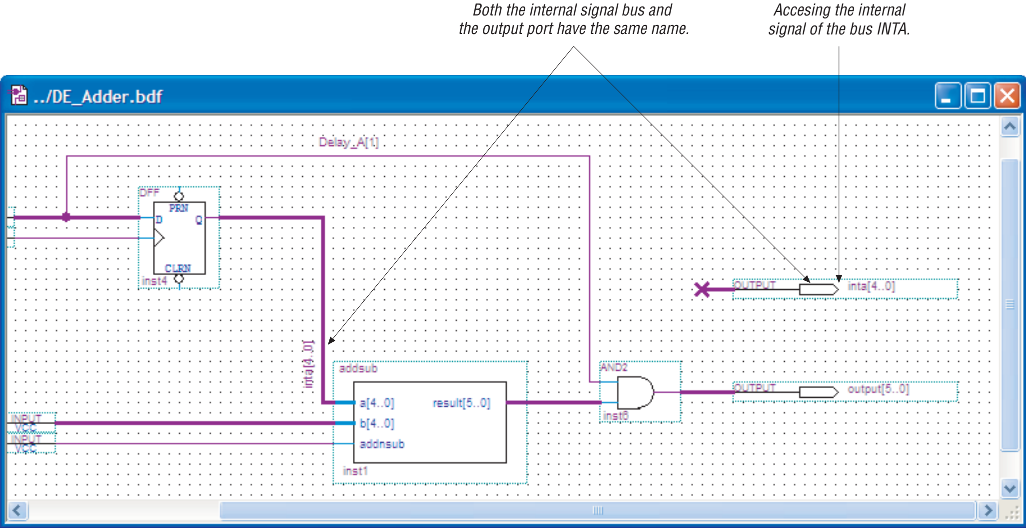
Conventional Debugging Process

During the design phase, tapping out internal signals is a common practice to debug simulation errors. Therefore, the Quartus II software enables you to tap out the signal for simulation debug and also enables you to pull out the internal signal to the physical I/O. The Quartus II software also offers SignalTap II and SignalProbe to further assist you with debugging.

Accessing Internal Signals for Simulation

You can conventionally debug by probing out the internal signals, which enables you to preserve the internal signals during synthesis. You can probe the internal signal by selecting the node or bus and specifying a name, and then adding an output port to the schematic with a similar name. Figure 1–9 shows an example of accessing internal signals for simulation from a schematic diagram.

Figure 1–9. Example of Tapping Out Internal Signal



For timing simulations, the simulation netlist is based on the Compilation post-Synthesis and post-Fitting netlist. Therefore, some of the internal nodes or buses are optimized away during compilation of the netlist. If an internal node is optimized away, the Quartus II software displays a warning message similar to the following in the **Warning** tab of the Messages window:

```
Warning: Compiler packed, optimized or synthesized away node "DataU".
Ignored vector source file node.
```

This internal node is ignored by the Quartus II Simulator.

If you would like to tap out the D and Q ports of registers, turn on **Add D and Q ports of register node to Simulation Output Waveform** from the Assignment Editor. This feature is only available for functional simulations.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF form.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

You can change the Functional, Timing, or Timing using Fast Timing Model simulation modes by typing the following at the command prompt:

```
simulation_mode <mode> ↵
```

To initialize the simulation for the current design, use the following command. During initialization, the Simulator builds the simulation netlist and sets the simulation time to zero.

The option `-ignore_vector_file` is set to **Off** by default when the source vector file exists for simulation. The Quartus II software ignores the source vector file during simulation if the option `-ignore_vector_file` is set to **On**. The `-end_time` option is used only when the `-ignore_vector_file` option is set to **On**.

```
initialize_simulation [-h | -help] [-long_help] [-check_outputs <On | Off>] \
[-end_time <end_time>] [-glitch_filtering <On | Off>] [-ignore_vector_file <On | Off>] \
[-memory_limiter <On | Off>] [-power_vcd_output <target_file>] \
[-read_settings_files <On | Off>] [-saf_output <target_file>] \
[-sim_mode <functional | timing | timing_using_fast_timing_model >] \
[-vector_source <vector_source_file>] [-write_settings_files <On | Off>] \
-simulation_results_format <.vwf | .cvwf | .vcd> -vector_source <vector source file>
```

To force the specified signal or group of signals to the specified value, type the following at the command prompt:

```
force_simulation_value [-h | -help] [-long_help] -node <hpath> <value> ↵
```

To turn on the simulator to simulate the design for a specified time, type the following at a command prompt:

```
run_simulation [-h | -help] [-long_help] [-time <time>] ↵
```

If you do not set a specific length of time for the simulation run, it runs a complete simulation.

To create a breakpoint with a specified equation and action, type the following at the command prompt:

```
create_simulation_breakpoint [-h | -help] [-long_help] \
-action [Give Warning | Give Info | Give Error] \
-breakpoint <breakpoint_name> -equation <equation> [-user_message <message_text>]↵
```

To delete a breakpoint with a specified name, type the following at the command prompt:

```
delete_simulation_breakpoint [-h | -help] [-long_help] -breakpoint <breakpoint_name> ↵
```

Conclusion

Simulation plays an important role in ensuring the quality of a product. The Quartus II software offers various tools to assist you with simulation and helps reduce debugging time with the introduction of features like Glitch Filtering and Breakpoints.

Referenced Documents

This chapter references the following documents:

- *Analyzing Designs with the Quartus II Netlist Viewers* chapter in volume 1 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Mentor Graphics ModelSim Support* chapter in volume 3 of the *Quartus II Handbook*
- *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Manual*
- *Section I: Simulation* section in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 1–5 shows the revision history for this chapter.

Table 1–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0	No change to content.	—
November 2008 v8.1	Changed to 8½” × 11” page size. No change to content.	—
May 2008 v8.0.0	■ Updated “Introduction” on page 1–1. ■ Updated “Referenced Documents” on page 1–27.	Updated for the Quartus II 8.0 software release

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

2. Mentor Graphics ModelSim Support

QII53001-9.0.0

Introduction

An Altera® Quartus® II software subscription includes a no-cost entry-level software version of the ModelSim®-Altera software on a PC or UNIX platform. Altera also offers the ModelSim-Altera Subscription Edition software that has full support for Altera devices. You can use the ModelSim-Altera Starter Edition software to perform register transfer level (RTL) functional, post-synthesis, and gate-level timing simulations for either Verilog HDL or VHDL designs that target an Altera FPGA. This chapter provides detailed instructions about how to simulate your design in the ModelSim-Altera version or the Mentor Graphics® ModelSim software version. This chapter provides details about the specific libraries that are needed for an RTL functional, post-synthesis, and gate-level timing simulation.

The following topics are discussed in this chapter:

- "Background"
- "Software Compatibility" on page 2–3
- "Altera Design Flow with ModelSim-Altera or ModelSim Software" on page 2–3
- "Simulation Libraries" on page 2–4
- "Simulation Netlist Files" on page 2–12
- "Perform Simulation Using the ModelSim-Altera Software" on page 2–18
- "Perform Simulation Using the ModelSim Software" on page 2–38
- "Simulating Designs that Include Transceivers" on page 2–64
- "Using the NativeLink Feature with ModelSim-Altera or ModelSim Software" on page 2–70
- "Generating a Timing VCD File for PowerPlay" on page 2–78
- "Viewing a Waveform from a .wlf File" on page 2–78
- "Scripting Support" on page 2–79
- "Software Licensing and Licensing Setup in ModelSim-Altera Subscription Edition" on page 2–80

For more information about the current Quartus II software version, refer to the Altera website at www.altera.com.

Background

ModelSim-Altera software is included with your Altera software subscription and can be licensed for PC, Solaris, or Linux platforms to support either Verilog HDL or VHDL simulation. ModelSim-Altera software supports RTL functional, post-synthesis, and gate-level timing simulations for all Altera devices.

Table 2–1 describes the differences between the Mentor Graphics ModelSim SE/PE and ModelSim-Altera software versions.

Table 2–1. Comparison of ModelSim Software Versions

Product Feature	ModelSim SE	ModelSim PE	ModelSim-Altera	ModelSim-Altera Starter Edition
100% VHDL, Verilog HDL, mixed-HDL support	Optional	Optional	Supports only single-HDL simulation	Supports only single-HDL simulation
Complete HDL debugging environment	✓	✓	✓	✓
Optimized direct compile architecture	✓	✓	✓	✓
Industry-standard scripting	✓	✓	✓	✓
Flexible licensing	✓	Optional	✓	—
Verilog PLI support. Interfaces Verilog HDL designs to customer C code and third-party software	✓	✓	✓	✓
VHDL FLI support. Interfaces VHDL designs to customer C code and third-party software	✓	—	—	—
Standard Delay Format File annotation	✓	✓	✓ (1)	✓ (1)
Advanced debugging features and language-neutral licensing	✓	—	—	—
Customizable, user-expandable GUI and integrated simulation performance analyzer	✓	—	—	—
Integrated code coverage analysis and SWIFT support	✓	—	—	—
Accelerated VITAL and Verilog HDL primitives (3 times faster), and register transfer level (RTL) acceleration (5 times faster)	✓	—	—	—
Platform support	PC, UNIX, Linux	PC only	PC, UNIX, Linux	PC, UNIX, Linux
Precompiled libraries	No	No	Yes	Yes

Note to Table 2–1:

(1) ModelSim-Altera software only allows SDF annotation to modules in the Altera library.

Software Compatibility

Table 2–2 shows which ModelSim-Altera and ModelSim software version is compatible with the Quartus II software versions. ModelSim versions provided directly from Mentor Graphics do not correspond to specific Quartus II software versions.

For help with the ModelSim-Altera licensing setup, refer to "Software Licensing and Licensing Setup in ModelSim-Altera Subscription Edition" on page 2–80.

Table 2–2. Compatibility Between Software Versions

ModelSim-Altera Software	Quartus II Software *(1)*
ModelSim-Altera and ModelSim software version 6.4a	Quartus II software version 9.0
ModelSim-Altera and ModelSim software version 6.3g_p1	Quartus II software version 8.1
ModelSim-Altera and ModelSim software version 6.1g	Quartus II software version 6.1, 7.0, 7.1, 7.2, and 8.0
ModelSim-Altera and ModelSim software version 6.1d	Quartus II software version 6.0
ModelSim-Altera and ModelSim software version 6.0e	Quartus II software version 5.1
ModelSim-Altera and ModelSim software version 6.0c	Quartus II software version 5.0
ModelSim-Altera and ModelSim software version 5.8.e ModelSim-Altera and ModelSim software version 5.8	Quartus II software version 4.2

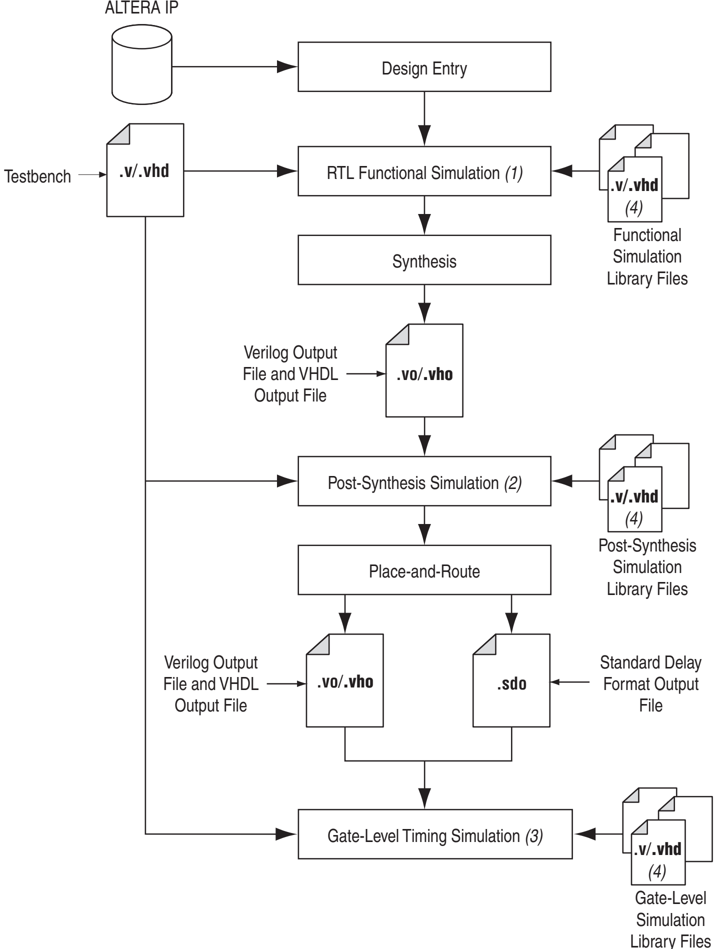
Note to Table 2–2:

(1) Updated ModelSim-Altera pre-compiled libraries are available for download on Altera's website for each release of the Quartus II service pack.

Altera Design Flow with ModelSim-Altera or ModelSim Software

Figure 2–1 illustrates an Altera design flow using the Mentor Graphics ModelSim-Altera software or the ModelSim software.

Figure 2–1. Altera Design Flow with ModelSim-Altera Software, ModelSim Software, and Quartus II Software



Notes to Figure 2–1:

(1) An RTL functional simulation is performed before a gate-level timing simulation or post-synthesis simulation. RTL functional simulation verifies the functionality of the design before synthesis and place-and-route. If you are performing an RTL functional simulation through NativeLink, you must complete Analysis and Elaboration first.

(2) A post-synthesis simulation verifies the functionality of a design after synthesis has been performed.

(3) Gate-level timing simulation is a post place-and-route simulation to verify the operation of the design after timing delays have been calculated.

(4) In the ModelSim-Altera software, you do not have to compile the simulation library files, as they are already precompiled for you.

Simulation Libraries

Simulation model libraries are required to run a simulation whether you are running an RTL functional simulation, post-synthesis simulation, or gate-level timing simulation. In general, running an RTL functional simulation requires the RTL functional simulation model libraries, while running a post-synthesis or gate-level timing simulation requires the gate-level timing simulation model libraries. You must compile the necessary library files before you can run the simulation.

However, there are a few exceptions where you are also required to compile gate-level timing simulation library files to perform RTL functional simulation. For example, the following list shows some of the Altera megafunctions' gate-level libraries required to perform an RTL functional simulation using third-party simulators:

- ALTCLKBUF
- ALTCLKCTRL
- ALTDQS
- ALTDQ
- ALTDDIO_IN
- ALTDDIO_OUT
- ALTDDIO_BIDIR
- ALTUFM_NONE
- ALTUFM_PARALLEL
- ALTUFM_SPI
- ALTMEMMULT
- ALTREMOTE_UPDATE

To identify which type of simulation libraries are required to run the simulation for a certain Altera megafunction, refer to the last page in the Altera megafunction MegaWizard™ Plug-In Manager. This explains which simulation library files are required to perform an RTL functional simulation for that particular megafunction.

Simulating the Transceiver Megafunction (for example, ALT2GXB) is also another exception that requires the gate-level libraries to perform RTL functional simulation and vice-versa.

For detailed, step-by-step instructions about how to simulate the Transceiver Megafunction, refer to "Simulating Designs that Include Transceivers" on page 2–64.

Pre-Compiled Simulation Libraries in the ModelSim-Altera Software

In the ModelSim-Altera software, the pre-compiled libraries for both functional and gate-level simulations are available. You do not have to explicitly compile these library files before running the simulation directly.

It is important that the pre-compiled libraries provided in <*ModelSim Altera path*>**/altera**> must be compatible with the version of the Quartus II software that is used to create the simulation netlist. To check whether the pre-compiled libraries are compatible with your version of the Quartus II software, refer to the <*ModelSim Altera path*>**/altera/version.txt** file. This file shows which version and build of the Quartus II software was used to create the pre-compiled libraries.

RTL Functional Simulation Libraries

RTL functional simulation libraries include the LPM simulation model, Altera megafunction simulation model, and the low-level primitive simulation model. Table 2–3 shows the pre-compiled library name and the location in the ModelSim-Altera software for all RTL functional simulation models in VHDL.

Table 2–3. Pre-Compiled RTL Functional Simulation Libraries in the ModelSim-Altera Software (VHDL)

RTL Simulation Model	Pre-Compiled Library Name	Location in ModelSim-Altera
LPM	lpm	<*ModelSim-Altera installation directory*>**\altera\vhdl\220model**
Altera Megafunction	altera_mf	<*ModelSim-Altera installation directory*>**\altera\vhdl\altera_mf**
Low-level Primitives	altera	<*ModelSim-Altera installation directory*>**\altera\vhdl\altera**
ALTGXB Megafunction (Stratix GX)	altgxb	<*ModelSim-Altera installation directory*>**\altera\vhdl\altgxb**
High-Level Primitives	sgate	<*ModelSim-Altera installation directory*>**\altera\vhdl\sgate**
Low-Level Primitives	alt_vtl	<*ModelSim-Altera installation directory*>**\altera\vhdl\alt_vtl**

Table 2–4 shows the pre-compiled library name and the location in the ModelSim-Altera software for all RTL functional simulation models in Verilog HDL.

Table 2–4. Pre-Compiled RTL Functional Simulation Libraries in the ModelSim-Altera Software (Verilog HDL)

RTL Simulation Model	Pre-Compiled Library Name	Location in ModelSim-Altera
LPM	lpm_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\220model**
Altera Megafunction	altera_mf_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\altera_mf**
Low-level Primitives	altera_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\altera**
ALTGXB Megafunction (Stratix GX)	altgxb_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\altgxb**
High-Level Primitives	sgate_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\sgate**
Low-Level Primitives	alt_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\alt_vtl**

Gate-Level Simulation Libraries

Gate-level simulation libraries include the supported Altera device atom simulation models. Table 2–5 shows the pre-compiled library name and location in the ModelSim-Altera software for all gate-level simulation models in VHDL.

Table 2–5. Pre-Compiled Gate-Level Simulation Libraries in the ModelSim-Altera Software (VHDL) (Part 1 of 2)

Device Simulation Model	Pre-Compiled Library Name	Location in ModelSim-Altera
Arria® II (without transceiver block)	arriaii	<*ModelSim-Altera installation directory*>**\altera\vhdl\arriaii**
Arria II (with transceiver block)	arriaii_hssi	<*ModelSim-Altera installation directory*>**\altera\vhdl\arriaii_hssi**
Arria II (with PCI Express)	arriaii_pcie_hip	<*ModelSim-Altera installation directory*>**\altera\vhdl\arriaii_pcie_hip**
Arria GX (without transceiver block)	arriagx	<*ModelSim-Altera installation directory*>**\altera\vhdl\arriagx**

Table 2–5. Pre-Compiled Gate-Level Simulation Libraries in the ModelSim-Altera Software (VHDL) (Part 2 of 2)

Device Simulation Model	Pre-Compiled Library Name	Location in ModelSim-Altera
Arria GX (with transceiver block)	arriagx_hssi	*<ModelSim-Altera installation directory>***\altera\vhdl\arriagx_hssi**
Stratix® IV	stratixiv	*<ModelSim-Altera installation directory>***\altera\vhdl\stratixiv**
Stratix IV (with transceiver block)	stratixiv_hssi	*<ModelSim-Altera installation directory>***\altera\vhdl\stratixiv_hssi**
Stratix IV (with PCI Express)	stratixiv_pcie_hip	*<ModelSim-Altera installation directory>***\altera\vhdl\stratixiv_pcie_hip**
Stratix III	stratixiii	*<ModelSim-Altera installation directory>***\altera\vhdl\stratixiii**
Stratix II	stratixii	*<ModelSim-Altera installation directory>***\altera\vhdl\stratixii**
Stratix II GX (without transceiver block)	stratixiigx	*<ModelSim-Altera installation directory>***\altera\vhdl\stratixiigx**
Stratix II GX (with transceiver block)	stratixiigx_hssi	*<ModelSim-Altera installation directory>***\altera\vhdl\stratixiigx_hssi**
Stratix	stratix	*<ModelSim-Altera installation directory>***\altera\vhdl\stratix**
Stratix GX (without transceiver block)	stratixgx	*<ModelSim-Altera installation directory>***\altera\vhdl\stratixgx**
Stratix GX (with transceiver block)	stratixgx_gxb	*<ModelSim-Altera installation directory>***\altera\vhdl\stratixgx_gxb**
HardCopy® II	hardcopyii	*<ModelSim-Altera installation directory>***\altera\vhdl\hardcopyii**
Cyclone® III	cycloneiii	*<ModelSim-Altera installation directory>***\altera\vhdl\cycloneiii**
Cyclone II	cycloneii	*<ModelSim-Altera installation directory>***\altera\vhdl\cycloneii**
Cyclone	cyclone	*<ModelSim-Altera installation directory>***\altera\vhdl\cyclone**
MAX® II	maxii	*<ModelSim-Altera installation directory>***\altera\vhdl\maxii**
MAX 7000 MAX 3000	max	*<ModelSim-Altera installation directory>***\altera\vhdl\max**
APEX™ II	apexii	*<ModelSim-Altera installation directory>***\altera\vhdl\apexii**
APEX 20K	apex20k	*<ModelSim-Altera installation directory>***\altera\vhdl\apex20k**
APEX 20KC APEX 20KE Excalibur™	apex20ke	*<ModelSim-Altera installation directory>***\altera\vhdl\apex20ke**
FLEX® 10KE ACEX® 1K	flex10ke	*<ModelSim-Altera installation directory>***\altera\vhdl\flex10ke**
FLEX 6000	flex6000	*<ModelSim-Altera installation directory>***\altera\vhdl\flex6000**

Table 2–6 shows the pre-compiled library name and the location in the ModelSim-Altera software for all the gate-level simulation models in Verilog HDL.

Table 2–6. Pre-Compiled Gate-Level Simulation Libraries in the ModelSim-Altera Software (Verilog HDL)

Device Simulation Model	Pre-Compiled Library Name	Location in ModelSim-Altera
Arria II (without transceiver block)	arriaii_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\arriaii**
Arria II (with transceiver block)	arriaii_hssi_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\arriaii_hssi**
Arria II (with PCI Express)	arriaii_pcie_hip_ver	<*ModelSim-Altera installation directory*> **\altera\verilog\arriaii_pcie_hip**
Arria GX (without transceiver block)	arriagx_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\arriagx**
Arria GX (with transceiver block)	arriagx_hssi_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\arriagx_hssi**
Stratix IV	stratixiv_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\stratixiv**
Stratix IV (with transceiver block)	stratixiv_hssi_ver	<*ModelSim-Altera installation directory*>**altera\verilog\stratixiv_hssi**
Stratix IV (with PCI Express)	stratixiv_pcie_hip_ver	<*ModelSim-Altera installation directory*> **\altera\verilog\stratixiv_pcie_hip**
Stratix III	stratixiii_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\stratixiii**
Stratix II	stratixii_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\stratixii**
Stratix II GX (without transceiver block)	stratixiigx_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\stratixiigx**
Stratix II GX (with transceiver block)	stratixiigx_hssi_ver	<*ModelSim-Altera installation directory*> **\altera\verilog\stratixiigx_hssi**
Stratix	stratix_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\stratix**
Stratix GX (without transceiver block)	stratixgx_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\stratixgx**
Stratix GX (with transceiver block)	stratixgx_gxb_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\stratixgx_gxb**
HardCopy II	hardcopyii_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\hardcopyii**
Cyclone III	cycloneiii_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\cycloneiii**
Cyclone II	cycloneii_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\cycloneii**
Cyclone	cyclone_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\cyclone**
MAX II	maxii_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\maxii**
MAX 7000 MAX 3000	max_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\max**
APEX II	apexii_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\apexii**
APEX 20K	apex20k_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\apex20k**
APEX 20KC APEX 20KE Excalibur™	apex20ke_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\apex20ke**
FLEX 10KE ACEX 1K	flex10ke_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\flex10ke**
FLEX 6000	flex6000_ver	<*ModelSim-Altera installation directory*>**\altera\verilog\flex6000**

Simulation Library Files in the Quartus II Software

In ModelSim SE/PE, no pre-compiled libraries are available. You must compile the necessary libraries to perform RTL functional or gate-level simulation. The following sections show the location of these library files in the Quartus II directory structure. You can refer to these library files for a particular simulation model.

RTL Functional Simulation Library Files

Table 2–7 shows the VHDL RTL simulation model location in the Quartus II directory.

Table 2–7. RTL Functional Simulation Library Files in the Quartus II Directory (VHDL)

RTL Simulation Model	VHDL Libraries
ALTGX Megafunction (Stratix IV GX)	<*path to Quartus II installation directory*>**\eda\sim_lib\stratixiv_hssi_components.vhd**
	<*path to Quartus II installation directory*>**\eda\sim_lib\stratixiv_hssi_atoms.vhd**
LPM	<*path to Quartus II installation directory*>**\eda\sim_lib\220pack.vhd**
	<*path to Quartus II installation directory*>**\eda\sim_lib\220model.vhd**
	<*path to Quartus II installation directory*>**\eda\sim_lib\220model_87.vhd** *(1)*
Altera Megafunction	<*path to Quartus II installation directory*>**\eda\sim_lib\altera_mf_components.vhd**
	<*path to Quartus II installation directory*>**\eda\sim_lib\altera_mf.vhd**
	<*path to Quartus II installation directory*>**\eda\sim_lib\altera_mf_87.vhd** *(1)*
ALT2GXB Megafunction (Stratix II GX)	<*path to Quartus II installation directory*>**\eda\sim_lib\stratixiigx_hssi_components.vhd**
	<*path to Quartus II installation directory*>**\eda\sim_lib\stratixiigx_hssi_atoms.vhd**
	<*path to Quartus II installation directory*>**\eda\sim_lib\arriagx_hssi_components.vhd**
	<*path to Quartus II installation directory*>**\eda\sim_lib\arriagx_hssi_atoms.vhd**
ALTGXB Megafunction (Stratix GX)	<*path to Quartus II installation directory*>**\eda\sim_lib\stratixgx_mf_components.vhd**
	<*path to Quartus II installation directory*>**\eda\sim_lib\stratixgx_mf.vhd**
High-Level Primitives	<*path to Quartus II installation directory*>**\eda\sim_lib\sgate_pack.vhd**
	<*path to Quartus II installation directory*>**\eda\sim_lib\sgate.vhd**
Low-Level Primitives	<*path to Quartus II installation directory*>**\eda\sim_lib\altera_primitives_components.vhd**
	<*path to Quartus II installation directory*>**\eda\sim_lib\altera_primitives.vhd**

Note to Table 2–7:

(1) Simulating a design that uses VHDL-1987.

Table 2–8 shows the Verilog RTL simulation model location in the Quartus II directory.

Table 2–8. RTL Functional Simulation Library Files in the Quartus II Directory (Verilog HDL) (Part 1 of 2)

RTL Simulation Model	Verilog HDL Libraries
ALTGX Megafunction (Stratix IV GX)	<*path to Quartus II installation directory*>**\eda\sim_lib\stratixiv_hssi_atoms.v**
LPM	<*path to Quartus II installation directory*>**\eda\sim_lib\220model.v**
Altera Megafunction	<*path to Quartus II installation directory*>**\eda\sim_lib\altera_mf.v**
ALT2GXB Megafunction (Stratix II GX)	<*path to Quartus II installation directory*>**\eda\sim_lib\stratixiigx_hssi_atoms.v**
	<*path to Quartus II installation directory*>**\eda\sim_lib\arriagx_hssi_atoms.v**
ALTGXB Megafunction (Stratix GX)	<*path to Quartus II installation directory*>**\eda\sim_lib\stratixgx_mf.v**

Table 2–8. RTL Functional Simulation Library Files in the Quartus II Directory (Verilog HDL) (Part 2 of 2)

RTL Simulation Model	Verilog HDL Libraries
High-Level Primitives	<*path to Quartus II installation directory*>**\eda\sim_lib\sgate.v**
Low-Level Primitives	<*path to Quartus II installation directory*>**\eda\sim_lib\altera_primitives.v**

Gate-Level Simulation Library Files

Table 2–9 shows the VHDL gate-level simulation model location in the Quartus II directory.

Table 2–9. Gate-Level Timing Simulation Library Files in Quartus II Software (VHDL) (Part 1 of 2)

Device Simulation Model	Location in Quartus II Directory Structure
Arria II (without transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\arriaii_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\arriaii_atoms.vhd**
Arria II (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\arriaii_hssi_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\arriaii_hssi_atoms.vhd**
Arria II (with PCI Express)	<*Quartus II installation directory*>**\eda\sim_lib\arriaii_pcie_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\arriaii_pcie_atoms.vhd**
Arria GX (without transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\arriagx_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\arriagx_atoms.vhd**
Arria GX (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\arriagx_hssi_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\arriagx_hssi_atoms.vhd**
Stratix IV	<*Quartus II installation directory*>**\eda\sim_lib\stratixiv_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixiv_atoms.vhd**
Stratix IV (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiv_hssi_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixiv_hssi_atoms.vhd**
Stratix IV (with PCI Express)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiv_pcie_hip_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixiv_pcie_hip_atoms.vhd**
Stratix III	<*Quartus II installation directory*>**\eda\sim_lib\stratixiii_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixiii_atoms.vhd**
Stratix II	<*Quartus II installation directory*>**\eda\sim_lib\stratixii_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixii_atoms.vhd**
Stratix II GX (without transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiigx_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixiigx_atoms.vhd**
Stratix II GX (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiigx_hssi_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixiigx_hssi_atoms.vhd**
Stratix	<*Quartus II installation directory*>**\eda\sim_lib\stratix_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratix_atoms.vhd**
Stratix GX	<*Quartus II installation directory*>**\eda\sim_lib\stratixgx_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixgx_atoms.vhd**
Stratix GX (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixgx_hssi_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixgx_hssi_atoms.vhd**

Table 2–9. Gate-Level Timing Simulation Library Files in Quartus II Software (VHDL) (Part 2 of 2)

Device Simulation Model	Location in Quartus II Directory Structure
HardCopy II	<*Quartus II installation directory*>**\eda\sim_lib\hardcopyii_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\hardcopyii_atoms.vhd**
Cyclone III	<*Quartus II installation directory*>**\eda\sim_lib\cycloneiii_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\cycloneiii_atoms.vhd**
Cyclone II	<*Quartus II installation directory*>**\eda\sim_lib\cycloneii_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\cycloneii_atoms.vhd**
Cyclone	<*Quartus II installation directory*>**\eda\sim_lib\cyclone.vhd** <*Quartus II installation directory*>**\eda\sim_lib\cyclone_atoms.vhd**
MAX II	<*Quartus II installation directory*>**\eda\sim_lib\maxii_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\maxii_atoms.vhd**
MAX 7000 MAX 3000	<*Quartus II installation directory*>**\eda\sim_lib\max_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\max_atoms.vhd**
APEX II	<*Quartus II installation directory*>**\eda\sim_lib\apexii_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\apexii_atoms.vhd**
APEX 20K	<*Quartus II installation directory*>**\eda\sim_lib\apex20k_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\apex20k_atoms.vhd**
APEX 20KC APEX 20KE Excalibur	<*Quartus II installation directory*>**\eda\sim_lib\apex20ke_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\apex20ke_atoms.vhd**
FLEX 6000	<*Quartus II installation directory*>**\eda\sim_lib\flex6000_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\flex6000_atoms.vhd**
FLEX 10KE ACEX 1K	<*Quartus II installation directory*>**\eda\sim_lib\flex10ke_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\flex10ke_atoms.vhd**

Table 2–10 shows the Verilog HDL gate-level simulation model location in the Quartus II directory.

Table 2–10. Gate-Level Timing Simulation Library Files in the Quartus II Software (Verilog HDL) (Part 1 of 2)

Device Simulation Model	Location in Quartus II Directory Structure
Arria II (without transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\arriaii_atoms.v**
Arria II (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\arriaii_hssi_atoms.v**
Arria II (with PCI Express)	<*Quartus II installation directory*>**\eda\sim_lib\arriaii_pcie_atoms.v**
Arria GX (without transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\arriagx_atoms.v**
Arria GX (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\arriagx_hssi_atoms.v**
Stratix IV	<*Quartus II installation directory*>**\eda\sim_lib\stratixiv_atoms.v**
Stratix IV (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiv_hssi_atoms.v**

Table 2–10. Gate-Level Timing Simulation Library Files in the Quartus II Software (Verilog HDL) (Part 2 of 2)

Device Simulation Model	Location in Quartus II Directory Structure
Stratix IV (with PCI Express)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiv_pcie_hip_atoms.v**
Stratix III	<*Quartus II installation directory*>**\eda\sim_lib\stratixiii_atoms.v**
Stratix II	<*Quartus II installation directory*>**\eda\sim_lib\stratixii_atoms.v**
Stratix II GX (without transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiigx_atoms.v**
Stratix II GX (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiigx_hssi_atoms.v**
Stratix	<*Quartus II installation directory*>**\eda\sim_lib\stratix_atoms.v**
Stratix GX	<*Quartus II installation directory*>**\eda\sim_lib\stratixgx_atoms.v**
Stratix GX (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixgx_hssi_atoms.v**
HardCopy II	<*Quartus II installation directory*>**\eda\sim_lib\hardcopyii_atoms.v**
Cyclone III	<*Quartus II installation directory*>**\eda\sim_lib\cycloneiii_atoms.v**
Cyclone II	<*Quartus II installation directory*>**\eda\sim_lib\cycloneii_atoms.v**
Cyclone	<*Quartus II installation directory*>**\eda\sim_lib\cyclone_atoms.v**
MAX II	<*Quartus II installation directory*>**\eda\sim_lib\maxii_atoms.v**
MAX 7000 MAX 3000	<*Quartus II installation directory*>**\eda\sim_lib\max_atoms.v**
APEX II	<*Quartus II installation directory*>**\eda\sim_lib\apexii_atoms.v**
APEX 20K	<*Quartus II installation directory*>**\eda\sim_lib\apex20k_atoms.v**
APEX 20KC APEX 20KE Excalibur	<*Quartus II installation directory*>**\eda\sim_lib\apex20ke_atoms.v**
FLEX 6000	<*Quartus II installation directory*>**\eda\sim_lib\flex6000_atoms.v**
FLEX 10KE ACEX 1K	<*Quartus II installation directory*>**\eda\sim_lib\flex10ke_atoms.v**

Simulation Netlist Files

Simulation netlist files are required to perform post-synthesis simulation or gate-level timing simulation. These simulation netlist files are generated from the EDA Netlist Writer.

If you are performing post-synthesis simulation, the Verilog HDL Output File (***.vo**) or VHDL Output File (***.vho**) is required. For the steps to generate post-synthesis simulation netlist files for ***.vo** or ***.vho** files, refer to "Generate Post-Synthesis Simulation Netlist Files" on page 2–13.

If you are performing gate-level timing simulation, the ***.vo** file or ***.vho** file and the Standard Delay Format Output File (***.sdo**) are also required. The ***.sdo** file is used to annotate the delay for the elements found in the ***.vo** or ***.vho** file. The section "Generate Gate-Level Timing Simulation Netlist Files" on page 2–14 shows you the steps to generate simulation netlist files for ***.vo** or ***.vho**, and ***.sdo** files.

 © March 2009 Altera Corporation

Generate Post-Synthesis Simulation Netlist Files

The following steps describe the process of generating post-synthesis simulation netlist files in the Quartus II software:

1. Perform Analysis and Synthesis. On the Processing menu, point to **Start** and click **Start Analysis & Synthesis** (you can also perform this after step 2).
2. Turn on the **Generate Netlist for Functional Simulation Only** option by performing the following steps:
   a. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
   b. In the **Category** list, expand **EDA Tool Settings** and click **Simulation**. The **Simulation** page appears.
   c. In the **Tool name** list:
      - If you are using the ModelSim-Altera software, select **ModelSim-Altera**.
      - If you are using the Mentor Graphics ModelSim software, select **ModelSim**.
   d. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist to be generated by editing or browsing to a directory in the **Output directory** box.
   e. Click **More EDA Netlist Writer Settings**. The **More EDA Netlist Writer Settings** dialog box appears.
   f. In the **Existing options settings** list, click **Generate netlist for functional simulation only**.
   g. In the **Setting** list under **Options**, select **On**.
   h. Click **OK**.
   i. In the **Settings** dialog box, click **OK**.
3. Run the EDA Netlist Writer. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer**.

During the EDA Netlist Writer stage, the Quartus II software produces a ***.vo** file or ***.vho** file that can be used for post-synthesis simulations in the ModelSim software. This netlist file is mapped to architecture-specific primitives. No timing information is included at this stage. The resulting netlist is located in the output directory you specified in the **Settings** dialog box, which defaults to the <*project directory*>**/simulation/modelsim** directory.

If you want to generate a post-synthesis simulation netlist with just the cell delays, you can generate an ***.sdo** file without running the Fitter. In this case, the ***.sdo** file includes all timing values for only the device cells. Interconnect delays are not included because fitting (placement and routing) has not been performed. To generate the post-synthesis netlist and the ***.sdo** file, type the following commands at a command prompt:

- `quartus_map <project name> -c <revision name>` ↵

- `quartus_sta <project name> -c <revision name> --post_map ↵`

  or

  ```
  quartus_tan <project name> -c <revision name> --post_map \
  --zero_ic_delays ↵
  ```

- ```
  quartus_eda <project name> -c <revision name> --simulation \
  --tool= <3rd party EDA tool> --format=<HDL language> ↵
  ```

For more information about the **–format** and **–tool** options, type the following command at a command prompt:

```
quartus_eda --help=<options> ↵
```

Generate Gate-Level Timing Simulation Netlist Files

To perform gate-level timing simulation, the ModelSim or ModelSim-Altera software requires information about how the design was placed into device-specific architectural blocks. The Quartus II software provides this information in the form of a ***.vo** file for Verilog HDL designs and a ***.vho** file for VHDL designs. The accompanying timing information is stored in the ***.sdo** file, which annotates the delay for the elements found in the ***.vo** file or ***.vho** file.

To generate a gate-level timing simulation netlist in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the "+" icon to expand **EDA Tool Settings** and select **Simulation**. The **Simulation** page appears.
3. In the **Tool name** list:
   - If you are using the ModelSim-Altera software, select **ModelSim-Altera**.
   - If you are using the Mentor Graphics ModelSim software, select **ModelSim**.
4. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist to be generated by editing or browsing to a directory in the **Output directory** box.
5. Click **OK**.
6. In the **Settings** dialog box, click **OK**.
7. If you have not run a full compilation, perform a full compilation. On the Processing menu, click **Start Compilation**.
8. If you have already run a full compilation, run the EDA Netlist Writer. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer**.

   During the full compilation or EDA Netlist Writer stage, the Quartus II software produces a ***.vo** file, ***.vho** file, and a ***.sdo** file used for gate-level timing simulations in the ModelSim software. This netlist file is mapped to architecture-specific primitives. The timing information for the netlist is included in the *.**sdo** file. The resulting netlist is located in the output directory you specified in the **Settings** dialog box, which defaults to the *<project directory>***/simulation/modelsim** directory.
```

Generating Timing Simulation Netlist Files with Different Timing Models

In Stratix III and later devices (for example, Cyclone III and Stratix IV), you can specify different temperature and voltage parameters to generate the timing simulation netlist files. If you enable the Quartus II Classic Timing Analyzer or Quartus II TimeQuest Timing Analyzer when generating the timing simulation netlist files (***.vo** or ***.vho** and ***.sdo**), different timing models for different operating conditions are used by default. Multi-corner timing analysis is run by default during the full compilation.

Table 2–11 shows an example of default-available operating conditions (model, voltage, and temperature) for Stratix III and Cyclone III devices.

Table 2–11. Default Available Operating Conditions for Stratix III and Cyclone III Devices

Device Family	Model	Voltage	Temperature (°C)
Stratix III	Slow	1100 mV	85°
	Slow	1100 mV	0°
	Fast	1100 mV	0°
Cyclone III	Slow	1200 mV	85°
	Slow	1200 mV	0°
	Fast	1200 mV	0°

If multi-corner timing analysis is not run during full compilation, perform the following steps to manually generate the simulation netlist files (***.vo** or ***.vho** and ***.sdo**) for the three different operating conditions listed in Table 2–11:

1. Generate all the available corner models at all operating conditions. Type the following command at a command prompt:

```
quartus_sta <project name> --multicorner ↵
```

2. Generate the timing simulation netlist files for all three corners specified in Table 2–11. Perform steps 2 through 8 in "Generate Gate-Level Timing Simulation Netlist Files" on page 2–14. The output files are generated in the simulation output directory.

The following examples show the timing simulation netlist files generated for the operating conditions of the preceding steps, when **Verilog** is selected as the output netlist format:

First slow corner (slow, 1100 mV, 85° C):

- **.vo** file—<*revision name*>**.vo**
- **.sdo** file—<*revision name*>**_v.sdo**

The <*revision_name*>**.vo** and <*revision name*>**_v.sdo** are generated for backward compatibility in case there are existing scripts that still use them.

- **.vo** file—<*revision name*>_<*speedgrade*>_1100mv_85c_slow**.vo**
- **.sdo** file—<*revision name*>_<*speedgrade*>_1100mv_85c_v_slow**.sdo**

Second slow corner (slow, 1100 mV, 0° C):

- **.vo** file—<*revision name*>_<*speedgrade*>_1100mv_0c_slow**.vo**
- **.sdo** file—<*revision name*>_<*speedgrade*>_1100mv_0c_v_slow**.sdo**

Fast corner (fast, 1100 mV, 0° C):

- **.vo** file—<*revision name*>_min_1100mv_0c_fast**.vo**
- **.sdo** file—<*revision name*>_min_1100mv_0c_v_fast**.sdo**

For older devices, a slow-corner (worst case) timing model is used by default. There are only two timing models available: slow-corner and fast-corner. To generate the timing simulation netlist files using a different timing model, you must run the Quartus II Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer with a different timing model before you start the EDA Netlist Writer.

To run the Quartus II Classic Timing Analyzer with the best-case model, on the Processing menu, point to **Start** and click **Start Classic Timing Analyzer (Fast Timing Model)**. After timing analysis is complete, the Compilation Report appears.

You can also type the following command at a command prompt:

```
quartus_tan <project_name> --fast_model=on ↵
```

To run the Quartus II TimeQuest Timing Analyzer with a best-case model, use the `-fast_model` option after you create the timing netlist.

The following command enables the fast timing models:

```
create_timing_netlist --fast_model ↵
```

After running the Quartus II Classic or Quartus II TimeQuest Timing Analyzer, perform steps 2 through 8 in "Generate Gate-Level Timing Simulation Netlist Files" on page 2–14 to generate the timing simulation netlist file. For fast corner timing models, the `-fast` post fix is added to the ***.vo** or ***.vho** and ***.sdo** file (for example, **my_project_fast.vo** or **my_project_fast.vho** and **my_project_fast.sdo**).

For more information about running multi-corner timing analysis, refer to the *Quartus II Classic Timing Analyzer* or the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Disable Timing Violation on Registers

In certain situations, a timing violation can be ignored and you might want to disable timing violations on registers. For example, timing violations that occur in internal synchronization registers used for asynchronous clock domain crossing.

By default, the **x_on_violation_option logic** option is **On**, which means simulation shows "**x**" whenever a timing violation occurs. To disable showing the timing violation on certain registers, set the **x_on_violation_option logic** option to **Off** on those registers. The following command is the Quartus II Tcl Command to disable timing violation on registers. This Tcl command is also stored in the **.qsf** file.

```
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to <register_name>
```

Compile Libraries Using the EDA Simulation Library Compiler

The EDA Simulation Library Compiler is used to compile Verilog HDL and VHDL simulation libraries for all Altera devices and supported third-party simulators. You can use this tool to compile all libraries required by RTL and gate-level simulation.

When the compilation targets third-party simulation tools such as ModelSim and NC-Sim, the compiled libraries are kept in either the directory you set or the default directory. When you perform the simulation using these simulators, you can use or re-use the compiled libraries and avoid the overhead associated with redundant library compilations.

However, if the Synopsys VCS or VCS-MX software performs the compilation while running the EDA Simulation Library Compiler, the option files (**simlib_comp.vcs**) are generated after compilation. You can then include your design and testbench files in the option files and invoke them with the `vcs` command.

Before using this option, ensure the appropriate simulation tools are already installed and their paths are specified. To specify the path, refer to "Setting Up NativeLink" on page 2–70.

Run the EDA Simulation Library Compiler through the GUI

Starting with the Quartus II software 9.0 release, the EDA Simulation Library Compiler contains a GUI. To compile libraries with the EDA Simulation Library Compiler GUI, perform the following steps:

1. On the Tools menu, click **EDA Simulation Library Compiler**. The **EDA Simulation Library Compiler** dialog box appears.
2. In the **Tool name** entry box under **EDA simulation tool**, select a simulation tool.

   The **Executable location** box displays location of the simulation tool you specified. This location must be set before running the EDA Simulation Library Compiler.
3. Under **Library families**, select one or more device families for your design compilation and move them to the **Selected families** box.
4. Under **Library language**, select **VHDL**, **Verilog**, or both.
5. In the **Output directory** field, specify a location in which to store the compiled libraries or option files.
6. Click **Start Compilation**.

For example, if you want to simulate a Verilog HDL design on a Stratix II device in the ModelSim simulator, do the following:

- Select **ModelSim** in the **Tool name** field.
- Move **Stratix II** from the **Available families** list to the **Selected families** list.
- Select **Verilog** in the **Library language** field.
- Specify a location in the **Output directory** field in which to keep the user-compiled libraries.
- Click **Start Compilation**.

When the EDA Simulation Library Compiler finishes, all required libraries are compiled and stored in the output location you specified. The next time you perform simulation in ModelSim, you only have to compile your design and testbench files. You do not have to compile the Altera libraries again.

The EDA Simulation Library Compiler supports only ModelSim SE/PE. It does not support ModelSim-Altera, because ModelSim-Altera already contains precompiled libraries.

If you use NativeLink to run the simulation, refer to "Using the NativeLink Feature with ModelSim-Altera or ModelSim Software" on page 2–70.

Run EDA Simulation Library Compiler In Command Line

To run the EDA Simulation Library Compiler in the command line, type the following command:

```
quartus_sh --simlib_comp -family <device> -tool <simulation tool name>
-language <language> -directory <directory> ↵
```

For more information about the command's options and how to define them, type the following command:

```
quartus_sh --help=simlib_comp ↵
```

Perform Simulation Using the ModelSim-Altera Software

Simulation of Verilog HDL or VHDL designs with ModelSim-Altera software can be done at various levels to verify designs from different aspects. Simulation is divided into three categories: RTL functional simulation, post-synthesis simulation, and gate-level timing simulation. Simulation helps you verify your designs and debug them against any possible errors in the designs.

You can perform the simulation through the GUI or on the command line. The following sections provide step-by-step instructions to perform the simulation through the GUI and on the command line.

For high-speed simulation, you must select **ps** in the **Resolution** list for your simulator resolutions. If you choose slower than **ps**, the high-speed simulation may fail.

Simulating the VHDL Designs through the GUI

Simulating the VHDL design using the ModelSim-Altera GUI is user-friendly. You do not have to remember the commands to compile the libraries, or load and simulate the VHDL design files. You can use the ModelSim-Altera GUI to perform RTL functional simulation, post-synthesis simulation, and gate-level timing simulation. The following sections show how to perform simulation at various levels through the ModelSim-Altera GUI.

Perform RTL Functional Simulation

RTL functional simulation is typically performed to verify the syntax of the code and to check the functionality of the design. The following sections show how to perform RTL functional simulation in the ModelSim-Altera software for VHDL designs.

Use the following instructions to perform an RTL functional simulation for VHDL designs in the ModelSim-Altera software.

The ModelSim-Altera software includes pre-compiled simulation libraries. Creating simulation libraries and compiling simulation models are not required.

Compile Testbench and Design Files into the Work Library

The following instructions show you how to compile your testbench and design files into the work library using the ModelSim-Altera GUI.

1. In the ModelSim-Altera software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your designs are located.
3. Click **OK**.

To create the work library, perform the following steps:

1. In the ModelSim-Altera software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name `Work` in the text box.
4. Click **OK**.

To compile the testbench and design files into the work library, perform the following steps:

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library **Work**. The design files and testbench file should be compiled into the **Work** library.
3. Select the design files and testbench file, and click **Compile**.
4. Click **Done**.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following steps:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. In the **Start Simulation** dialog box, click the **Design** tab. In the **Resolution** list, select **ps**.
3. In the **Library** list, select and expand the **Work** library.
4. Select the top-level design unit (your testbench).
5. In the **Resolution** list, select **ps**.

6. For VHDL designs, if you have not included the libraries' mapped name in your design files or sub-files, perform the following steps:
   a. Click the **Libraries** tab.
   b. In the **Search Libraries** text box, click the **Add** button.
   c. Browse to the required pre-compiled library in the ModelSim-Altera software. You can either click **Browse** and go to the path <*ModelSim-Altera installation directory*>**/altera/vhdl/**<*pre-compiled library*> or you can just click the arrow button to select the <*pre-compiled library mapped name*>.

      Examples of <*pre-compiled library*> or <*pre-compiled library mapped name*> are **altera_mf** and **lpm**. The functional RTL simulation libraries are usually required for performing RTL functional simulation. For the complete set of libraries, refer to "Pre-Compiled Simulation Libraries in the ModelSim-Altera Software" on page 2–5.
   d. Click **OK** to add the libraries to the **Search Libraries** text box.
7. Click **OK**.

Running the Simulation

To run a simulation, perform the following steps:

1. On the View menu, point to **Debug Windows** and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to **Debug Windows** and click **Wave**.
3. Drag signals to monitor from the Objects window and drop them into the Wave window.
4. On the Processing menu, point to **Run** and click **Run 100 ps** to run the simulation for 100 ps.

Perform Post-Synthesis Simulation

Post-synthesis simulation verifies that functionality of the design is not lost after synthesis. You can create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the ModelSim-Altera software.

Before running post-synthesis simulation, generate post-synthesis simulation netlist files. Refer to the instructions in "Generate Post-Synthesis Simulation Netlist Files" on page 2–13.

The following sections help you perform a post-synthesis simulation for a VHDL design in the ModelSim-Altera software.

The ModelSim-Altera software includes pre-compiled simulation libraries. Creating simulation libraries and compiling simulation models are not required.

Compile Testbench and VHDL Output File into the Work Library

The following instructions show how you can compile your testbench and ***.vho** file into the work library using the ModelSim-Altera GUI.

To change to the simulation output directory, perform the following steps:

1. In the ModelSim-Altera software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your testbench or ***.vho** file is located. By default, the ***.vho** file is located in <*project directory*>**/simulation/modelsim**.
3. Click **OK**.

To create the work library, perform the following steps:

1. In the ModelSim-Altera software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name `Work` in the text box.
4. Click **OK**.

To compile the testbench and VHDL output (***.vho**) files into the work library, perform the following steps:

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library **Work**. The testbench and VHDL output (***.vho**) files should be compiled into the **Work** library.
3. Select the testbench and ***.vho** design files and click **Compile**.
4. Click **Done**.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following steps:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. In the **Start Simulation** dialog box, click the **Design** tab. In the **Resolution** list, select **ps**.
3. In the **Library** list, select and expand the **Work** library.
4. Select the top-level design unit (your testbench).
5. In the **Resolution** list, select **ps**.

6. For VHDL designs, if you have not included the libraries' mapped name in your design files or sub-files, perform the following steps:
   a. Click the **Libraries** tab.
   b. In the **Search Libraries** text box, click the **Add** button.
   c. Browse to the required pre-compiled library in the ModelSim-Altera software. You can either click **Browse** and go to the path <*ModelSim-Altera installation directory*>**/altera/vhdl/**<*pre-compiled library*> or you can just click the arrow button to select the <*pre-compiled library mapped name*>.

      Examples of <*pre-compiled library*> or <*pre-compiled library mapped name*> are **stratixiii** and **cycloneiii**. The gate-level simulation libraries are usually required for performing post-synthesis simulation. For the complete set of libraries, refer to "Pre-Compiled Simulation Libraries in the ModelSim-Altera Software" on page 2–5.
   d. Click **OK** to add the libraries to the **Search Libraries** text box.
7. Click **OK**.

Running the Simulation

To run a simulation, perform the following steps:

1. On the View menu, point to **Debug Windows** and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to **Debug Windows** and click **Wave**.
3. Drag signals to monitor from the Objects window and drop them into the Wave window.
4. On the Processing menu, point to **Run** and click **Run 100 ps** to run the simulation for 100 ps.

Perform Gate-Level Simulation

Gate-level simulation is a very important step in ensuring that the FPGA device's functionality is still correct and meets all required timing requirements after the design was placed and routed. You can create the gate-level netlist in the Quartus II software and use the netlist to perform gate-level simulation with the ModelSim-Altera software.

Before running gate-level simulation, generate gate-level timing simulation netlist files. Refer to the instructions in "Generate Gate-Level Timing Simulation Netlist Files" on page 2–14.

The following sections help you perform a gate-level simulation for a VHDL design in the ModelSim-Altera software.

The ModelSim-Altera software includes pre-compiled simulation libraries. Creating simulation libraries and compiling simulation models are not required.

Compile Testbench and VHDL Output File into the Work Library

The following instructions show how you can compile your testbench and ***.vho** file into the work library using the ModelSim-Altera GUI.

© March 2009 Altera Corporation

To change to the simulation output directory, perform the following steps:

1. In the ModelSim-Altera software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your testbench or ***.vho** file is located. By default, the ***.vho** file is located in *<project directory>***/simulation/modelsim**.
3. Click **OK**.

To create the work library, perform the following steps:

1. In the ModelSim-Altera software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name `Work` in the text box.
4. Click **OK**.

To compile the testbench and VHDL output (***.vho**) files into the work library, perform the following steps:

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library **Work**. The testbench and VHDL output (***.vho**) files should be compiled into the **Work** library.
3. Select the testbench and VHDL output (***.vho**) design files, and click **Compile**.
4. Click **Done**.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following steps:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. Click the **SDF** tab, and click **Add**. The **Add SDF Entry** dialog box appears.
3. In the **Add SDF Entry** dialog box, click **Browse** and select the ***.sdo** file. By default, the ***.sdo** file is located in *<project directory>***/simulation/modelsim**.
4. In the **Apply to Region** dialog box, type the instance path to which the ***.sdo** file is to be applied. For example, if you are using a testbench exported into the Quartus II software from a Vector Waveform File, the instance path is set to **/i1**.

You do not have to choose from the **Delay** list because the Quartus II EDA Netlist Writer generates the ***.sdo** file using the same value for the triplet (minimum, typical, and maximum timing values).

5. Click **OK**.
6. In the **Start Simulation** dialog box, click the **Design** tab. In the **Resolution** list, select **ps**.
7. In the **Library** list, select and expand the **Work** library.

8. Select the top-level design unit (your testbench).
9. In the **Resolution** list, select **ps**.
10. For VHDL designs, if you have not included the libraries' mapped name in your design files or sub-files, perform the following steps:
    a. Click the **Libraries** tab.
    b. In the **Search Libraries** text box, click the **Add** button.
    c. Browse to the required pre-compiled library in the ModelSim-Altera software. You can either click **Browse** and go to the path <*ModelSim-Altera installation directory*>**/altera/vhdl/**<*pre-compiled library*> or you can just click the arrow button to select the <*pre-compiled library mapped name*>.

       Examples of <*pre-compiled library*> or <*pre-compiled library mapped name*> are **stratixiii** and **cycloneiii**. The gate-level simulation libraries are usually required for performing gate-level simulation. For the complete set of libraries, refer to "Pre-Compiled Simulation Libraries in the ModelSim-Altera Software" on page 2–5.
    d. Click **OK** to add the libraries to the **Search Libraries** text box.
11. Click **OK**.

Running the Simulation

To run a simulation, perform the following steps:

1. On the View menu, point to **Debug Windows** and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to **Debug Windows** and click **Wave**.
3. Drag signals to monitor from the Objects window and drop them into the Wave window.
4. On the Processing menu, point to **Run** and click **Run 100 ps** to run the simulation for 100 ps.

Simulating Verilog HDL Designs through the GUI

Simulating the Verilog HDL design using the ModelSim-Altera GUI is user-friendly. You do not have to remember the commands to compile the libraries or load and simulate the Verilog HDL design files. You can use the ModelSim-Altera GUI to perform RTL functional simulation, post-synthesis simulation, and gate-level timing simulation. The following sections show how to perform simulation at various levels through the ModelSim-Altera GUI.

Perform RTL Functional Simulation

RTL functional simulation is typically performed to verify the syntax of the code and to check the functionality of the design. The following sections show how to perform RTL functional simulation in ModelSim-Altera for Verilog HDL designs.

Use the following instructions to perform an RTL functional simulation for Verilog HDL designs in the ModelSim-Altera software.

The ModelSim-Altera software includes pre-compiled simulation libraries. Creating simulation libraries and compiling simulation models are not required.

Compile Testbench and Design Files into the Work Library

The followings instructions show you how to compile your testbench and design files into the work library using the ModelSim-Altera GUI.

To change to the design directory, perform the following steps:

1. In the ModelSim-Altera software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your designs are located.
3. Click **OK**.

To create the work library, perform the following steps:

1. In the ModelSim-Altera software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name `Work` in the text box.
4. Click **OK**.

To compile the testbench and design files into the work library, perform the following steps:

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library **Work**. The design files and testbench file should be compiled into the **Work** library.
3. Select the design files and testbench file and click **Compile**.
4. Click **Done**.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following steps:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. In the **Start Simulation** dialog box, click the **Design** tab. In the **Resolution** list, select **ps**.
3. In the **Library** list, select and expand the **Work** library.
4. Select the top-level design unit (your testbench).
5. In the **Resolution** list, select **ps**.
6. Click the **Libraries** tab.
7. In the **Search Libraries** text box, click the **Add** button.

8. Browse to the required pre-compiled library in the ModelSim-Altera software. You can either click **Browse** and go to the path <*ModelSim-Altera installation directory*>**/altera/verilog/**<*pre-compiled library*> or you can just click the arrow button to select the <*pre-compiled library mapped name*>.

   Examples of <*pre-compiled library*> or <*pre-compiled library mapped name*> are **altera_mf_ver** and **lpm_ver**. The RTL simulation libraries are usually required for performing RTL functional simulation. For the complete set of libraries, refer to "Pre-Compiled Simulation Libraries in the ModelSim-Altera Software" on page 2–5.

9. Click **OK** to add the libraries to the **Search Libraries** text box.
10. Click **OK**.

Running the Simulation

To run a simulation, perform the following steps:

1. On the View menu, point to **Debug Windows** and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to **Debug Windows** and click **Wave**.
3. Drag signals to monitor from the Objects window and drop them into the Wave window.
4. On the Processing menu, point to **Run** and click **Run 100 ps** to run the simulation for 100 ps.

Perform Post-Synthesis Simulation

Post-synthesis simulation can be performed to verify that functionality of the design is not lost after synthesis. You can create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the ModelSim-Altera software.

Before running post-synthesis simulation, generate post-synthesis simulation netlist files. Refer to the instructions in "Generate Post-Synthesis Simulation Netlist Files" on page 2–13.

The following sections help you perform a post-synthesis simulation for a Verilog HDL design in the ModelSim-Altera software.

The ModelSim-Altera software includes pre-compiled simulation libraries. Creating simulation libraries and compiling simulation models are not required.

Compile Testbench and Verilog HDL Output File into the Work Library

The following instructions show how you can compile your testbench and Verilog HDL output file (***.vo**) into the work library using the ModelSim-Altera GUI.

To change to the simulation output directory, perform the following steps:

1. In the ModelSim-Altera software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your testbench or ***.vo** file is located. By default, the ***.vho** file is located in <*project directory*>**/simulation/modelsim**.

3. Click **OK**.

To create the work library, perform the following steps:

1. In the ModelSim-Altera software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name `Work` in the text box.
4. Click **OK**.

To compile the testbench and Verilog HDL output (***.vo**) files into the work library, perform the following steps:

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library **Work**. The testbench and Verilog HDL output (***.vo**) files should be compiled into the **Work** library.
3. Select the testbench and ***.vo** design files, and click **Compile**.
4. Click **Done**.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following steps:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. In the **Start Simulation** dialog box, click the **Design** tab. In the **Resolution** list, select **ps**.
3. In the **Library** list, select and expand the **Work** library.
4. Select the top-level design unit (your testbench).
5. In the **Resolution** list, select **ps**.
6. Click the **Libraries** tab.
7. In the **Search Libraries** text box, click the **Add** button.
8. Browse to the required pre-compiled library in the ModelSim-Altera software. You can either click **Browse** and go to the path <*ModelSim-Altera installation directory*>**/altera/verilog/**<*pre-compiled library*> or you can just click the arrow button to select the <*pre-compiled library mapped name*>.

   Examples of <*pre-compiled library*> or <*pre-compiled library mapped name*> are **stratixiii_ver** and **cycloneiii_ver**. The gate-level simulation libraries are usually required for performing post-synthesis simulation. For the complete set of libraries, refer to "Pre-Compiled Simulation Libraries in the ModelSim-Altera Software" on page 2–5.
9. Click **OK** to add the libraries to the **Search Libraries** text box.
10. Click **OK**.

Running the Simulation

To run a simulation, perform the following steps:

1. On the View menu, point to **Debug Windows** and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to **Debug Windows** and click **Wave**.
3. Drag signals to monitor from the Objects window and drop them into the Wave window.
4. On the Processing menu, point to **Run** and click **Run 100 ps** to run the simulation for 100 ps.

Perform Gate-Level Simulation

Gate-level simulation is a very important step in ensuring that the FPGA device's functionality is still correct and meets all required timing requirements after the design was placed and routed. You can create the gate-level netlist in the Quartus II software and use the netlist to perform gate-level simulation with the ModelSim-Altera software.

Before running gate-level simulation, generate gate-level timing simulation netlist files. Refer to the instructions in "Generate Gate-Level Timing Simulation Netlist Files" on page 2–14.

The following sections help you perform a gate-level simulation for a Verilog HDL design in the ModelSim-Altera software.

The ModelSim-Altera software includes pre-compiled simulation libraries. Creating simulation libraries and compiling simulation models are not required.

Compile Testbench and Verilog HDL Output File into the Work Library

The following instructions show how you can compile your testbench and ***.vo** file into the work library using the ModelSim-Altera GUI.

To change to the simulation output directory, perform the following steps:

1. In the ModelSim-Altera software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your testbench or ***.vo** file is located. By default, the ***.vo** file is located in <*project directory*>**/simulation/modelsim**.
3. Click **OK**.

To create the work library, perform the following steps:

1. In the ModelSim-Altera software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name `Work` in the text box.
4. Click **OK**.

To compile the testbench and ***.vo** files into the work library, perform the following steps:

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library **Work**. The testbench and ***.vo** files should be compiled into the **Work** library.
3. Select the testbench and ***.vo** design files, and click **Compile**.
4. Click **Done**.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

When simulating in Verilog HDL, the ***.sdo** file does not have to be manually specified. In the **$sdf_annotate** task, when the Quartus II software generates the ***.vo** file, the ModelSim-Altera software looks for the ***.sdo** file in the folder in which the VSIM was run. If your ***.sdo** file is not in the same directory in which you ran VSIM, copy the ***.sdo** file into your current directory.

To load a design, perform the following steps:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. In the **Start Simulation** dialog box, click the **Design** tab. In the **Resolution** list, select **ps**.
3. In the **Library** list, select and expand the **Work** library.
4. Select the top-level design unit (your testbench).
5. In the **Resolution** list, select **ps**.
6. Click the **Libraries** tab.
7. In the **Search Libraries** text box, click the **Add** button.
8. Browse to the required pre-compiled library in the ModelSim-Altera software. You can either click **Browse** and go to the path <*ModelSim-Altera installation directory*>**/altera/verilog/**<*pre-compiled library*> or you can just click the arrow button to select the <*pre-compiled library mapped name*>.

   Examples of <*pre-compiled library*> are **altera_mf_ver** and **lpm_ver**. The gate-level simulation libraries are usually required for performing gate-level simulation. For the complete set of libraries, refer to "Pre-Compiled Simulation Libraries in the ModelSim-Altera Software" on page 2–5.
9. Click **OK** to add the libraries to the **Search Libraries** text box.
10. Click **OK**.

Running the Simulation

To run a simulation, perform the following steps:

1. On the View menu, point to **Debug Windows** and click **Objects**. This command displays all objects in the current scope.

2. On the View menu, point to **Debug Windows** and click **Wave**.
3. Drag signals to monitor from the Objects window and drop them into the Wave window.
4. On the Processing menu, point to **Run** and click **Run 100 ps** to run the simulation for 100 ps.

Simulating the VHDL Designs at the Command Line

Simulating VHDL designs using the ModelSim-Altera command line gives you more flexibility and control to compile the libraries, and load and simulate the VHDL design files. All simulation commands are Tcl commands, which can be put into the ModelSim Macros file (***.do**). Using the ***.do** file allows you to run simulation in batch mode. You only have to execute the ***.do** file and the ModelSim-Altera tool automatically executes all commands in the ***.do** script macro file.

You can use the ModelSim-Altera command line to perform RTL functional simulation, post-synthesis simulation, and gate-level simulation. The following sections show how to perform simulation at various levels through the ModelSim-Altera command line.

Perform RTL Functional Simulation

RTL functional simulation is typically performed to verify the syntax of the code and to check the functionality of the design. The following sections show how to perform RTL functional simulation in ModelSim-Altera for VHDL designs.

The ModelSim-Altera software includes pre-compiled simulation libraries. Creating simulation libraries and compiling simulation models are not required.

Compile Testbench and Design Files into the Work Library

The following commands show how to compile your testbench and design files into the work library in the ModelSim-Altera command prompt.

To change to the design library, type the following command:

```
cd <your_design_directory> ↵
```

(for example, *cd:/designs*)

To create the work library, type the following commands:

```
vlib work ↵
vmap work work ↵
```

To compile the testbench and design files into the work library, type the following command:

```
vcom -work work <my_testbench.vhd> <my_design_files.vhd> ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, type the following command:

```
vsim -t ps -L <pre-compiled-library1> -L <pre-compiled-library2> work.<my_testbench> ↵
```

The <*pre-compiled-library1*> and <*pre-compiled-library2*> variables are the libraries required to compile your testbench. If you have multiple libraries, use the `-L` option for each library in the `vsim` command.

Examples of <*pre-compiled library*> are **altera_mf** and **lpm**. The functional RTL simulation libraries are usually required for performing RTL functional simulation. For the complete set of libraries, refer to "Pre-Compiled Simulation Libraries in the ModelSim-Altera Software" on page 2–5.

You can choose not to invoke `-L` in the `vsim` command for VHDL designs if you have already included the libraries' mapped name in your design files or sub-files.

Running the Simulation

To run a simulation, type the following commands:

```
add wave <signal name> ↵
run <time period> ↵
```

To add all signals in your testbench hierarchy, type the following command:

```
add wave * ↵
```

To run the simulation for 100 ps, type the following command:

```
run 100 ps ↵
```

Perform Post-Synthesis Simulation

Post-synthesis simulation can be performed to verify that functionality of the design is not lost after synthesis. You can create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the ModelSim-Altera software.

The following sections help you perform a post-synthesis simulation for a VHDL design in the ModelSim-Altera software.

The ModelSim-Altera software includes pre-compiled simulation libraries. Creating simulation libraries and compiling simulation models are not required.

Compile Testbench and VHDL Output File into the Work Library

Before running post-synthesis simulation, generate post-synthesis simulation netlist files. Refer to the instructions in "Generate Post-Synthesis Simulation Netlist Files" on page 2–13.

The following instructions show how to compile your testbench and ***.vho** file into the work library using the ModelSim-Altera GUI.

To change to the simulation output directory, type the following command:

```
cd <simulation_output_directory> ↵
```

(for example, **cd:/designs/modelsim/simulation**)

This directory contains the ***.vho** file, which is generated by the netlist writer.

To create the work library, type the following commands:

```
vlib work ↵
vmap work work ↵
```

To compile the testbench and ***.vho** files into the work library, type the following command:

```
vcom -work work <my_testbench.vhd> <my_design_netlists.vho> ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, type the following command:

```
vsim -t ps -L <pre-compiled-library1> -L <pre-compiled-library2> work.<my_testbench> ↵
```

The <*pre-compiled-library1*> and <*pre-compiled-library2*> variables are the libraries required to compile your testbench. If you have multiple libraries, use the `-L` option for each library in the `vsim` command.

Examples of <*pre-compiled library*> are **stratixiii** and **cycloneiii**. The gate-level simulation libraries are usually required for performing post-synthesis simulation. For the complete set of libraries, refer to "Pre-Compiled Simulation Libraries in the ModelSim-Altera Software" on page 2–5.

You can choose not to invoke `-L` in the `vsim` command for VHDL designs if you have already included the libraries' mapped name in your design files or sub-files.

Running the Simulation

To run a simulation, type the following commands:

```
add wave <signal name> ↵
run <time period> ↵
```

To add all signals in your testbench hierarchy, type the following command:

```
add wave * ↵
```

To run the simulation for 100 ps, type the following command:

```
run 100 ps ↵
```

Perform Gate-Level Simulation

Gate-level simulation is a very important step in ensuring that the FPGA device's functionality is still correct and meets all required timing requirements after the design was placed and routed. You can create the gate-level netlist in the Quartus II software and use the netlist to perform gate-level simulation with the ModelSim-Altera software.

The following sections help you perform a gate-level simulation for a VHDL design in the ModelSim-Altera software.

The ModelSim-Altera software includes pre-compiled simulation libraries. Creating simulation libraries and compiling simulation models are not required.

Compile Testbench and VHDL Output File into the Work Library

Before running gate-level simulation, generate gate-level timing simulation netlist files. Refer to the instructions in "Generate Gate-Level Timing Simulation Netlist Files" on page 2–14.

The following instructions show how to compile your testbench and ***.vho** file into the work library using the ModelSim-Altera GUI.

To change to the simulation output directory, type the following command:

```
cd <simulation_output_directory> ↵
```

(for example, **cd:/designs/modelsim/simulation**)

☞ This directory contains the ***.vho** file, which is generated by the netlist writer.

To create the work library, type the following commands:

```
vlib work ↵
vmap work work ↵
```

To compile the testbench and ***.vho** files into the work library, type the following command:

```
vcom -work work <my_testbench.vhd> <my_design_netlists.vho> ↵
```

☞ Resolve compile-time errors before proceeding to the next section.

Loading the Design

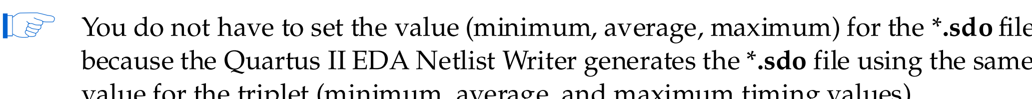
To load a design, type the following command:

```
vsim -t ps -sdftyp <design instance> = <path to *.sdo file> -L \
<pre-compiled-library1> -L <pre-compiled-library2> work.<my_testbench> ↵
```

The *<pre-compiled-library1>* and *<pre-compiled-library2>* variables are the libraries required to compile your testbench. If you have multiple libraries, use the `-L` option for each library in the `vsim` command.

Examples of *<pre-compiled library>* are **stratixiii** and **cycloneiii**. The gate-level simulation libraries are usually required for performing gate-level simulation. For the complete set of libraries, refer to "Pre-Compiled Simulation Libraries in the ModelSim-Altera Software" on page 2–5.

You can choose not to invoke `-L` in the `vsim` command for VHDL designs if you have already included the libraries' mapped name in your design files or sub-files.



☞ You do not have to set the value (minimum, average, maximum) for the ***.sdo** file because the Quartus II EDA Netlist Writer generates the ***.sdo** file using the same value for the triplet (minimum, average, and maximum timing values).

☞ If your design under test is instantiated in the testbench file under the `i1` label, the *<design instance>* should be "`i1`" (for example, **/i1=<*my design*>.sdo**).

Running the Simulation

To run a simulation, type the following commands:

```
add wave <signal name> ↵
run <time period> ↵
```

To add all signals in your testbench hierarchy, type the following command:

```
add wave * ↵
```

To run the simulation for 100 ps, type the following command:

```
run 100 ps ↵
```

Simulating the Verilog HDL Designs at the Command Line

Simulating Verilog HDL design using the ModelSim-Altera command line gives you more flexibility and control to compile the libraries, and load and simulate the Verilog HDL design files. All simulation commands are Tcl commands which can be put into the ModelSim Macros file (***.do**). Using the ***.do** file allows you to run simulation in batch mode. You only have to execute the ***.do** file and the ModelSim-Altera tool automatically executes all the commands in the ***.do** script macro file.

You can use the ModelSim-Altera command line to perform the RTL functional simulation, post-synthesis simulation, and gate-level simulation. The following sections show how to perform simulation at various levels on the ModelSim-Altera command line.

Perform RTL Functional Simulation

RTL functional simulation is typically performed to verify the syntax of the code and to check the functionality of the design. The following sections show how to perform RTL functional simulation in ModelSim-Altera for Verilog HDL designs.

The ModelSim-Altera software includes pre-compiled simulation libraries. Creating simulation libraries and compiling simulation models are not required.

Compile Testbench and Design Files into the Work Library

The following commands compile your testbench and design files into the work library at the ModelSim-Altera command prompt.

To change to the design library, type the following command:

```
cd <your_design_directory> ↵
```

(for example, **cd:/designs**)

To create the work library, type the following commands:

```
vlib work ↵
vmap work work ↵
```

To compile the testbench and design files into the work library, type the following command:

```
vlog -work work <my_testbench.v> <my_design_files.v> ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, type the following command:

```
vsim -t ps -L <library1> -L <library2> work.<my_testbench> ↵
```

The <*library1*> and <*library2*> variables are the required libraries to compile your testbench. If you have multiple libraries, use the `-L` option multiple times in the `vsim` command.

Examples of <*pre-compiled library*> are **altera_mf_ver** and **lpm_ver**. The functional RTL simulation libraries are usually required for performing RTL functional simulation. For the complete set of libraries, refer to "Pre-Compiled Simulation Libraries in the ModelSim-Altera Software" on page 2–5.

Running the Simulation

To run a simulation, type the following commands:

```
add wave <signal name> ↵
run <time period> ↵
```

To add all signals in your testbench hierarchy, type the following command:

```
add wave * ↵
```

To run the simulation for 100 ps, type the following command:

```
run 100 ps ↵
```

Perform Post-Synthesis Simulation

Post-synthesis simulation can be done to verify that functionality of the design is not lost after synthesis. You can create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the ModelSim-Altera software.

The following sections help you perform a post-synthesis simulation for a Verilog HDL design in the ModelSim-Altera software.

The ModelSim-Altera software includes pre-compiled simulation libraries. Creating simulation libraries and compiling simulation models are not required.

Compile Testbench and Verilog Output File into the Work Library

Before running post-synthesis simulation, generate post-synthesis simulation netlist files. Refer to the instructions in "Generate Post-Synthesis Simulation Netlist Files" on page 2–13.

The following instructions show how to compile your testbench and ***.vo** file into the work library using the ModelSim-Altera software.

To change to the simulation output directory, type the following command:

```
cd <simulation_output_directory> ↵
```

(for example, **cd:/designs/modelsim/simulation**)

This directory contains the ***.vo** file, which is generated by the netlist writer.

To create the work library, type the following commands:

```
vlib work ↵
vmap work work ↵
```

To compile the testbench and ***.v** files into the work library, type the following command:

```
vlog -work work <my_testbench.v> <my_design_netlists.vo> ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, type the following command:

```
vsim -t ps -L <library1> -L <library2> work.<my_testbench> ↵
```

The <*library1*> and <*library2*> variables are the required libraries to compile your testbench. Examples of <*pre-compiled library*> are **stratixiii_ver**, **stratixii_ver**, and **stratixiigx_ver**. Gate-level libraries are usually required for performing post-synthesis simulation. If you have multiple libraries, use the `-L` option multiple times in the `vsim` command. For the complete set of libraries, refer to "Pre-Compiled Simulation Libraries in the ModelSim-Altera Software" on page 2–5.

Running the Simulation

To run a simulation, type the following commands:

```
add wave <signal name> ↵
run <time period> ↵
```

To add all signals in your testbench hierarchy, type the following command:

```
add wave * ↵
```

To run the simulation for 100 ps, type the following command:

```
run 100 ps ↵
```

Perform Gate-Level Simulation

Gate-level simulation is a very important step in ensuring that the FPGA device's functionality is still correct and meets all required timing requirements after the design was placed and routed. You can create the gate-level netlist in the Quartus II software and use the netlist to perform gate-level simulation with the ModelSim-Altera software.

The following sections help you perform a gate-level simulation for a Verilog HDL design in the ModelSim-Altera software.

The ModelSim-Altera software includes pre-compiled simulation libraries. Creating simulation libraries and compiling simulation models are not required.

Compile Testbench and Verilog Output File into the Work Library

Before running gate-level simulation, generate gate-level timing simulation netlist files. Refer to the instructions in "Generate Gate-Level Timing Simulation Netlist Files" on page 2–14.

The following instructions show how to compile your testbench and ***.vo** file into the work library using the ModelSim-Altera software.

To change to the simulation output directory, type the following command:

```
cd <simulation_output_directory> ↵
```

(for example, **cd:/designs/modelsim/simulation**)

This directory contains the ***.vho** file, which is generated by the netlist writer.

To create the work library, type the following commands:

```
vlib work ↵
vmap work work ↵
```

To compile the testbench and VHDL output (***.vho**) files into the work library, type the following command:

```
vlog -work work <my_testbench.v> <my_design_netlists.vo> ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

When simulating in Verilog HDL, the ***.sdo** file does not have to be manually specified. In the **$sdf_annotate** task, when the Quartus II software generates the ***.vo** file, the ModelSim-Altera software looks for the ***.sdo** file in the folder in which the VSIM was run. If your ***.sdo** file is not in the same directory in which you ran VSIM, copy the ***.sdo** file into your current directory.

To load a design, type the following command:

```
vsim -t ps -L <library1> -L <library2> work.<my_testbench> ↵
```

The <*library1*> and <*library2*> variables are the required libraries to compile your testbench. Examples of pre-compiled libraries are **stratixiii_ver**, **stratixii_ver**, and **stratixiigx_ver**. Gate-level libraries are usually required for performing gate-level timing simulation. If you have multiple libraries, use the `-L` option multiple times in the `vsim` command. For the complete set of libraries, refer to "Pre-Compiled Simulation Libraries in the ModelSim-Altera Software" on page 2–5.

Running the Simulation

To run a simulation, type the following commands:

```
add wave <signal name> ↵
run <time period> ↵
```

To add all signals in your testbench hierarchy, type the following command:

```
add wave * ↵
```

To run the simulation for 100 ps, type the following command:

```
run 100 ps ↵
```

Perform Simulation Using the ModelSim Software

Simulation of Verilog HDL or VHDL designs with ModelSim software can be done at various levels to verify designs from different aspects. Simulation is divided into three categories: RTL functional simulation, post-synthesis simulation, and gate-level simulation. Simulation helps you verify your designs and debug them against any possible errors in the designs.

You can perform the simulation through the GUI or command line. The following sections provide step-by-step instructions to perform the simulation through the GUI and the command line. You can proceed to the specific section that meets your needs.

For high-speed simulation, you must select **ps** in the **Resolution** list for your simulator resolutions. If you choose slower than **ps**, the high-speed simulation may fail.

Simulating the VHDL Designs Using the GUI

Simulating VHDL design using the ModelSim GUI is user-friendly. You do not have to remember the commands to compile the libraries or load and simulate the VHDL design files. You can use the ModelSim GUI to perform RTL functional simulation, post-synthesis simulation, and gate-level timing simulation. The following sections show how to perform simulation at various levels through the ModelSim GUI.

Perform RTL Functional Simulation

RTL functional simulation is typically performed to verify the syntax of the code and to check the functionality of the design. The following sections show how to perform RTL functional simulation in ModelSim for VHDL designs.

Use the following instructions to perform an RTL functional simulation for VHDL designs in the ModelSim software.

Create Simulation Libraries

Simulation libraries are required to simulate a design that contains an Altera primitive, LPM function, or Altera megafunction. Depending on your design, you must create the required simulation libraries and link them to your design correctly.

To change to the design directory, perform the following steps:

1. In the ModelSim software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your designs are located.
3. Click **OK**.

If you are not using the EDA Simulation Library Compiler, perform the following steps to create the simulation library. (If you are using this utility, you can skip these steps.)

1. In the ModelSim software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name of the newly created library.

For example, the library name for Altera megafunctions is **altera_mf**, and the library name for LPM is **lpm**. To see all the functional simulation library files, refer to "RTL Functional Simulation Libraries" on page 2–6.

4. Click **OK**.

Compile Simulation Models into Simulation Libraries

If you are not using the Altera Simulation Library Compiler, perform the following steps to compile simulation models into simulation libraries. (If you are using this utility, you can skip these steps.)

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library that you created (for example, **altera_mf**, **lpm**).
3. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** and add the necessary simulation model files to your project. Select the simulation model files and click **Compile**.

The **altera_mf_components.vhd** and **altera_mf.vhd** model files should be compiled into the **altera_mf** library. The **220pack.vhd** and **220model.vhd** model files should be compiled into the **lpm** library.

4. Repeat step 2 and step 3 to compile other simulation models.
5. Click **Done**.

Compile Testbench and Design Files into the Work Library

The following instructions show you how to compile your testbench and design files into the work library using the ModelSim GUI.

To create the work library, perform the following steps:

1. In the ModelSim software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name `Work` in the text box.
4. Click **OK**.

To compile the testbench and design files into the work library, perform the following steps:

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library **Work**. The design files and testbench file should be compiled into the **Work** library.
3. Select the design files and the testbench file and click **Compile**.
4. Click **Done**.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following steps:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. In the **Start Simulation** dialog box, click the **Design** tab. In the **Resolution** list, select **ps**.
3. In the **Library** list, select and expand the **Work** library.
4. Select the top-level design unit (your testbench).
5. In the **Resolution** list, select **ps**.
6. For VHDL designs, if you have not included the libraries' mapped name in your design files or sub-files, perform the following steps:
   a. Click the **Libraries** tab.
   b. In the **Search Libraries** (-L) text box, click the **Add** button.
   c. Browse to the required simulation library that you previously compiled (for example, **altera_mf, lpm**, or **altera**).
   d. Click **OK** to add the libraries to the **Search Libraries** (-L) text box.
7. Click **OK**.

Running the Simulation

To run a simulation, perform the following steps:

1. On the View menu, point to **Debug Windows** and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to **Debug Windows** and click **Wave**.
3. Drag signals to monitor from the Objects window and drop them into the Wave window.
4. On the Processing menu, point to **Run** and click **Run 100 ps** to run the simulation for 100 ps.

Perform Post-Synthesis Simulation

Post-synthesis simulation can be performed to verify that functionality of the design is not lost after synthesis. You can create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the ModelSim software.

Before running post-synthesis simulation, generate post-synthesis simulation netlist files. Refer to the instructions in "Generate Post-Synthesis Simulation Netlist Files" on page 2–13.

The following sections help you perform a post-synthesis simulation for a VHDL design in the ModelSim software.

Create Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-synthesis primitives. Depending on the device family you are using, you must create the required simulation libraries and link them to your design correctly.

To change to the simulation output directory, perform the following steps:

1. In the ModelSim software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your testbench or ***.vho** file is located. By default, the ***.vho** file is located in *<project directory>/simulation/modelsim*.
3. Click **OK**.

If you are not using the EDA Simulation Library Compiler, perform the following steps to create the simulation library. (If you are using this utility, you can skip these steps.)

1. In the ModelSim software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name of the newly created library.

For example, the library name for Stratix III family is **stratixiii**. To see all the gate-level timing simulation library files, refer to "Gate-Level Simulation Library Files" on page 2–10.

4. Click **OK**.

Compile Simulation Models into Simulation Libraries

If you are not using the Altera Simulation Library Compiler, perform the following steps to compile simulation models into simulation libraries. (If you are using this utility, you can skip these steps.)

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library that you created (for example, **stratixiii**).
3. Browse to the *<Quartus II installation directory>***/eda/sim_lib** and add the necessary simulation model files to your project. Select the simulation model files and click **Compile**.

The **stratixiii_components.vhd** and **stratixiii.vhd** model files are compiled into the **stratixiii** library.

4. Repeat step 2 and step 3 to compile other simulation models.
5. Click **Done**.

Compile Testbench and VHDL Output File into the Work Library

The following instructions show you how to compile your testbench and ***.vho** file into the work library using the ModelSim GUI.

To create the work library, perform the following steps:

1. In the ModelSim software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name `Work` in the text box.
4. Click **OK**.

To compile the testbench file and ***.vho** file into the work library, perform the following steps:

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library **Work**. The testbench and ***.vho** file should be compiled into the **Work** library.
3. Select the testbench and ***.vho** file design files and click **Compile**.
4. Click **Done**.

Loading the Design

To load a design, perform the following steps:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. In the **Start Simulation** dialog box, click the **Design** tab. In the **Resolution** list, select **ps**.
3. In the **Library** list, select and expand the **Work** library.
4. Select the top-level design unit (your testbench).
5. In the **Resolution** list, select **ps**.
6. For VHDL designs, if you have not included the libraries' mapped name in your design files or sub-files, perform the following steps:
   a. Click the **Libraries** tab.
   b. In the **Search Libraries** (-L) text box, click the **Add** button.
   c. Browse to the required simulation library that you previously compiled (for example, **stratixii, stratixiii**, or **cycloneiii**).
   d. Click **OK** to add the libraries to the **Search Libraries** (-L) text box.
7. Click **OK**.

Running the Simulation

To run a simulation, perform the following steps:

1. On the View menu, point to **Debug Windows** and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to **Debug Windows** and click **Wave**.
3. Drag signals to monitor from the Objects window and drop them into the Wave window.

4. On the Processing menu, point to **Run** and click **Run 100 ps** to run the simulation for 100 ps.

Perform Gate-Level Simulation

Gate-level simulation is a very important step in ensuring that the FPGA device's functionality is still correct and meets all required timing requirements after the design was placed and routed. You can create the gate-level netlist in the Quartus II software and use the netlist to perform gate-level simulation with the ModelSim software.

Before running gate-level simulation, generate gate-level timing simulation netlist files. Refer to the instructions in "Generate Gate-Level Timing Simulation Netlist Files" on page 2–14.

The following sections help you perform a gate-level simulation for a VHDL design in the ModelSim software.

Create Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-synthesis primitives. Depending on the device family you are using, you must create the required simulation libraries and link them to your design correctly.

To change to the simulation output directory, perform the following steps:

1. In the ModelSim software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your testbench or ***.vho** file is located. By default, the ***.vho** file is located in <*project directory*>*/simulation/modelsim*>.
3. Click **OK**.

If you are not using the EDA Simulation Library Compiler, perform the following steps to compile simulation models into simulation libraries. (If you are using this utility, you can skip these steps.)

1. In the ModelSim software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name of the newly created library.

For example, the library name for Stratix III family is **stratixiii**. To see all gate-level timing simulation library files, refer to "Gate-Level Simulation Libraries" on page 2–6.

4. Click **OK**.

Compile Simulation Models into Simulation Libraries

If you are not using the Altera Simulation Library Compiler, perform the following steps to compile simulation models into simulation libraries. (If you are using this utility, you can skip these steps.)

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library that you created (for example, **stratixiii**).

3. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** and add the necessary simulation model files to your project. Select the simulation model files and click **Compile**.

The **stratixiii_components.vhd** and **stratixiii.vhd** model files should be compiled into the **stratixiii** library.

4. Repeat step 2 and step 3 to compile other simulation models.
5. Click **Done**.

Compile Testbench and Design Files into the Work Library

The following instructions show you how to compile your testbench and ***.vho** file into the work library using the ModelSim GUI.

To create the work library, perform the following steps:

1. In the ModelSim software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name `Work` in the text box.
4. Click **OK**.

To compile the testbench file and ***.vho** file into the work library, perform the following steps:

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library **Work**. The testbench and ***.vho** file should be compiled into the **Work** library.
3. Select the testbench and ***.vho** design files and click **Compile**.
4. Click **Done**.

Loading the Design

To load a design, perform the following steps:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. Click the **SDF** tab and click **Add**.
3. In the **Add SDF Entry** dialog box, click **Browse** and select the ***.sdo** file. By default, the ***.sdo** file is located in <*project directory*>**/simulation/modelsim**.
4. In the **Apply to Region** dialog box, type in the instance path to which the ***.sdo** file is to be applied. For example, if you are using a testbench exported into the Quartus II software from a Vector Waveform File, the instance path is set to `/i1`.

   You do not have to choose from the **Delay** list because the Quartus II EDA Netlist Writer generates the ***.sdo** file using the same value for the triplet (minimum, typical, and maximum timing values).

5. Click **OK**.

6. Click the **Design** tab. In the **Resolution** list, select **ps**.
7. In the **Library** list, select the **Work** library.
8. In the **Start Simulation** dialog box, expand the **Work** library.
9. Select the top-level design unit (your testbench).
10. In the **Resolution** list, select **ps**.
11. For VHDL designs, if you have not included the libraries' mapped name in your design files or sub-files, perform the following steps:
    a. Click the **Libraries** tab.
    b. In the **Search Libraries** (-L) text box, click the **Add** button.
    c. Browse to the required simulation library that you previously compiled (for example, **stratixii, stratixiii**, or **cycloneiii**).
    d. Click **OK** to add the libraries to the **Search Libraries** (-L) text box.
12. Click **OK**.

Running the Simulation

To run a simulation, perform the following steps:

1. On the View menu, point to **Debug Windows** and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to **Debug Windows** and click **Wave**.
3. Drag signals to monitor from the Objects window and drop them into the Wave window.
4. On the Processing menu, point to **Run** and click **Run 100 ps** to run the simulation for 100 ps.

Simulating the Verilog HDL Designs Using the GUI

Simulating Verilog HDL design using the ModelSim GUI is user-friendly. You do not have to remember the commands to compile the libraries or load and simulate the Verilog HDL design files. You can use the ModelSim GUI to perform RTL functional simulation, post-synthesis simulation, and gate-level timing simulation. The following sections show how to perform simulation at various levels through the ModelSim GUI.

Perform RTL Functional Simulation

RTL functional simulation is typically performed to verify the syntax of the code and to check the functionality of the design. The following sections show how to perform RTL functional simulation in ModelSim for Verilog HDL designs.

Use the following instructions to perform an RTL functional simulation for Verilog HDL designs in the ModelSim software.

Create Simulation Libraries

Simulation libraries are required to simulate a design that contains an Altera primitive, LPM function, or Altera megafunction. Depending on your design, you must create the required simulation libraries and link them to your design correctly.

If you are not using the Altera Simulation Library Compiler, perform the following steps to compile simulation models into simulation libraries. (If you are using this utility, you can skip these steps.)

1. In the ModelSim software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your designs are located.
3. Click **OK**.

To create the simulation library, perform the following steps:

1. In the ModelSim software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name of the newly created library.

For example, the library name for Altera megafunctions is **altera_mf_ver**, and the library name for LPM is **lpm_ver**. To see all the functional simulation library files, refer to "RTL Functional Simulation Libraries" on page 2–6.

4. Click **OK**.

Compile Simulation Models into Simulation Libraries

If you are not using the EDA Simulation Library Compiler, perform the following steps to compile simulation models into simulation libraries. (If you are using this utility, you can skip these steps.)

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library that you created (for example, **altera_mf_ver** or **lpm_ver**).
3. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** and add the necessary simulation model files to your project. Select the simulation model files and click **Compile**.

The **altera_mf.v** model files should be compiled into the **altera_mf_ver** library. The **220model.v** model files should be compiled into the **lpm_ver** library.

4. Repeat step 2 and step 3 to compile other simulation models.
5. Click **Done**.

Compile Testbench and Design Files into the Work Library

The following instructions show you how to compile your testbench and design files into the work library using the ModelSim GUI.

To change to the design directory, perform the following steps:

1. In the ModelSim software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your designs are located.
3. Click **OK**.

To compile the testbench and design files into the work library, perform the following steps:

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library **Work**. The design files and testbench file should be compiled into the **Work** library.
3. Select the design files and the testbench file and click **Compile**.
4. Click **Done**.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following steps:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. Click the **Design** tab. In the **Resolution** list, select **ps**.
3. In the **Library** list, select and expand the **Work** library.
4. Select the top-level design unit (your testbench).
5. In the **Resolution** list, select **ps**.
6. Click the **Libraries** tab.
7. In the **Search Libraries** (-L) text box, click the **Add** button to browse to the required simulation library that you previously compiled (for example, **altera_mf_ver**, **lpm_ver**, or **altera_ver**) and click **OK** to add them into the **Search Libraries** (-L) text box.
8. Click **OK**.

Running the Simulation

To run a simulation, perform the following steps:

1. On the View menu, point to **Debug Windows** and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to **Debug Windows** and click **Wave**.
3. Drag signals to monitor from the Objects window and drop them into the Wave window.
4. On the Processing menu, point to **Run** and click **Run 100 ps** to run the simulation for 100 ps.

Perform Post-Synthesis Simulation

Post-synthesis simulation can be performed to verify that functionality of the design is not lost after synthesis. You can create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the ModelSim software.

Before running post-synthesis simulation, generate post-synthesis simulation netlist files. Refer to the instructions in "Generate Post-Synthesis Simulation Netlist Files" on page 2–13.

The following sections help you perform a post-synthesis simulation for a Verilog HDL design in the ModelSim software.

Create Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-synthesis primitives. Depending on the device family you are using, you must create the required simulation libraries and link them to your design correctly.

To change to the simulation output directory, perform the following steps:

1. In the ModelSim software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your testbench or ***.vo** file is located. By default, the ***.vo** file is located in <*project directory*>**/simulation/modelsim**.
3. Click **OK**.

To create the simulation library, perform the following steps:

1. In the ModelSim software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name of the newly created library.

For example, the library name for Stratix III family is **stratixiii**. To see all the gate-level timing simulation library files, refer to "Gate-Level Simulation Library Files" on page 2–10.

4. Click **OK**.

Compile Simulation Models into Simulation Libraries

If you are not using the EDA Simulation Library Compiler, perform the following steps to compile simulation models into simulation libraries. (If you are using this utility, you can skip these steps.)

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library that you created (for example, **stratixiii_ver**).
3. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** and add the necessary simulation model files to your project. Select the simulation model files and click **Compile**.

The **stratixiii_atoms.v** model files should be compiled into the **stratixiii_ver** library.

4. Repeat step 2 and step 3 to compile other simulation models, if needed.
5. Click **Done**.

Compile Testbench and Verilog Output File into the Work Library

The following instructions show you how to compile your testbench and ***.vo** into the work library using the ModelSim GUI.

To create the work library, perform the following steps:

1. In the ModelSim software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name `Work` in the text box.
4. Click **OK**.

To compile the testbench file and ***.vo** file into the work library, perform the following steps:

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library **Work**. The testbench and ***.vo** file should be compiled into the **Work** library.
3. Select the testbench and ***.vo** design files, and click **Compile**.
4. Click **Done**.

Loading the Design

To load a design, perform the following steps:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. Click the **Design** tab. In the **Resolution** list, select **ps**.
3. In the **Library** list, select the **Work** library.
4. In the **Start Simulation** dialog box, expand the **Work** library.
5. Select the top-level design unit (your testbench).
6. In the **Resolution** list, select **ps**.
7. Click the **Libraries** tab.
8. In the **Search Libraries** (-L) text box, click the **Add** button to browse to the required simulation library that you previously compiled (for example, **stratixiii_ver** or **stratixiiigx_ver**) and click **OK** to add them into the **Search Libraries** (-L) text box.
9. Click **OK**.

Running the Simulation

To run a simulation, perform the following steps:

1. On the View menu, point to **Debug Windows** and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to **Debug Windows** and click **Wave**.
3. Drag signals to monitor from the Objects window and drop them into the Wave window.
4. On the Processing menu, point to **Run** and click **Run 100 ps** to run the simulation for 100 ps.

Perform Gate-Level Simulation

Gate-level simulation is a very important step in ensuring that the FPGA device's functionality is still correct and meets all required timing requirements after the design was placed and routed. You can create the gate-level netlist in the Quartus II software and use the netlist to perform gate-level simulation with the ModelSim software.

Before running gate-level simulation, generate gate-level timing simulation netlist files. Refer to the instructions in "Generate Gate-Level Timing Simulation Netlist Files" on page 2–14.

The following sections help you perform a gate-level simulation for a Verilog HDL design in the ModelSim software.

Create Simulation Libraries

Simulation libraries are required to simulate a design that contains an Altera primitive, LPM function, or Altera megafunction. Depending on your design, you must create the required simulation libraries and link them to your design correctly.

To change to the design directory, perform the following steps:

1. In the ModelSim software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your designs are located.
3. Click **OK**.

To change to the simulation output directory, perform the following steps:

1. In the ModelSim software, on the File menu, click **Change Directory**. The **Choose Folder** dialog box appears.
2. Browse to the directory where your testbench or ***.vo** file is located. By default, the ***.vo** file is located in <*project directory*>**/simulation/modelsim**.
3. Click **OK**.

If you are not using the EDA Simulation Library Compiler, perform the following steps to create the simulation library. (If you are using the utility, you can skip these steps.)

1. In the ModelSim software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.

3. In the **Library Name** box, type the library name of the newly created library.

For example, the library name for Stratix III family is **stratixiii**. To see all the gate-level timing simulation library files, refer to "Gate-Level Simulation Library Files" on page 2–10.

4. Click **OK**.

Compile Simulation Models into Simulation Libraries

If you are not using the Altera Simulation Library Compiler, perform the following steps to compile simulation models into simulation libraries. (If you are using this utility, you can skip these steps.)

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library that you created (for example, **stratixiii_ver**).
3. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** and add the necessary simulation model files to your project. Select the simulation model files and click **Compile**.

The **stratixiii_atoms.v** model files should be compiled into the **stratixiii_ver** library.

4. Repeat step 2 and step 3 to compile other simulation models, if needed.
5. Click **Done**.

Compile Testbench and Verilog Output File into the Work Library

The following instructions show you how to compile your testbench and ***.vo** file into the work library using the ModelSim GUI.

To create the work library, perform the following steps:

1. In the ModelSim software, on the File menu, point to **New** and click **Library**. The **Create a New Library** dialog box appears.
2. Select a new library and a logical mapping to it.
3. In the **Library Name** box, type the library name `Work` in the text box.
4. Click **OK**.

To compile the testbench file and Verilog HDL output file (***.vo**) into the work library, perform the following steps:

1. On the Compile menu, click **Compile**. The **Compile Source Files** dialog box appears.
2. Select the library **Work**. The testbench and ***.vo** file should be compiled into the **Work** library.
3. Select the testbench and ***.vo** design files and click **Compile**.
4. Click **Done**.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

When simulating in Verilog HDL, the ***.sdo** file does not have to be manually specified. In the **$sdf_annotate** task, when the Quartus II software generates the ***.vo** file, the ModelSim-Altera software looks for the ***.sdo** file in the folder in which the VSIM was run. If your ***.sdo** file is not in the same directory in which you ran VSIM, copy the ***.sdo** file into your current directory.

To load a design, perform the following steps:

1. On the Simulate menu, click **Start Simulation**. The **Start Simulation** dialog box appears.
2. Click the **Design** tab. In the **Resolution** list, select **ps**.
3. In the **Library** list, select and expand the **Work** library.
4. Select the top-level design unit (your testbench).
5. In the **Resolution** list, select **ps**.
6. Click the **Libraries** tab.
7. In the **Search Libraries** (-L) text box, click the **Add** button to browse to the required simulation library that you previously compiled (for example, **stratixiii_ver** or **stratixiiigx_ver**) and click **OK** to add them into the **Search Libraries** (-L) text box.
8. Click **OK**.

Running the Simulation

To run a simulation, perform the following steps:

1. On the View menu, point to **Debug Windows** and click **Objects**. This command displays all objects in the current scope.
2. On the View menu, point to **Debug Windows** and click **Wave**.
3. Drag signals to monitor from the Objects window and drop them into the Wave window.
4. On the Processing menu, point to **Run** and click **Run 100 ps** to run the simulation for 100 ps.

Simulating the VHDL Designs at the Command Line

Simulating VHDL design using the ModelSim command line gives you more flexibility and control to compile the libraries and load and simulate the VHDL design files. All simulation commands are Tcl commands which can be put into the ModelSim Macros file (***.do**). Using the ***.do** file allows you to run simulation in batch mode. You only have to execute the ***.do** file and the ModelSim tool automatically executes all the commands in the ***.do** script macro file.

You can use the ModelSim command line to perform RTL functional simulation, post-synthesis simulation, and gate-level simulation. The following sections show how to perform simulation at various levels through the ModelSim command line.

Perform RTL Functional Simulation

RTL functional simulation is typically performed to verify the syntax of the code and to check the functionality of the design. The following sections show how to perform RTL functional simulation in ModelSim for VHDL designs.

Create Simulation Libraries

Simulation libraries are required to simulate a design that contains an Altera primitive, LPM function, or Altera megafunction. Depending on your design, you must create the required simulation libraries and link them to your design correctly.

To change to the design library, type the following command:

```
cd <your_design_directory> ↵
```

(for example, **cd:/designs**)

To create the simulation libraries, type the following commands:

```
vlib <library_name> ↵
vmap <logical_library_name> <library_name> ↵
```

For example, the library name for Altera megafunction is **altera_mf**, and the library name for LPM is **lpm**. To see all the functional simulation library files, refer to "RTL Functional Simulation Library Files" on page 2–9.

To create simulation libraries for **altera_mf**, **lpm**, and **altera**, type the following commands:

```
vlib altera_mf ↵
vmap altera_mf altera_mf ↵
vlib lpm ↵
vmap lpm lpm ↵
vlib altera ↵
vmap altera altera ↵
```

Compile Simulation Models into Simulation Libraries

To compile simulation models into simulation libraries, type the following command:

```
vcom -work <simulation_library> <Quartus II installation directory> \
/eda/sim_lib/<simulation_library_files> ↵
```

For example, the **altera_mf_components.vhd** and **altera_mf.vhd** model files should be compiled into the **altera_mf** library. The **220pack.vhd** and **220model.vhd** model files should be compiled into the **lpm** library.

Use Example 2–1 to compile the simulation model files to the simulation libraries for **altera_mf, lpm,** and **altera**.

Example 2–1.

```
vcom -work altera_mf <Quartus II installation directory> \
/eda/sim_lib/altera_mf_components.v <Quartus II installation directory> \
/eda/sim_lib/altera_mf.vhd ↵

vcom -work lpm <Quartus II installation directory>/eda/sim_lib/220model.vhd \
<Quartus II installation dir>/eda/sim_lib/220model.vhd ↵

vcom -work altera <Quartus II installation directory> \
/eda/sim_lib/altera_primitives_components.vhd \
<Quartus II installation directory>/eda/sim_lib/altera_primitives.vhd ↵
```

If you are using the EDA Simulation Library Compiler, skip the steps for creating and compiling the simulation libraries.

Compile Testbench and Design Files into the Work Library

The following commands show how to compile your testbench and design files into the work library using the ModelSim software.

To create the work library, type the following commands:

```
vlib work ↵
vmap work work ↵
```

To compile the testbench and design files into the work library, type the following command:

```
vcom -work work <my_testbench.vhd> <my_design_files.vhd> ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, type the following command:

```
vsim -t ps -L <compiled-library1> -L <compiled-library2> work.<my_testbench> ↵
```

The *<compiled-library1>* and *<compiled-library2>* variables are the libraries you compiled previously (for example, **altera_mf** or **lpm**) that are required to compile your testbench. RTL libraries are usually required for performing RTL simulation. If you have multiple libraries, use the `-L` option for each library in the `vsim` command.

Running the Simulation

To run a simulation, type the following commands:

```
add wave <signal name> ↵
run <time period> ↵
```

To add all signals in your testbench hierarchy, type the following command:

```
add wave * ↵
```

To run the simulation for 100 ps, type the following command:

```
run 100 ps ↵
```

Perform Post-Synthesis Simulation

Post-synthesis simulation can be performed to verify that functionality of the design is not lost after synthesis. You can create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the ModelSim software.

Before running post-synthesis simulation, generate post-synthesis simulation netlist files. Refer to the instructions in "Generate Post-Synthesis Simulation Netlist Files" on page 2–13.

The following sections help you perform a post-synthesis simulation for a VHDL design in the ModelSim software.

Create Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-synthesis primitives. Depending on the device family you are using, you must create the required simulation libraries and link them to your design correctly.

To change to the simulation output directory, type the following command:

```
cd <simulation_output_directory> ↵
```

(for example, **cd:/designs/modelsim/simulation**)

To create the simulation libraries, type the following commands:

```
vlib <library_name> ↵
vmap <logical_library_name> <library_name> ↵
```

For example, the library name for the Stratix III family is **stratixiii**. To see all the gate-level timing simulation library files, refer to "Gate-Level Simulation Libraries" on page 2–6.

To create simulation libraries for **stratixiii**, type the following commands:

```
vlib stratixiii ↵
vmap stratixiii stratixiii ↵
```

Compile Simulation Models into Simulation Libraries

To compile simulation models into simulation libraries, type the following command:

```
vcom -work <simulation_library> <Quartus II installation directory> \
/eda/sim_lib/<simulation_library_files> ↵
```

For example, the **stratixiii_atoms_components.vhd** and **stratixiii_atoms.vhd** model files should be compiled into the **stratixiii** library.

Use Example 2–2 to compile the simulation model files to the simulation libraries for **stratixiii**:

Example 2–2.

```
vcom -work stratixiii <Quartus II installation directory> \
/eda/sim_lib/stratixiii_atoms_components.vhd <Quartus II installation directory> \
/eda/sim_lib/stratixiii_atoms.vhd ↵
```

If you are using the EDA Simulation Library Compiler, skip the steps for creating and compiling the simulation libraries.

Compile Testbench and VHDL Output Files into the Work Library

The following commands show how to compile your testbench and ***.vho** file into the work library using the ModelSim software.

To create the work library, type the following commands:

```
vlib work ↵
vmap work work ↵
```

To compile the testbench and ***.vho** file into the work library, type the following command:

```
vcom -work work <my_testbench.vhd> <my_design_files.vho> ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, type the following command:

```
vsim -t ps -L <compiled-library1> -L <compiled-library2> work.<my_testbench> ↵
```

The <*compiled-library1*> and <*compiled-library2*> variables are the libraries you compiled previously (for example, **stratixiii** or **cycloneiii**) that are required to compile your testbench. Gate-level libraries are usually required for performing post-synthesis simulation. If you have multiple libraries, use the `-L` option for each library in the `vsim` command.

Running the Simulation

To run a simulation, type the following commands:

```
add wave <signal name> ↵
run <time period> ↵
```

To add all signals in your testbench hierarchy, type the following command:

```
add wave * ↵
```

To run the simulation for 100 ps, type the following command:

```
run 100 ps ↵
```

Perform Gate-Level Simulation

Gate-level simulation is a very important step in ensuring that the FPGA device's functionality is still correct and meets all required timing requirements after the design was placed and routed. You can create the gate-level netlist in the Quartus II software and use the netlist to perform gate-level simulation with the ModelSim software.

Before running gate-level simulation, generate gate-level timing simulation netlist files. Refer to the instructions in "Generate Gate-Level Timing Simulation Netlist Files" on page 2–14.

The following sections help you perform a gate-level simulation for a VHDL design in the ModelSim software.

Create Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-synthesis primitives. Depending on the device family that you are using, you must create the required simulation libraries and link them to your design correctly.

To change to the simulation output directory, type the following command:

```
cd <simulation_output_directory> ↵
```

(for example, **cd:/designs/modelsim/simulation**)

To create the simulation libraries, type the following commands:

```
vlib <library_name> ↵
vmap <logical_library_name> <library_name> ↵
```

For example, the library name for the Stratix III family is **stratixiii**. To see all the gate-level timing simulation library files, refer to "Gate-Level Simulation Libraries" on page 2–6.

To create simulation libraries for **stratixiii**, type the following commands:

```
vlib stratixiii ↵
vmap stratixiii stratixiii ↵
```

Compile Simulation Models into Simulation Libraries

To compile simulation models into simulation libraries, type the following command:

```
vcom -work <simulation_library> <Quartus II installation directory> \
/eda/sim_lib/<simulation_library_files> ↵
```

For example, the **stratixiii_atoms_components.vhd** and **stratixiii_atoms.vhd** model files should be compiled into the **stratixiii** library.

Use Example 2–3 to compile the simulation model files to the simulation libraries for **stratixiii**.

Example 2–3.

```
vcom -work stratixiii <Quartus II installation directory> \
/eda/sim_lib/stratixiii_atoms_components.vhd <Quartus II installation directory> \
/eda/sim_lib/stratixiii_atoms.vhd ↵
```

Be sure that the user-compiled libraries are stored using the same path as the design and testbench files you want to compile.

Compile Testbench and VHDL Output Files into the Work Library

The following commands show how to compile your testbench and ***.vho** file into the work library using the ModelSim GUI.

To create the work library, type the following commands:

```
vlib work ↵
vmap work work ↵
```

To compile the testbench and ***.vho** file into the work library, type the following command:

```
vcom -work work <my_testbench.vht> <my_design_files.vho>  ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, type the following command:

```
vsim -t ps -sdftyp <design instance> = <path to *.sdo file> -L <compiled-library1> \
-L <compiled-library2> work.<my_testbench> ↵
```

The *<compiled-library1>* and *<compiled-library2>* variables are the libraries you compiled previously (for example, **stratixiii** or **cycloneiii**) that are required to compile your testbench. Gate-level libraries are usually required for performing gate-level simulation. If you have multiple libraries, use the `-L` option for each library in the `vsim` command.

You do not have to set the value (minimum, average, maximum) for the ***.sdo** file because the Quartus II EDA Netlist Writer generates the ***.sdo** file using the same value for the triplet (minimum, average, and maximum timing values).

If your design under test is instantiated in the testbench file under the `i1` label, the *<design instance>* should be "`i1`" (for example, `/i1=`*<my design>***.sdo**).

Running the Simulation

To run a simulation, type the following commands:

```
add wave <signal name> ↵
run <time period> ↵
```

To add all signals in your testbench hierarchy, type the following command:

```
add wave * ↵
```

To run the simulation for 100 ps, type the following command:

```
run 100 ps ↵
```

Simulating the Verilog HDL Designs at the Command Line

Simulating Verilog HDL design using the ModelSim command line gives you more flexibility and control to compile the libraries, and load and simulate the Verilog HDL design files. All simulation commands are Tcl commands which can be put into the ***.do** file. Using the ***.do** file allows you to run simulation in batch mode. You only have to execute the ***.do** file and the ModelSim tool automatically executes all the commands in the ***.do** script macro file.

You can use the ModelSim command line to perform RTL functional simulation, post-synthesis simulation, and gate-level simulation. The following sections show how to perform simulation at various levels through the ModelSim command line.

Perform RTL Functional Simulation

RTL functional simulation is typically performed to verify the syntax of the code and to check the functionality of the design. The following sections show how to perform RTL functional simulation in ModelSim for Verilog HDL designs.

Create Simulation Libraries

Simulation libraries are required to simulate a design that contains an Altera primitive, LPM function, or Altera megafunction. Depending on your design, you must create the required simulation libraries and link them to your design correctly.

To change to the design library, type the following command:

```
cd <your_design_directory> ↵
```

(for example, **cd:/designs**)

To create the simulation libraries, type the following commands:

```
vlib <library_name> ↵
vmap <logical_library_name> <library_name> ↵
```

For example, the library name for Altera megafunction is **altera_mf_ver**, and the library name for LPM is **lpm_ver**. To see all the functional simulation library files, refer to "RTL Functional Simulation Library Files" on page 2–9.

To create simulation libraries for **altera_mf**, **lpm_ver**, and **altera_ver**, type the following commands:

```
vlib altera_mf_ver ↵
vmap altera_mf altera_mf_ver ↵
vlib lpm_ver ↵
vmap lpm lpm_ver ↵
vlib altera_ver ↵
vmap altera altera_ver ↵
```

Compile Simulation Models into Simulation Libraries

To compile simulation models into simulation libraries, type the following command:

```
vlog -work <simulation_library> <Quartus II installation directory>\
/eda/sim_lib/<simulation_library_files> ↵
```

For example, the **altera_mf.v** model files should be compiled into the **altera_mf_ver** library. The **220model.v** model files should be compiled into the **lpm_ver** library.

To compile the simulation model files to the simulation libraries for **altera_mf_ver**, **lpm_ver**, and **altera_ver**, type the following commands:

```
vlog -work altera_mf_ver \
<Quartus II installation directory>/eda/sim_lib/altera_mf.v ↵

vlog -work lpm_ver \
<Quartus II installation directory>/eda/sim_lib/220model.v ↵

vlog -work altera_ver \
<Quartus II installation directory>/eda/sim_lib/altera_primitives.v ↵
```

Compile Testbench and Design Files into the Work Library

The following commands show how to compile your testbench and design files into the work library in the ModelSim command prompt.

To create the work library, type the following commands:

```
vlib work ↵
vmap work work ↵
```

To compile the testbench and design files into the work library, type the following command:

```
vlog -work work <my_testbench.v> <my_design_files.v>  ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, type the following command:

```
vsim -t ps -L <library1> -L <library2> work.<my_testbench> ↵
```

The <*library1*> and <*library2*> variables are the required libraries to compile your testbench. If you have multiple libraries, use the `-L` option multiple times in the `vsim` command.

Running the Simulation

To run a simulation, type the following commands:

```
add wave <signal name> ↵
run <time period> ↵
```

To add all signals in your testbench hierarchy, type the following command:

```
add wave * ↵
```

To run the simulation for 100 ps, type the following the command:

```
run 100 ps ↵
```

Perform Post-Synthesis Simulation

Post-synthesis simulation can be done to verify that functionality of the design is not lost after synthesis. You can create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the ModelSim software.

Before running post-synthesis simulation, generate post-synthesis simulation netlist files. Refer to the instructions in "Generate Post-Synthesis Simulation Netlist Files" on page 2–13.

The following sections help you perform a post-synthesis simulation for a Verilog HDL design in the ModelSim software.

Create Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-synthesis primitives. Depending on the device family you are using, you must create the required simulation libraries and link them to your design correctly.

To change to the simulation output directory, type the following command:

```
cd <simulation_output_directory> ↵
```

(for example, **cd:/designs/modelsim/simulation**)

© March 2009 Altera Corporation

To create the simulation libraries, type the following commands:

```
vlib <library_name> ↵
vmap <logical_library_name> <library_name> ↵
```

For example, the library name for the Stratix III family is **stratixiii_ver**. To see all the gate-level timing simulation library files, refer to "Gate-Level Simulation Libraries" on page 2–6.

To create simulation libraries for **stratixiii_ver**, type the following commands:

```
vlib stratixiii_ver ↵
vmap stratixiii_ver stratixiii_ver ↵
```

If you are using the EDA Simulation Library Compiler, skip the steps for creating and compiling the simulation libraries.

Compile Simulation Models into Simulation Libraries

To compile simulation models into simulation libraries, type the following command:

```
vlog -work <simulation_library> <Quartus II installation dir> \
/eda/sim_lib/<simulation_library_files> ↵
```

For example, the **stratixiii_atoms.v** model files should be compiled into the **stratixiii_ver** library.

To compile the simulation model files to the simulation libraries for **stratixiii_ver**, type the following command:

```
vlog -work stratixiii_ver <Quartus II installation directory> \
/eda/sim_lib/stratixiii_atoms.v ↵
```

If you are using the Altera Simulation Library Compiler, skip the steps for creating and compiling the simulation libraries.

Compile Testbench and Verilog Output Files into the Work Library

The following commands show how to compile your testbench and ***.vo** file into the work library using the ModelSim software.

To create the work library, type the following commands:

```
vlib work ↵
vmap work work ↵
```

To compile the testbench and ***.vo** file into the work library, type the following command:

```
vlog -work work <my_testbench.vt> <my_design_netlists.vo> ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, type the following command:

```
vsim -t ps -L <library1> -L <library2> work.<my_testbench> ↵
```

The <*library1*> and <*library2*> variables are the libraries you compiled previously (for example, **stratixiii** or **stratixiigx**) that are required to compile your testbench. Gate-level libraries are usually required for performing post-synthesis simulation. If you have multiple libraries, use the `-L` option multiple times in the `vsim` command.

Running the Simulation

To run a simulation, type the following commands:

```
add wave <signal name> ↵
run <time period> ↵
```

To add all signals in your testbench hierarchy, type the following command:

```
add wave * ↵
```

To run the simulation for 100 ps, type the following the command:

```
run 100 ps ↵
```

Perform Gate-Level Simulation

Gate-level simulation is a very important step in ensuring that the FPGA device's functionality is still correct and meets all required timing requirements after the design was placed and routed. You can create the gate-level netlist in the Quartus II software and use the netlist to perform gate-level simulation with the ModelSim software.

Before running gate-level simulation, generate gate-level timing simulation netlist files. Refer to the instructions in "Generate Gate-Level Timing Simulation Netlist Files" on page 2–14.

The following sections help you perform a gate-level simulation for a Verilog HDL design in the ModelSim software.

Create Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-synthesis primitives. Depending on the device family that you are using, you must create the required simulation libraries and link them to your design correctly.

To change to the simulation output directory, type the following command:

```
cd <simulation_output_directory> ↵
```

(for example, **cd:/designs/modelsim/simulation**)

This directory contains the ***.vo** file, which is generated by the netlist writer.

To create the simulation libraries, type the following commands:

```
vlib <library_name> ↵
vmap <logical_library_name> <library_name> ↵
```

For example, the library name for the Stratix III family is **stratixiii_ver**. To see all the gate-level timing simulation library files, refer to "Gate-Level Simulation Libraries" on page 2–6.

To create simulation libraries for **stratixiii_ver**, type the following commands:

```
vlib stratixiii_ver ↵
vmap stratixiii_ver stratixiii_ver ↵
```


© March 2009 Altera Corporation

Compile Simulation Models into Simulation Libraries

To compile simulation models into simulation libraries, type the following command:

```
vlog -work <simulation_library> <Quartus II installation directory> \
/eda/sim_lib/<simulation_library_files> ↵
```

For example, the **stratixiii_atoms.v** model files should be compiled into the **stratixiii_ver** library.

Use the following example to compile the simulation model files to the simulation libraries for **stratixiii_ver**:

```
vlog -work stratixiii <Quartus II installation directory> \
/eda/sim_lib/stratixiii_atoms.v ↵
```

If you are using the EDA Simulation Library Compiler, skip the steps for creating and compiling the simulation libraries.

Compile Testbench and Verilog Output Files into the Work Library

The following commands show how to compile your testbench and ***.vo** file into the work library using the ModelSim command line.

To create the work library, type the following commands:

```
vlib work ↵
vmap work work ↵
```

To compile the testbench and ***.vo** files into the work library, type the following command:

```
vlog -work work <my_testbench.v> <my_design_netlists.vo>  ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

When simulating in Verilog HDL, the ***.sdo** file does not have to be manually specified. In the **$sdf_annotate** task, when the Quartus II software generates the ***.vo** file, the ModelSim-Altera software looks for the ***.sdo** file in the folder in which the VSIM was run. If your ***.sdo** file is not in the same directory in which you ran VSIM, copy the ***.sdo** file into your current directory.

To load a design, type the following command:

```
vsim -t ps -L <library1> -L <library2> work.<my_testbench> ↵
```

The *<library1>* and *<library2>* variables are the libraries you compiled previously (for example, **stratixiii_ver** or **stratixiigx_ver**) that are required to compile your testbench. Gate-level libraries are usually required for performing gate-level timing simulation. If you have multiple libraries, use the `-L` option multiple times in the `vsim` command.

Running the Simulation

Perform the following commands to run a simulation:

```
add wave <signal name> ↵
run <time period> ↵
```

To add all signals in your testbench hierarchy, type the following command:

```
add wave * ↵
```

To run the simulation for 100 ps, type the following the command:

```
run 100 ps ↵
```

Simulating Designs that Include Transceivers

If your design includes a Stratix GX, Stratix II GX, Stratix IV, Arria II GX, or Arria GX transceiver, you must compile additional library files to perform RTL functional or gate-level timing simulations. The following example shows how to perform simulation on designs that include transceivers in Stratix GX and Stratix II GX devices.

Performing simulation with transceivers in Arria GX and Stratix II GX are very similar. You only have to replace **stratixiigx_atoms** and **stratixiigx_hssi_atoms** model files with **arriagx_atoms** and **arriagx_hssi_atoms** model files.

For high-speed simulation, you must select **ps** in the **Resolution** list for your simulator resolutions. If you choose slower than **ps**, the high-speed simulation may fail.

Stratix GX RTL Functional Simulation

To perform an RTL functional simulation of your design that instantiates the ALTGXB megafunction, which enables the gigabit transceiver block on Stratix GX devices, compile the **stratixgx_mf** model file into the **altgxb** library.

The **stratixiigx_mf** model file references the **lpm** and **sgate** libraries. If you are using ModelSim PE/SE, you must create these libraries to perform a simulation.

Perform RTL Functional Simulation for Stratix GX in VHDL

If you are using the ModelSim-Altera software, compiling the libraries is not necessary. You can simulate the design directly by typing the commands in Example 2–4.

Example 2–4.

```
vcom -work <my design>.vhd <my testbench>.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L altgxb work.<my testbench> ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. To compile and simulate the design, type the commands in Example 2–5 at the ModelSim command prompt.

Example 2–5.

```
vcom -work altera_mf altera_mf_components.vhd altera_mf.vhd ↵
vcom -work lpm 220pack.vhd 220model.vhd ↵
vcom -work sgate sgate_pack.vhd sgate.vhd ↵
vcom -work altgxb stratixgx_mf.vhd stratixgx_mf_components.vhd ↵
vcom -work <my design>.vhd <my testbench>.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L altgxb work.<my testbench> ↵
```

Perform RTL Functional Simulation for Stratix GX in Verilog HDL

If you are using the ModelSim-Altera software, compiling the libraries is not necessary. You can simulate the design directly by typing the commands in Example 2–6.

Example 2–6.

```
vlog -work <my design>.v <my testbench>.v ↵
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L altgxb work.<my testbench> ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. To compile and simulate the design, type the commands in Example 2–7 at the ModelSim command prompt.

Example 2–7.

```
vlib work_ver ↵
vlib lpm_ver ↵
vlib altera_mf_ver ↵
vlib sgate_ver ↵
vlib altgxb_ver ↵
vlog -work lpm_ver 220model.v ↵
vlog -work altera_mf_ver altera_mf.v ↵
vlog -work sgate_ver sgate.v ↵
vlog -work altgxb_ver stratixgx_mf.v ↵
vlog -work <my design>.v <my testbench>.v ↵
vsim -L lpm_ver -L sgate_ver-L altgxb_ver work.<my testbench> ↵
```

Stratix GX Gate-Level Timing Simulation

Perform a gate-level timing simulation of your design that includes a Stratix GX transceiver by compiling the **stratixgx_atoms** and **stratixgx_hssi_atoms** model files into the **stratixgx** and **stratixgx_gxb** libraries, respectively.

The **stratixgx_hssi_atoms** model file references the **lpm** and **sgate** libraries. If you are using ModelSim PE/SE, you must create these libraries to perform a simulation.

Perform Gate-Level Timing Simulation for Stratix GX in VHDL

If you are using the ModelSim-Altera software, compiling the libraries is not necessary. You can simulate the design directly by typing the commands in Example 2–8.

Example 2–8.

```
vcom -work <my design>.vho <my testbench>.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L stratixgx -L stratixgx_gxb \
-sdftyp <design instance>=<path to .sdo file>.sdo work.<my testbench> \
-t ps - +transport_int_delays+transport_path_delays ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. To compile and simulate the design, type the commands in Example 2–9 at the ModelSim command prompt.

Example 2–9.

```
vcom -work lpm 220pack.vhd 220model.vhd ↵
vcom -work altera_mf altera_mf_components.vhd altera_mf.vhd ↵
vcom -work sgate sgate_pack.vhd sgate.vhd ↵
vcom -work stratixgx stratixgx_atoms.vhd stratixgx_components.vhd ↵
vcom -work stratixgx_gxb stratixgx_hssi_atoms.vhd \
stratixgx_hssi_components.vhd ↵
vcom -work <my design>.vho <my testbench>.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L stratixgx -L stratixgx_gxb \
-sdftyp <design instance>=<path to .sdo file>.sdo work.<my testbench> \
-t ps +transport_int_delays +transport_path_delays ↵
```

Perform Gate-Level Timing Simulation for Stratix GX in Verilog HDL

If you are using the ModelSim-Altera software, compiling the libraries is not necessary. You can simulate the design directly by typing the command in Example 2–10.

Example 2–10.

```
vlog -work <my design>.vo <my testbench>.v ↵
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixgx_ver -L \
stratixgx_gxb_ver work.<my testbench> -t ps +transport_int_delays \
+transport_path_delays ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. To compile and simulate the design, type the commands in Example 2–11 at the ModelSim command prompt.

Example 2–11.

```
vlog -work lpm_ver 220model.v ↵
vlog -work altera_mf_ver altera_mf.v ↵
vlog -work sgate_ver sgate.v ↵
vlog -work stratixgx_ver stratixgx_atoms.v ↵
vlog -work stratixgx_gxb_ver stratixgx_hssi_atoms.v ↵
vlog -work <my design>.vo <my testbench>.v ↵
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixgx_ver \
-L stratixgx_gxb_ver work.<my testbench> -t ps +transport_int_delays \
+transport_path_delays ↵
```

Stratix II GX RTL Functional Simulation

Stratix II GX function simulation is similar to Arria GX functional simulation. The following example shows only the functional simulation for designs that include transceivers in Stratix II GX devices. To simulate transceivers in Arria GX devices, you only have to replace the **stratixiigx_hssi** model file with the **arriagx_hssi** model file.

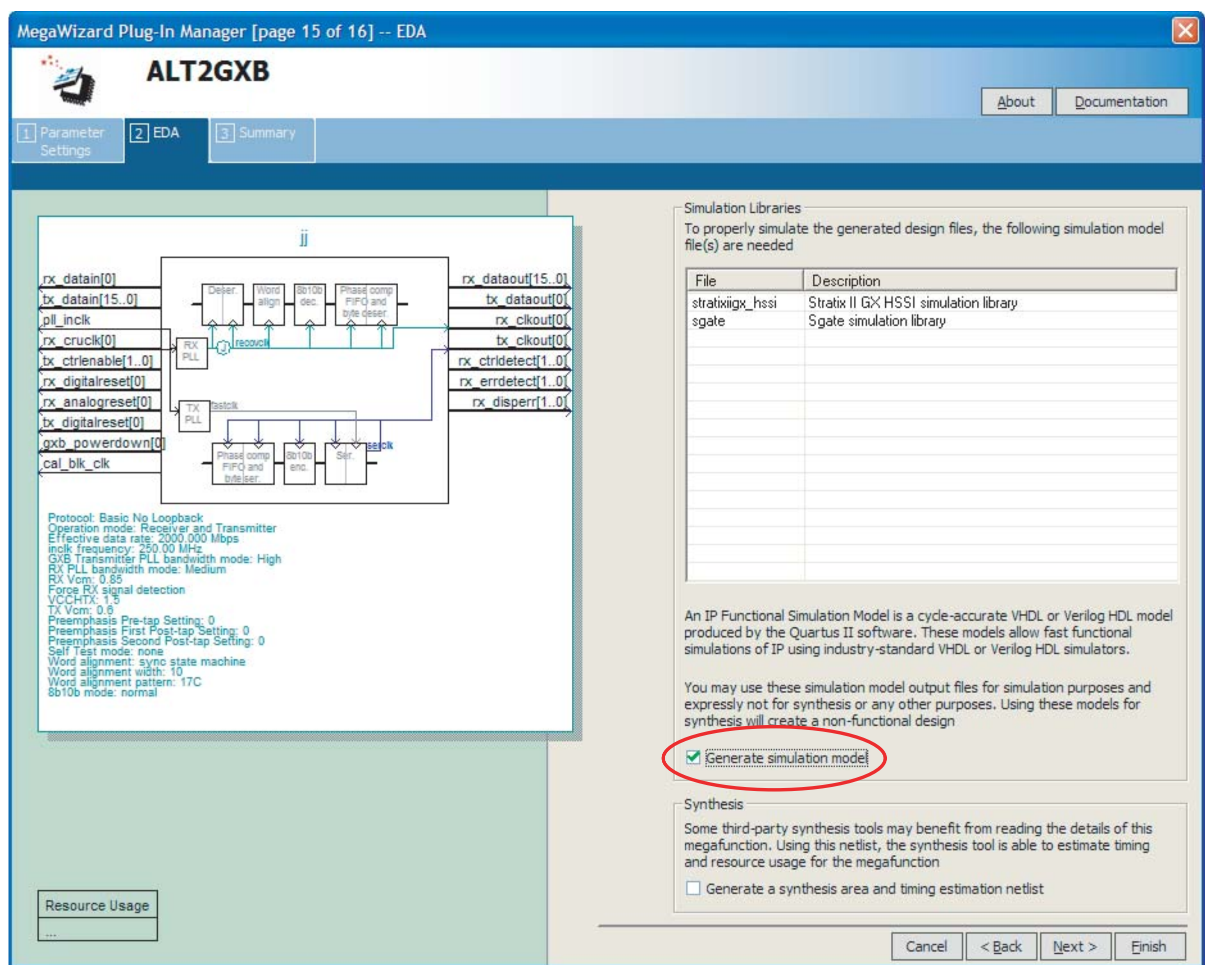
To perform a functional simulation of your design that instantiates the ALT2GXB megafunction, which enables the gigabit transceiver block on Stratix II GX devices, compile the **stratixiigx_hssi** model file into the **stratixiigx_hssi** library.

The **stratixiigx_hssi_atoms** model file references the **lpm** and **sgate** libraries. If you are using ModelSim PE/SE, you must create these libraries to perform a simulation.

Generate a functional simulation netlist by turning on **Generate Simulation Model** in the **Simulation Library** tab of the ALT2GXB MegaWizard Plug-In Manager (Figure 2–2). The <*alt2gxb entity name*>**.vho** or <*alt2gxb module name*>**.vo** is generated in the current project directory.

The Quartus II-generated ALT2GXB functional simulation library file references `stratixiigx_hssi` wysiwyg atoms.

Figure 2–2. ALT2GXB MegaWizard Plug-In Manager, Generate Simulation Model



Perform RTL Functional Simulation for Stratix II GX in VHDL

If you are using the ModelSim-Altera software, compiling the libraries is not necessary. You can simulate the design directly by typing the commands in Example 2–12.

Example 2–12.

```
vcom -work work <alt2gxb entity name>.vho ↵
vcom -work <my design>.vhd <my testbench>.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L stratixgx_hssi work.<my design> ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. To compile and simulate the design, type the commands in Example 2–13 at the ModelSim command prompt.

Example 2–13.

```
vcom -work lpm 220pack.vhd 220model.vhd ↵
vcom -work altera_mf altera_mf_components.vhd altera_mf.vhd ↵
vcom -work sgate sgate_pack.vhd sgate.vhd ↵
vcom -work stratixiigx_hssi stratixiigx_hssi_components.vhd \
stratixiigx_hssi_atoms.vhd ↵
vcom -work work <alt2gxb entity name>.vho ↵
vcom -work <my design>.vhd <my testbench>.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L stratixgx_hssi work.<my testbench> ↵
```

Perform RTL Functional Simulation for Stratix II GX in Verilog HDL

If you are using the ModelSim-Altera software, compiling the libraries is not necessary. You can simulate the design directly by typing the commands in Example 2–14.

Example 2–14.

```
vlog -work work <alt2gxb module name>.vo ↵
vlog -work <my design>.v <my testbench>.v ↵
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixgx_hssi_ver \
work.<my testbench> ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. To compile and simulate the design, type the commands in Example 2–15 at the ModelSim command prompt.

Example 2–15.

```
vlog -work lpm_ver 220model.v ↵
vlog -work altera_mf_ver altera_mf.v ↵
vlog -work sgate_ver sgate.v ↵
vlog -work stratixiigx_hssi_ver stratixiigx_hssi_atoms.v ↵
vlog -work work <alt2gxb module name>.vo ↵
vlog -work <my design>.v <my testbench>.v ↵
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixgx_hssi_ver \
work.<my testbench> ↵
```

Stratix II GX Gate-Level Timing Simulation

Stratix II GX gate-level timing simulation is similar to Arria GX gate-level timing simulation. The following example shows only the gate-level timing simulation for designs that include transceivers in Stratix II GX devices. To simulate transceivers in Arria GX devices, you only have to replace the **stratixiigx_hssi** model file with the **arriagx_hssi** model file.

To perform a gate-level timing simulation of your design that includes a Stratix II GX transceiver, compile **stratixiigx_atoms** and **stratixiigx_hssi_atoms** into the **stratixiigx** and **stratixiigx_hssi** libraries, respectively.

The **stratixiigx_hssi_atoms** model file references the **lpm** and **sgate** libraries. If you are using ModelSim PE/SE, you must create these libraries to perform a simulation.

Perform Gate-Level Timing Simulation for Stratix II GX in VHDL

If you are using the ModelSim-Altera software, compiling the libraries is not necessary. You can simulate the design directly by typing the commands in Example 2–16.

Example 2–16.

```
vcom -work <my design>.vho <my testbench>.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L stratixiigx -L stratixiigx_hssi \
-sdftyp <design instance>=<path to .sdo file>.sdo work.<my testbench> \
-t ps +transport_int_delays +transport_path_delays ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. To compile and simulate the design, type the commands in Example 2–17 at the ModelSim command prompt.

Example 2–17.

```
vcom -work lpm 220pack.vhd 220model.vhd ↵
vcom -work altera_mf altera_mf_components.vhd altera_mf.vhd ↵
vcom -work sgate sgate_pack.vhd sgate.vhd ↵
vcom -work stratixiigx stratixiigx_atoms.vhd \
stratixiigx_components.vhd ↵
vcom -work stratixiigx_hssi stratixiigx_hssi_components.vhd \
stratixiigx_hssi_atoms.vhd ↵
vcom -work <my design>.vho <my testbench>.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L stratixiigx -L stratixiigx_hssi \
-sdftyp <design instance>=<path to .sdo file>.sdo work.<my testbench> \
-t ps +transport_int_delays +transport_path_delays ↵
```

Perform Gate-Level Timing Simulation for Stratix II GX in Verilog HDL

If you are using the ModelSim-Altera software, compiling the libraries is not necessary. You can simulate the design directly by typing the commands in Example 2–18.

Example 2–18.

```
vlog -work <my design>.vo <my testbench>.v ↵
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixiigx_ver \
-L stratixiigx_hssi_ver work.<my testbench> -t ps \
+transport_int_delays +transport_path_delays ↵
```

If you are using ModelSim SE/PE, you must compile the necessary libraries before you can simulate the designs. To compile and simulate the design, type the commands in Example 2–19 at the ModelSim command prompt.

Example 2–19.

```
vlog -work lpm_ver 220model.v ↵
vlog -work altera_mf_ver altera_mf.v ↵
vlog -work sgate_ver sgate.v ↵
vlog -work stratixiigx_ver stratixiigx_atoms.v ↵
vlog -work stratixiigx_hssi_ver stratixiigx_hssi_atoms.v ↵
vlog -work <my design>.vo <my testbench>.v ↵
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixiigx_ver \
-L stratixiigx_hssi_ver work.<my testbench> -t ps \
+transport_int_delays +transport_path_delays ↵
```

Transport Delays

By default, the ModelSim software filters out all pulses that are shorter than the propagation delay between primitives. Turning on the transport delay options in the ModelSim software prevents the simulation tool from filtering out these pulses. Use the following options to ensure that all signal pulses are seen in the simulation results.

+transport_path_delays

Use this option when the pulses in your simulation are shorter than the delay within a gate-level primitive.

+transport_int_delays

Use this option when the pulses in your simulation are shorter than the interconnect delay between gate-level primitives.

The **+transport_path_delays** and **+transport_int_delays** options are also used by default in the NativeLink feature for gate-level timing simulation.

For more information about either of these options, refer to the ModelSim-Altera Command Reference installed with the ModelSim software.

The following ModelSim software command shows the command line syntax to perform a gate-level timing simulation with the device family library:

```
vsim -t 1ps -L stratixii -sdftyp /i1=filtref_vhd.sdo work.filtref_vhd_vec_tst \
+transport_int_delays +transport_path_delays
```

Using the NativeLink Feature with ModelSim-Altera or ModelSim Software

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools and allows you to run ModelSim within the Quartus II software.

Setting Up NativeLink

To run ModelSim automatically from the Quartus II software using the NativeLink feature, you must specify the path to your simulation tool by performing the following steps:

1. On the Tools menu, click **Options**. The **Options** dialog box appears.
2. In the **Category** list, select **EDA Tool Options**.
3. Double-click the entry under **Location of executable** beside the name of your EDA Tool.
4. Type or browse to the directory containing the executables of your EDA tool.

For ModelSim-Altera software and ModelSim SE/PE, executable files are stored in the **win32aloem** and **win32** directories, respectively.

c:\\<*ModelSim-Altera installation path***>\\win32aloem**

c:\\<*ModelSim installation path***>\\win32**

5. Click **OK**.

You can also specify the path to the simulator's executables by using the `set_user_option` Tcl command:

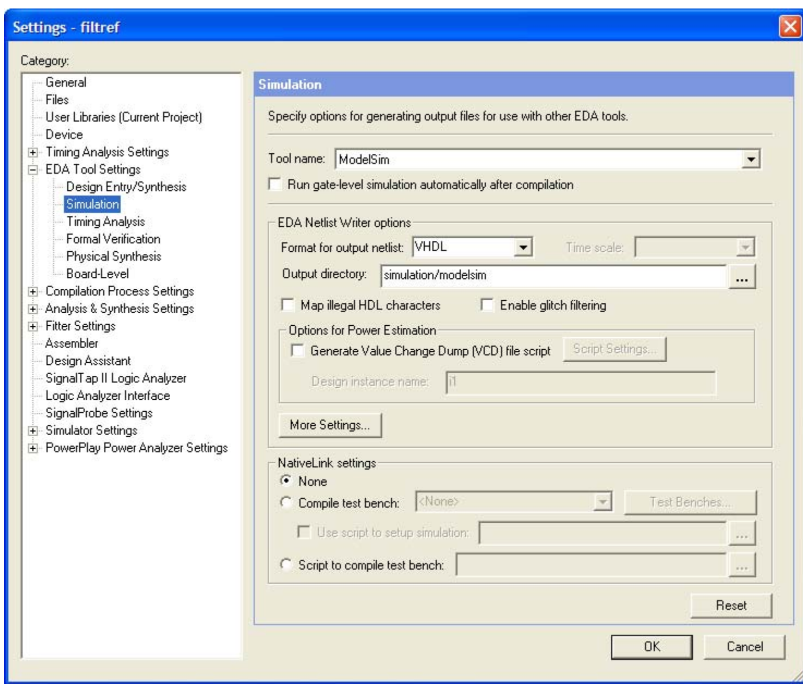
```
set_user_option -name EDA_TOOL_PATH_MODELSIM <path to executables> ↵
set_user_option -name EDA_TOOL_PATH_MODELSIM_ALTERA <path to executables> ↵
```

Perform an RTL Simulation Using NativeLink

To run an RTL functional simulation with the ModelSim software in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears (Figure 2–3).

Figure 2–3. Simulation Page in the Settings Dialog Box



3. In the **Tool name** list, select one of the following options:
   - **ModelSim**
   - **ModelSim-Altera**

4. If your design is written entirely in Verilog HDL or VHDL, the NativeLink feature automatically chooses the correct language and Altera simulation libraries. If your design is written with mixed languages, the NativeLink feature uses the default language specified in the **Format for output netlist** list. To change the default language when there is a mixed language design, under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. Table 2–12 shows the design languages for output netlists and simulation models.

Table 2–12. NativeLink Design Languages

Design File	Format for Output Netlist	Simulation Models Used
Verilog HDL	Any	Verilog HDL
VHDL	Any	VHDL
Mixed	Verilog HDL	Verilog HDL
Mixed	VHDL	VHDL

For mixed language simulation, choose the same language that was used to generate your megafunctions to ensure correct parameter passing between the megafunctions and the Altera libraries. For example, if your ALTSYNCRAM megafunction was generated in VHDL, choose **VHDL** as the format for the output netlist.

When creating mixed language designs, it is important to be aware of the following:

- EDA simulation tools do not allow seamless passing of parameters when a VHDL entity is instantiated in Verilog HDL designs.
- The ModelSim and ModelSim-Altera software does not allow the use of Verilog User Defined Primitives (UDPs) to be instantiated in VHDL designs.

5. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.

   For more information about setting up a testbench file with NativeLink, refer to "Setting Up a Testbench" on page 2–75.

6. If you have compiled libraries using the EDA Simulation Library Compiler, perform the following steps:

   a. On the **Simulation** page, click **More EDA Netlist Writer Settings**. The **More EDA Netlist Writer Settings** dialog box appears.

   b. Under **Existing option settings**, click **Location of user compiled simulation library**.

   c. In the **Setting** field, type the path that contains the user-compiled libraries that are generated from the EDA Simulation Library Compiler; for example, **c:**<*design_path*>**/simulation/modelsim**.

For more information about the EDA Simulation Library Compiler, refer to "Compile Libraries Using the EDA Simulation Library Compiler" on page 2–17.

7. Click **OK.**

8. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration** to perform an Analysis and Elaboration. This command collects all your file name information and builds your design hierarchy in preparation for simulation.
9. On the Tools menu, point to **Run EDA Simulation Tool** and click **EDA RTL Simulation** to automatically run ModelSim, compile all necessary design files, and complete a simulation.
10. If you want to run ModelSim in command-line mode when running it automatically after full compilation, perform the following steps:
    a. On the **Simulation** page, click **More NativeLink Settings**. The **More NativeLink Settings** dialog box appears.
    b. Under **Existing option settings**, click **Launch third-party EDA tool in command-line mode**.
    c. In the **Setting** field, select **On**.
    d. Click **OK**.
11. If you want to generate only the **.do** script without launching ModelSim during the NativeLink process, you can perform the following steps:
    a. On the **Simulation** page, click **More NativeLink Settings**. The **More NativeLink Settings** dialog box appears.
    b. Under **Existing option settings**, click **Generate third-party EDA tool command scripts without running the EDA tool**.
    c. In the **Setting** field, select **On**.

    If you turn this option on and run NativeLink, the **.do** file for ModelSim simulation is generated without launching ModelSim. You can then run the simulation by typing the following command:

    ```
    do <your_design_name>_run_msim_rtl_level_<verilog/vhdl>.do
    ```

Perform a Gate-Level Simulation Using NativeLink

To run a gate-level timing simulation with the ModelSim software in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears (Figure 2–3 on page 2–71).
3. In the **Tool name** list, select one of the following options:
    - **ModelSim**
    - **ModelSim-Altera**
4. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, choose **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
5. To run a gate-level simulation after each full compilation, turn on **Run gate-level simulation automatically after compilation**.

6. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.
7. If you have compiled libraries using the EDA Simulation Library Compiler, perform the following steps:
   a. On the **Simulation** page, click **More EDA Netlist Writer Settings**. The **More EDA Netlist Writer Settings** dialog box appears.
   b. Under **Existing option settings**, click **Location of user compiled simulation library**.
   c. In the **Setting** field, type the path that contains the pre-compiled libraries that are generated from the EDA Simulation Library Compilation tool; for example, **c:**<*design_path*>**/simulation/modelsim**.
8. Click **OK**.
9. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer** to generate a simulation netlist of your design.

   If you must run full compilation after you set the **EDA Tool Settings**, the **Start EDA Netlist Writer** command is not required.

10. On the Tools menu, point to **Run EDA Simulation Tool** and click **EDA Gate Level Simulation** to automatically run ModelSim, compile all necessary design files, and complete a simulation.

    If multi-corner Timing Analysis is performed, a dialog box appears, asking you to select the timing corner to run simulation. Select the timing corner and click **Run**.

    A ***.do** file is generated in the <*project_directory*>**\simulation\modelsim** directory while running NativeLink. You can perform a simulation with the ***.do** file directly from ModelSim when you rerun a simulation without using NativeLink. To perform the simulation directly without NativeLink, type the following command in the ModelSim console:
    do <*generated_do_file*>**.do**.

11. If you want to run ModelSim in command-line mode when running it automatically after full compilation, you can perform the following steps:
    a. On the **Simulation** page, click **More NativeLink Settings**. The **More NativeLink Settings** dialog box appears.
    b. Under **Existing option settings**, click **Launch third-party EDA tool in command-line mode**.
    c. In the **Setting** field, select **On**.
    d. Click **OK**.

12. If you want to generate only the **.do** script without launching ModelSim during the NativeLink process, you can perform the following steps:

    a. On the **Simulation** page, click **More NativeLink Settings**. The **More NativeLink Settings** dialog box appears.

    b. Under **Existing option settings**, click **Generate third-party EDA tool command scripts without running the EDA tool**.

    c. In the **Setting** field, select **On**.

    If you turn this option on and run the NativeLink, the **.do** file for the simulation process is generated without showing the results in the GUI. You can then run the simulation by typing the following command:

    ```
    do <your_design_name>_run_msim_rtl_level_<verilog/vhdl>.do
    ```

Setting Up a Testbench

You can use NativeLink to compile your design files and testbench files, and run an EDA simulation tool to automatically perform a simulation.

To set up NativeLink for simulation, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the "+" icon to expand **EDA Tool Settings** and select **Simulation**. The **Simulation** page appears.
3. Under **NativeLink settings**, select **None**, **Compile test bench**, or **Script to compile test bench** (Table 2–13).

Table 2–13. NativeLink Settings

Settings	Description
None	Compile simulation models and design files.
Compile test bench	NativeLink compiles simulation models, design files, testbench files, and starts simulation.
Script to compile test bench	NativeLink compiles the simulation models and design files. The script you provide is sourced after design files compile. Use this option when you want to create your own script to compile your testbench and perform simulation.

4. If you select **Compile test bench**, select your testbench setup from the **Compile test bench** list. You can use different testbench setups to specify different test scenarios. If there are no testbench setups entered, create a testbench setup by performing the following steps:
   a. Click **Test Benches**. The **Test Benches** dialog box appears.
   b. Click **New**. The **New Test Bench Settings** dialog box appears.
   c. In the **Test Bench name** box, type in the testbench setup name that identifies the different testbench setups.
   d. In the **Test level module** box, type in the top-level testbench entity name. For example, for a Quartus II-generated VHDL testbench, type in <*Vector Waveform File name*>`_vhd_vec_tst`.
   e. In the **Instance** box, type in the full instance path to the top level of your FPGA design. For example, for a Quartus II-generated VHDL testbench, type in `i1`.
   f. Under **Simulation period**, select **Run simulation until all vector stimuli are used** or specify the end time of the simulation.
   g. Under **Test bench files**, browse and add all your testbench files in the **File name** box. Use the **Up** and **Down** button to reorder your files. The script used by NativeLink compiles the files in order from top to bottom.

You can also specify the library name and HDL version to compile the testbench file. NativeLink compiles the testbench to a library name using the specified HDL version.

   h. Click **OK**.
   i. In the **Test Benches** dialog box, click **OK**.
5. Under **NativeLink settings**, turn on **Use script to set up simulation** and browse to your script. Your script is executed to set up and run simulation after loading the design using the `vsim` command.
6. If you choose **Script to compile test bench**, browse to your script and click **OK**.

Creating a Testbench

In the Quartus II software, you can create a Verilog HDL or VHDL testbench from a Vector Waveform File. The generated testbench includes the behavior of the input stimulus and applies it to your instantiated top-level FPGA design.

1. On the File menu, click **Open**. The **Open** dialog box appears.
2. Click the **Files of type** arrow and select **Waveform/Vector Files**. Select your Vector Waveform File.
3. Click **Open**.
4. On the File menu, click **Export**. The **Export** dialog box appears.
5. Click the **Save as type** arrow and select **VHDL Test Bench File (*.vht)** or **Verilog Test Bench File (*.vt)**.
6. Turn on **Add self-checking code to file** to check your simulation results against your Vector Waveform File.

7. Click **Export**. Your VHDL or Verilog HDL testbench file is generated in your project directory.

Generate Simulation Script from EDA Netlist Writer

In the Quartus II software version 9.0 and later, you can generate the simulation script (including the **.do** file, Tcl script, and option file) when you run the EDA Netlist Writer. You can use the simulation script to run your simulation in the preferred simulator in stand-alone mode.

To generate a simulation script with the EDA Netlist Writer, perform the following steps:

1. On the **Simulation** page, click **More EDA Netlist Writer Settings**. The **More EDA Netlist Writer Settings** dialog box appears.
2. Under **Existing option settings**, click **Generate third-party EDA tool command script for RTL simulation** (if you want to generate an RTL simulation script) or click **Generate third-party EDA tool command script for Gate-Level simulation** (if you want to generate a Gate-Level simulation script).
3. In the **Setting** field, select **On**.
4. Click **OK**.
5. Follow the steps in "Generate Gate-Level Timing Simulation Netlist Files" on page 2–14.

In the command-line, to generate the simulation script with the EDA Netlist Writer, type the following command:

```
quartus_eda --simulation --tools=<Your decided tool> --format=<vhdl or verilog>
    --gen_script=<rtl or gate_level> <Project Name> -c <Revision name>
```

ModelSim Error Message Verification

ModelSim error and warning messages are tagged with a `vsim` or `vcom` code. To find out the cause and resolution for a `vsim` or `vcom` error or warning, use the `verror` command.

For example, ModelSim may display the following error message:

```
# ** Error:
C:/altera_trn/DUALPORT_TRY/simulation/modelsim/DUALPORT_TRY.vho(31):
(vcom-1136) Unknown identifier "stratixiii".
```

In this case, type the following command:

```
verror 1136 ↵
```

At that point, the error message appears as follows:

```
# vcom Message # 1136:
# The specified name was referenced but was not found. This indicates
# that either the name specified does not exist or is not visible at
# this point in the code.
```

Generating a Timing VCD File for PowerPlay

To generate a timing Value Change Dump (***.vcd**) file for PowerPlay, you must first generate a ***.vcd** script file in the Quartus II software and run the ***.vcd** script file from the ModelSim or ModelSim-Altera software to generate a timing ***.vcd** file. This timing ***.vcd** file can then be used by PowerPlay for power estimation. The following instructions show you step-by-step how to generate a timing ***.vcd** file.

To generate timing VCD Scripts in the Quartus II software, perform the following steps:

1. In the Quartus II software, on the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, under **EDA Tool Settings**, select **Simulation**. The **Simulation** page appears.
3. Choose the appropriate third-party simulation tool (ModelSim or ModelSim-Altera) in the **Tool name** list. Turn on the **Generate Value Change Dump (VCD) file script** option.
4. To generate the ***.vcd** script file, perform a full compilation.

To generate a timing ***.vcd** file in the ModelSim-Altera or ModelSim software, perform the following steps:

1. In the ModelSim or ModelSim-Altera software, before simulating your design, source the `<`*revision_name*`>_dump_all_vcd_nodes.tcl` script. To source the Tcl script, run the following command before running the `vsim` command. For example:

   ```
   source <revision_name>_dump_all_vcd_nodes.tcl ↵
   ```

2. Continue to run the simulation as usual until the end of the simulation. Exit the ModelSim or ModelSim-Altera software. If you do not exit the software, the ModelSim software may end the writing process of the timing ***.vcd** files improperly, resulting in a corrupted timing ***.vcd** file.

For more information about using the timing ***.vcd** file for power estimation, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Viewing a Waveform from a .wlf File

A **.wlf** file is automatically generated when your simulation is done. The **.wlf** file is not readable. It is used for generating the waveform view through ModelSim.

To view a waveform from a **.wlf** file through ModelSim, perform the following steps:

1. Type `vsim` on a command line. The **ModelSim** dialog box appears.
2. On the View menu, click **Datasets**. The **Datasets Browser** dialog box appears.
3. Click **Open** and browse to the directory that contains your **.wlf** file.
4. If you have found your **.wlf** file, click **Open**, then click **OK**.
5. Click **Done**.
6. In the Object browser, select the signals that you want to observe.

7. On the Add menu, click **Wave** and then click **Selected Signals**.

You cannot view a waveform from a **.vcd** file in ModelSim directly. The **.vcd** file must first be converted to a **.wlf** file.

1. Use the `vcd2wlf` command to convert the file. For example, type the following on a command-line:

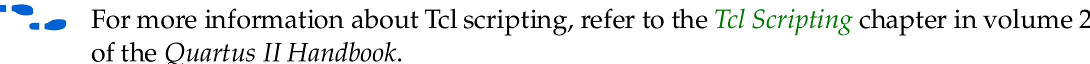
```
vcd2wlf <example>.vcd <example>.wlf ↵
```

2. After you convert the **.vcd** file to a **.wlf** file, follow the procedures for viewing a waveform from a **.wlf** file through ModelSim.

You can also convert your **.wlf** file to a **.vcd** file by using the `wlf2vcd` command.

Scripting Support

You can run procedures and create settings described in this chapter in a Tcl script. You can also run some procedures at the command line prompt.



For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For more information about command line scripting, refer to the *Command Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For detailed information about scripting command options, refer to the `Qhelp` command line and Tcl API help browser. To access this information, type the following command to start the Qhelp browser:

```
quartus_sh --qhelp ↵
```

Generating a Post-Synthesis Simulation Netlist for ModelSim

You can use the Quartus II software to generate a post-synthesis simulation netlist with Tcl commands or with a command at the command-line prompt. The following example assumes that you are selecting ModelSim (Verilog HDL output from the Quartus II software).

Tcl Commands

Use the following Tcl commands to set the output format to Verilog HDL, the simulation tool to ModelSim for Verilog HDL, and to generate a functional netlist:

```
set_global_assignment-name EDA_SIMULATION_TOOL "ModelSim (Verilog)" ↵
set_global_assignment-name EDA_GENERATE_FUNCTIONAL_NETLIST ON ↵
```

Command Prompt

Use the following command to generate a simulation output file for the ModelSim simulator. Specify VHDL or Verilog HDL for the format:

```
quartus_eda <project name> --simulation=on --format=<format> \
--tool=ModelSim --functional ↵
```

Generating a Gate-Level Timing Simulation Netlist for ModelSim

Use the Quartus II software to generate a gate-level timing simulation netlist with Tcl commands or with a command at the command prompt.

Tcl Commands

Use one of the following Tcl commands:

- `set_global_assignment -name EDA_SIMULATION_TOOL \`
  `"ModelSim-Altera (Verilog)"` ↵
- `set_global_assignment -name EDA_SIMULATION_TOOL \`
  `"ModelSim-Altera (VHDL)"` ↵
- `set_global_assignment -name EDA_SIMULATION_TOOL \`
  `"ModelSim (Verilog)"` ↵
- `set_global_assignment -name EDA_SIMULATION_TOOL \`
  `"ModelSim (VHDL)"` ↵

Command Line

Generate a simulation output file for the ModelSim simulator by specifying VHDL or Verilog HDL for the format by typing the following command at the command prompt:

```
quartus_eda <project name> --simulation=on --format=<format> \
--tool=ModelSim ↵
```

Software Licensing and Licensing Setup in ModelSim-Altera Subscription Edition

License the ModelSim-Altera Subscription Edition software subscription with a parallel port FIXEDPC license, or a network FLOATNET or FLOATPC license. Each Altera software subscription includes a license for both VHDL and Verilog HDL. The ModelSim-Altera Subscription Edition software supports both VHDL and Verilog HDL, but the software does not support mixed language simulation.

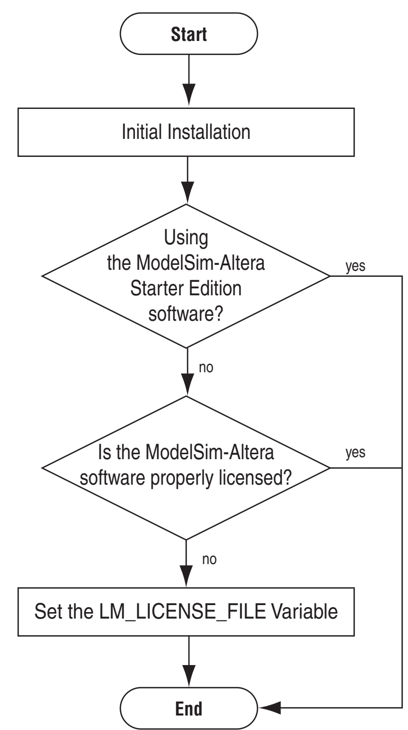
The USB software guard is not supported by versions earlier than Mentor Graphics ModelSim software 5.8d.

You can obtain a license for the ModelSim-Altera Subscription Edition software from the Altera website at www.altera.com. Get licensing information for the Mentor Graphics ModelSim software directly from Mentor Graphics. Refer to Figure 2–4 for the set-up process.

For ModelSim-Altera software versions prior to 5.5b, use the **PCLS** utility included with the software to set up the license.

For the Quartus II software version 8.1 and later, the no-cost entry level of the ModelSim-Altera software does not require a license file. However, you must request a license file to use the ModelSim-Altera Subscription Edition software.

Figure 2–4. ModelSim-Altera Subscription Edition Software Licensing Set Up Process



LM_LICENSE_FILE Variable

Altera recommends setting the `LM_LICENSE_FILE` environment variable to the location of the license file. For example, the value for the `LM_LICENSE_FILE` environment variable should point to <*path to license file*>**\license.dat.**

For more information about setting up the license for ModelSim-Altera Subscription Edition software, refer to *AN 340: Altera Software Licensing*.

Conclusion

Using the ModelSim and ModelSim-Altera simulation software within the Altera FPGA design flow enables Altera software users to easily and accurately perform RTL functional simulations, post-synthesis simulations, and gate-level simulations on their designs. Proper verification of designs at the functional, post-synthesis, and post place-and-route stages using the ModelSim and ModelSim-Altera software helps ensure design functionality and success and, ultimately, a quick time-to-market.

Referenced Documents

This chapter references the following documents:

- *AN 340: Altera Software Licensing*
- *Command Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*

- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 2–14 shows the revision history for this chapter.

Table 2–14. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Added the following sections: ■ "Compile Libraries Using the EDA Simulation Library Compiler" on page 2–17 ■ "Generate Simulation Script from EDA Netlist Writer" on page 2–77 ■ "Viewing a Waveform from a .wlf File" on page 2–78 Updated the following: ■ Table 2–1, Table 2–2, Table 2–5, Table 2–6, Table 2–7, Table 2–8, Table 2–9, Table 2–10 ■ Figure 2–4 on page 2–81 ■ All sections titled "Loading the Design"	Updated for the Quartus II software version 9.0 release.

Table 2–14. Document Revision History (Part 2 of 2)

Date and Document Version	Changes Made	Summary of Changes
November 2008 v8.1.0	Updated the following: ■ Table 2–2, Table 2–3, Table 2–4, Table 2–5, Table 2–6 ■ Removed `--zero_ic_delays` from `quartus_sta` option in "Generate Post-Synthesis Simulation Netlist Files" on page 2–11 ■ Removed steps to include the library when the simulation is run in VHDL mode from all procedures; this is no longer necessary ■ Added information about the Altera Simulation Library Compiler throughout the chapter ■ Added "Compile Libraries Using the Altera Simulation Library Compiler" on page 2–15 ■ Added "Disabling Simulation" on page 2–72 ■ Minor editorial updates ■ Updated entire chapter using 8½" × 11" chapter template	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Updated "Altera Design Flow with ModelSim-Altera or ModelSim Software" on page 2–3 ■ Updated "Simulation Libraries" on page 2–4 ■ Updated "Simulation Netlist Files" on page 2–11 ■ Updated "Perform Simulation Using ModelSim-Altera Software" on page 2–15 ■ Updated "Perform Simulation Using ModelSim Software" on page 2–33 ■ Updated "Simulating Designs that Include Transceivers" on page 2–57 ■ Updated "Using the NativeLink Feature with ModelSim-Altera or ModelSim Software" on page 2–63 ■ Updated "Generating a Timing VCD File for PowerPlay" on page 2–68	Updated for the Quartus II software version 8.0.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

3. Synopsys VCS and VCS-MX Support

QII53002-9.0.0

Introduction

This chapter is an overview of using the Synopsys VCS and VCS-MX software to simulate designs that target Altera® FPGAs. It provides a step-by-step explanation of how to perform register transfer level (RTL) functional simulations, post-synthesis simulations, and gate-level timing simulations using the VCS and VCS-MX software.

Verilog HDL design simulation is the same in both the VCS and VCS-MX software. In this chapter, VCS-MX is used in VHDL design simulation examples.

This chapter discusses the following topics:

- "Software Requirements"
- "Using the VCS or VCS-MX Software in the Quartus II Design Flow" on page 3–2
- "Common VCS and VCS-MX Software Compiler Options" on page 3–17
- "Using VirSim" on page 3–18
- "Using DVE" on page 3–18
- "Debugging Support Command-Line Interface" on page 3–19
- "Simulating Designs that Include Transceivers" on page 3–19
- "Transport Delays" on page 3–22
- "Using NativeLink with the VCS or VCS-MX Software" on page 3–22
- "Generating a Simulation Script from the EDA Netlist Writer" on page 3–27
- "Generating a Timing .vcd File for PowerPlay" on page 3–28
- "Viewing a Waveform from a .vpd or .vcd File" on page 3–28
- "Scripting Support" on page 3–29

Software Requirements

To simulate your design using the Synopsys VCS software, you must first set up the Altera libraries. These libraries are installed with the Quartus® II software.

Table 3–1 shows the compatibility between versions of the Quartus II software and the Synopsys VCS software.

Table 3–1. Supported Quartus II and VCS Software Version Compatibility (Part 1 of 2)

Synopsys	Altera
VCS and VCS-MX software version Y-2006-06-SP1	Quartus II software version 9.0
VCS software version Y-2006-06-SP1	Quartus II software version 8.1
VCS software version Y-2006.06-SP1	Quartus II software version 7.2 and 8.0
VCS software version 2006.06	Quartus II software version 7.1
VCS software version 2005.06-SP2	Quartus II software version 7.0 and 6.1
VCS software version 2005.06-SP1	Quartus II software version 6.0

Table 3–1. Supported Quartus II and VCS Software Version Compatibility (Part 2 of 2)

Synopsys	Altera
VCS software version 7.2	Quartus II software version 5.1
VCS software version 7.2	Quartus II software version 5.0
VCS software version 7.1.1	Quartus II software version 4.2

For more information about installing the software and the directories created during the Quartus II software installation, refer to the *Quartus II Installation and Licensing for Windows* or the *Quartus II Installation and Licensing for UNIX and Linux Workstation* manuals.

Using the VCS or VCS-MX Software in the Quartus II Design Flow

You can perform the following types of simulations using the VCS and VCS-MX software:

- RTL Functional Simulation
- Post-synthesis Simulation
- Gate-level timing Simulation

Figure 3–1 shows the VCS, VCS-MX, and Quartus II software design flow.

© March 2009 Altera Corporation

Figure 3–1. Altera Design Flow with the VCS, VCS-MX, and Quartus II Software *(Note 1)*, *(2)*, *(3)*, *(4)*

ALTERA IP
Design Entry
Testbench
.v / .vhd
RTL Functional Simulation *(1)*
.v/.vhd *(4)*
Functional Simulation Library Files
Synthesis
Verilog Output File and VHDL Output File
.vo / .vho
Post-Synthesis Simulation *(2)*
.v/.vhd *(4)*
Post-Synthesis Simulation Library Files
Place-and-Route
Verilog Output File and VHDL Output File
.vo / .vho
.sdo
Standard Delay Format Output File
Gate-Level Timing Simulation *(3)*
.v/.vhd *(4)*
Gate-Level Simulation Library Files

Notes to Figure 3–1:

(1) RTL functional simulation is performed before a gate-level timing simulation or post-synthesis simulation. RTL functional simulation verifies the functionality of the design before synthesis and place-and-route. If you are performing a functional simulation through NativeLink, you must complete analysis and elaboration first.

(2) A post-synthesis simulation verifies the functionality of a design after synthesis has been performed.

(3) Gate-level timing simulation is a post place-and-route simulation to verify the operation of the design after the timing delays have been calculated.

(4) All these simulation libraries are located at <*quartus installation directory*>**\eda\sim_lib**.

Compile Libraries Using the EDA Simulation Library Compiler

The EDA Simulation Library Compiler is used to compile Verilog HDL and VHDL simulation libraries for all Altera devices and supported third-party simulators. You can use this tool to compile all libraries required by RTL and gate-level simulation.

When you use this tool to perform compilation by targeting third-party simulation tools, such as ModelSim, NC-Sim, VCS_MX, Active-HDL, and Riviera-PRO, the compiled libraries are kept in either the directory you set or the default directory after compilation. When you perform the simulation using these simulators, you can reuse the compiled libraries and avoid the overhead associated with redundant library compilations.

However, if you targeted the Synopsys VCS software to perform compilation while running the EDA Simulation Library Compiler, the option file (**simlib_comp.vcs**) is generated after compilation. You can then include your design and test bench files into the option file and invoke it with the VCS command.

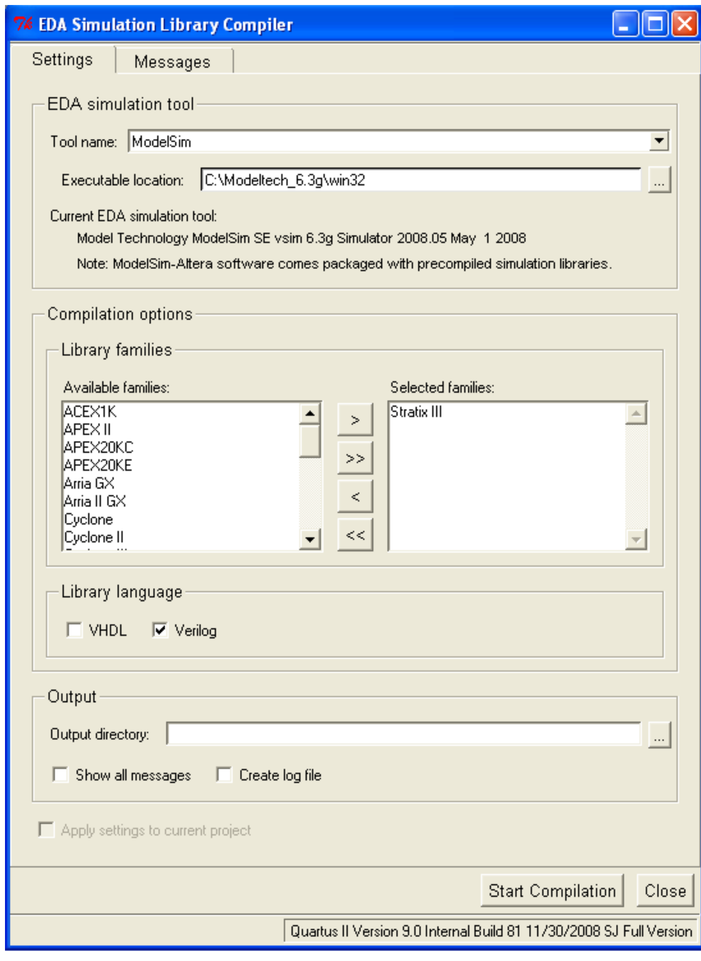
Before using this option, ensure the appropriate simulation tools are already installed with the paths specified. To specify the path, refer to "Setting Up the EDA Tool Executable Location" on page 3–22.

Using the EDA Simulation Library Compiler GUI

Beginning with the Quartus II software version 9.0, the EDA Simulation Library Compiler contains a GUI. To compile libraries using the EDA Simulation Library Compiler GUI, perform the following steps:

1. On the Tools menu, click **EDA Simulation Library Compiler**. The **EDA Simulation Library Compiler** dialog box appears (Figure 3–2).

Figure 3–2. EDA Simulation Library Compiler Dialog Box



2. In the **Tool name** entry box, select the third-party simulation tool you want to use.

   After you have selected the simulation tool, the location of the tool is displayed in the **Executable location** entry box.

3. Select one or more device families for your design compilation from the list in the **Available families** entry box. Click **>** to move them into the **Selected families** box.
4. In the Library language field, turn on **VHDL**, **Verilog**, or both.
5. In the **Output directory** entry box, specify the directory in which to store the compiled libraries or option files.
6. Turn on the **Show all messages** option, **Create log file** option, or both, if desired.
7. Turn on the **Apply settings to current project** option, if desired.
8. Click **Start Compilation**.

Verilog HDL example

To run a simulation for your Verilog HDL design with a Stratix® II device in the VCS simulator, do the following:

- Select **VCS** in the **Tool name** entry box
- Move **Stratix II** from the **Available families** list to **Selected families**
- Turn on **Verilog** in the **Library language** box
- Specify the location of the **Output directory**
- Click **Start Compilation**

After you run the EDA Simulation Library Compiler, an option file (**simlib_comp.vcs**) is generated and stored in the output location you specified. The option file is shown in Example 3–1.

Example 3–1.

```
+cli+1 -line -timescale=1ps/1ps \
-v /apps/quartus/9.0/quartus/eda/sim_lib/altera_primitives.v \
-v /apps/quartus/9.0/quartus/eda/sim_lib/220model.v \
-v /apps/quartus/9.0/quartus/eda/sim_lib/sgate.v \
-v /apps/quartus/9.0/quartus/eda/sim_lib/altera_mf.v \
-v /apps/quartus/9.0/quartus/eda/sim_lib/stratixii_atoms.v \
 +incdir+/apps/quartus/9.0/quartus/eda/sim_lib
```

If you manually run a simulation using the Synopsys VCS simulator, you must include your design file and test bench file in the option file, as shown in Example 3–2:

Example 3–2.

```
+cli+1 -line -timescale=1ps/1ps design_file.v test_bench_file.vt\
-v /apps/quartus/9.0/quartus/eda/sim_lib/altera_primitives.v \
-v /apps/quartus/9.0/quartus/eda/sim_lib/220model.v \
-v /apps/quartus/9.0/quartus/eda/sim_lib/sgate.v \
-v /apps/quartus/9.0/quartus/eda/sim_lib/altera_mf.v \
-v /apps/quartus/9.0/quartus/eda/sim_lib/stratixiii_atoms.v \
 +incdir+/apps/quartus/9.0/quartus/eda/sim_lib
```

To compile all of the libraries, design files, and test bench files, type the following command:

```
vcs -f simlib_comp.vcs ↵
```

VHDL example

To run a simulation for your VHDL design with a Stratix II device in the VCS-MX simulator, do the following:

- Select **VCSMX** in the **Tool name** entry box
- Move **Stratix II** from the **Available families** list to **Selected families**
- Turn on **VHDL** in the **Library language** box
- Specify the location of the **Output directory**
- Click **Start Compilation**

After you run the EDA Simulation Library Compiler, all of the required libraries are compiled and stored in the output location you specified.

If you use NativeLink to run the simulation, refer to "Using NativeLink with the VCS or VCS-MX Software" on page 3–22.

Using the EDA Simulation Library Compiler in the Command Line

To run the EDA Simulation Library Compiler in the command line, type the following:

```
quartus_sh --simlib_comp -family <device> -tool <simulation tool name>
-language <language> -directory <directory> ↵
```

For more information about the command line options and how to define them, you can type the command:

```
quartus_sh --help=simlib_comp ↵
```

RTL Functional Simulations

RTL functional simulations verify the functionality of the design before synthesis, placement, and routing. These simulations are independent of any Altera FPGA architecture implementation. After the HDL designs are verified to be functionally correct, the next step is to synthesize the design and use the Quartus II software to place-and-route the design in an Altera device.

To functionally simulate an Altera FPGA design in the VCS or VCS-MX software that uses Altera intellectual property (IP) megafunctions or a library of parameterized modules (LPM) functions, you must include certain libraries during the compilation. Table 3–2 summarizes the Verilog HDL library files that are required to compile LPM functions and Altera megafunctions.

Table 3–2. Altera Verilog HDL Functional/Behavioral Simulation Library Files

RTL Simulation Model	Verilog HDL Libraries
ALTGX Megafunction (Stratix IV GX)	<*path to Quartus II installation*>/eda/sim_lib/stratixiv_hssi_atoms.v *(1)*
LPM	<*path to Quartus II installation*>/eda/sim_lib/220model.v
Altera Megafunction	<*path to Quartus II installation*>/eda/sim_lib/altera_mf.v
ALTGXB Megafunction (Stratix GX)	<*path to Quartus II installation*>/eda/sim_lib/stratixgx_mf.v *(1)*
ALT2GXB Megafunction (Stratix II GX)	<*path to Quartus II installation*>/eda/sim_lib/stratixiigx_hssi_atoms.v *(1)* <*path to Quartus II installation*>/eda/sim_lib/arriagx_hssi_atoms.v *(1)*
High-Level Primitives	<*path to Quartus II installation*>/eda/sim_lib/sgate.v
Low-Level Primitives	<*path to Quartus II installation*>/eda/sim_lib/altera_primitives.v

Note to Table 3–2:

(1) The **stratixgx_mf.v**, **stratixiigx_hssi_atoms.v**, **arriagx_hssi_atoms.v**, and **stratixiv_hssi_atoms.v** library files require the LPM and SGATE libraries.

Table 3–3 summarizes the VHDL library files that are required to compile LPM functions and Altera megafunctions.

Table 3–3. Altera VHDL Functional/Behavioral Simulation Library Files

RTL Simulation Model	VHDL Libraries
ALTGX Megafunction (Stratix IV GX)	<*path to Quartus II installation*>/eda/sim_lib/stratixiv_hssi_components.vhd <*path to Quartus II installation*>/eda/sim_lib/stratixiv_hssi_atoms.vhd *(1)*
LPM	<*path to Quartus II installation*>/eda/sim_lib/220pack.vhd <*path to Quartus II installation*>/eda/sim_lib/220model.vhd <*path to Quartus II installation*>/eda/sim_lib/220model_87.vhd
Altera Megafunction	<*path to Quartus II installation*>/eda/sim_lib/altera_mf_components.vhd <*path to Quartus II installation*>/eda/sim_lib/altera_mf.vhd <*path to Quartus II installation*>/eda/sim_lib/altera_mf_87.vhd
ALTGXB Megafunction (Stratix GX)	<*path to Quartus II installation*>/eda/sim_lib/stratixgx_mf_components.vhd <*path to Quartus II installation*>/eda/sim_lib/stratixgx_mf.vhd *(1)*
ALT2GXB Megafunction (Stratix II GX)	<*path to Quartus II installation*>/eda/sim_lib/stratixiigx_hssi_components.vhd <*path to Quartus II installation*>/eda/sim_lib/stratixiigx_hssi_atoms.vhd *(1)* <*path to Quartus II installation*>/eda/sim_lib/arriagx_hssi_components.vhd <*path to Quartus II installation*>/eda/sim_lib/arriagx_hssi_atoms.vhd *(1)*
High-Level Primitives	<*path to Quartus II installation*>/eda/sim_lib/sgate_pack.vhd <*path to Quartus II installation*>/eda/sim_lib/sgate.vhd
Low-Level Primitives	<*path to Quartus II installation*>/eda/sim_lib/altera_primitives_components.vhd <*path to Quartus II installation*>/eda/sim_lib/altera_primitives.vhd

Note to Table 3–3:

(1) The **stratixgx_mf.vhd**, **stratixiigx_hssi_atoms.vhd**, **arriagx_hssi_atoms.vhd**, and **stratixiv_hssi_atoms.vhd** library files require the LPM and SGATE libraries.

RTL Functional Simulation (Verilog HDL Designs)

The following VCS command performs an RTL functional simulation for Verilog HDL designs with one of the libraries in Table 3–2:

```
vcs -R <testbench>.v <design name>.v -v <library file1>.v -v <library file 2>.v ↵
```

If you have already generated the option file (**simlib_comp.vcs**) from the EDA Simulation Library Compiler, type the following command:

```
vcs -f simlib_comp.vcs ↵
```

Be sure to include the design files and testbench files in **simlib_comp.vcs**.

Alternatively, you can use the following commands to perform RTL functional simulation for Verilog HDL designs:

```
# Create Libraries Directories
mkdir <Directory_to_store_compiled_library1>
mkdir <Directory_to_store_compiled_library2>
# Create Work Directory
mkdir <Directory_to_store_compiled_design_and_testbench_files>
# Compilation
# (Before this step, make sure the mapped file ".synopsys_vss.setup" is created.)
# Libraries Compilation
vlogan -work <library1_name> <library1>.v
vlogan -work <library2_name> <library2>.v
# Design and Test bench files Compilation
vlogan -work <work_library_name> <design>.v <test_bench>.v
# Eleboration
vcs -debug_all <work_library_name>.<testbench_top_level_module>
# Run Simulation
simv -gui
```

.synopsys_vss.setup contains the following mapping commands to map the libraries:

```
<library1_name> : <Directory_to_store_compiled_library1>
<library2_name> : <Directory_to_store_compiled_library2>
<work_library_name> : <Directory_to_store_compiled_design_and_testbench_files>
```

RTL Functional Simulation (VHDL Designs)

For VHDL designs, you need to use VCS-MX software to perform all types of simulations. The following commands are for performing an RTL functional simulation for VHDL designs with one of the libraries in Table 3–3 on page 3–8:

```
# Create Libraries Directories
mkdir <Directory_to_store_compiled_library1>
mkdir <Directory_to_store_compiled_library2>
# Create Work Directory
mkdir <Directory_to_store_compiled_design_and_testbench_files>
# Compilation
# (Before this step, make sure the mapped file ".synopsys_vss.setup" is created.)
# Libraries Compilation
vhdlan -work  <library1_name> <library1>.vhd
vhdlan -work  <library2_name> <library2>.vhd
# Design and Test bench files Compilation
vhdlan -work <work_library_name> <design>.vhd <test_bench>.vhd
# Eleboration
scs -debug_all <work_library_name>.<testbench_top_level_module>
# Run Simulation
scsim -gui
```

.synopsys_vss.setup contains the following mapping commands to map the libraries:

```
<library1_name> : <Directory_to_store_compiled_library1>
<library2_name> : <Directory_to_store_compiled_library2>
<work_library_name> : <Directory_to_store_compiled_design_and_testbench_files>
```

If you have generated the Altera libraries with the EDA Simulation Library Compiler, ignore the steps **# Create Libraries Directories** and **# Libraries Compilation**.

Post-Synthesis Simulation

A post-synthesis simulation verifies the functionality of a design after synthesis has been performed. You can create a post-synthesis netlist in the Quartus II software and use this netlist to perform a post-synthesis simulation in the VCS or VCS-MX software. When the post-synthesis version of the design has been verified, the next step is to place-and-route the design in the target architecture using the Quartus II software.

Generating a Post-Synthesis Simulation Netlist

The following steps describe the process of generating a post-synthesis simulation netlist in the Quartus II software:

1. Perform Analysis and Synthesis. On the Processing menu, point to **Start** and click **Start Analysis & Synthesis**.

2. Turn on the **Generate Netlist for Functional Simulation Only** option by performing the following steps:
   a. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
   b. In the **Category** list, under **EDA Tool Settings,** select **Simulation**. The **Simulation** dialog box appears.
   c. In the **Tool name** list, select **VCS or VCS MX**.
   d. You can modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
   e. Click **More EDA Netlist Writer Settings**. The **More EDA Netlist Writer Settings** dialog box appears.
   f. In the **Existing option settings** list, click **Generate netlist for functional simulation only**.
   g. Under **Option**, from the **Setting** list, select **On**.
   h. Click **OK**.
   i. In the **Settings** dialog box, click **OK**.
3. Run the EDA Netlist Writer. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer**.

During the EDA Netlist Writer stage, the Quartus II software produces a Verilog Output File (**.vo**) or VHDL Output File (**.vho**) that can be used for post-synthesis simulation in the VCS software. This netlist file is mapped to architecture-specific primitives. No timing information is included at this stage.

The resulting netlist is located in the output directory you specified in the **Settings** dialog box, which defaults to the <*project directory*>**/simulation/vcs** directory for VCS or to the <*project directory*>**/simulation/scsim** directory for VCS-MX. This netlist, along with the device family library listed in Table 3–4 and Table 3–5, can be used to perform a post-synthesis simulation in the VCS or VCS-MX software.

Table 3–4. Altera Gate-Level Timing Simulation Library Files (Verilog HDL) (Part 1 of 2)

Device Simulation Model	Location in Quartus II Directory Structure
Arria® II (without transceiver block)	<*Quartus II installation directory*>\eda\sim_lib\arriaii_atoms.v
Arria II (with transceiver block)	<*Quartus II installation directory*>\eda\sim_lib\arriaii_hssi_atoms.v
Arria II (with PCI Express)	<*Quartus II installation directory*>\eda\sim_lib\arriaii_pcie_atoms.v
Arria GX (without transceiver block)	<*Quartus II installation directory*>\eda\sim_lib\arriagx_atoms.v
Arria GX (with transceiver block)	<*Quartus II installation directory*>\eda\sim_lib\arriagx_hssi_atoms.v
Stratix IV (without transceiver block)	<*Quartus II installation directory*>\eda\sim_lib\stratixiv_atoms.v
Stratix IV (with transceiver block)	<*Quartus II installation directory*>\eda\sim_lib\stratixiv_hssi_atoms.v

Table 3–4. Altera Gate-Level Timing Simulation Library Files (Verilog HDL) (Part 2 of 2)

Device Simulation Model	Location in Quartus II Directory Structure
Stratix IV (with PCI Express)	*<Quartus II installation directory>*\eda\sim_lib\stratixiv_pcie_hip_atoms.v
Stratix III	*<Quartus II installation directory>*\eda\sim_lib\stratixiii_atoms.v
Stratix II	*<Quartus II installation directory>*\eda\sim_lib\stratixii_atoms.v
Stratix II GX (without transceiver block)	*<Quartus II installation directory>*\eda\sim_lib\stratixiigx_atoms.v
Stratix II GX (with transceiver block)	*<Quartus II installation directory>*\eda\sim_lib\stratixiigx_hssi_atoms.v
Stratix	*<Quartus II installation directory>*\eda\sim_lib\stratix_atoms.v
Stratix GX (without transceiver block)	*<Quartus II installation directory>*\eda\sim_lib\stratixgx_atoms.v
Stratix GX (with transceiver block)	*<Quartus II installation directory>*\eda\sim_lib\stratixgx_hssi_atoms.v
Cyclone® III	*<Quartus II installation directory>*\eda\sim_lib\cycloneiii_atoms.v
Cyclone II	*<Quartus II installation directory>*\eda\sim_lib\cycloneii_atoms.v
Cyclone	*<Quartus II installation directory>*\eda\sim_lib\cyclone_atoms.v
MAX® II	*<Quartus II installation directory>*\eda\sim_lib\maxii_atoms.v
MAX 7000 MAX 3000	*<Quartus II installation directory>*\eda\sim_lib\max_atoms.v
APEX™ II	*<Quartus II installation directory>*\eda\sim_lib\apexii_atoms.v
APEX 20K	*<Quartus II installation directory>*\eda\sim_lib\apex20k_atoms.v
APEX 20KC APEX 20KE Excalibur™	*<Quartus II installation directory>*\eda\sim_lib\apex20ke_atoms.v
HardCopy® II	*<Quartus II installation directory>*\eda\sim_lib\hardcopyii_atoms.v
FLEX® 6000	*<Quartus II installation directory>*\eda\sim_lib\flex6000_atoms.v
FLEX 10KE ACEX® 1K	*<Quartus II installation directory>*\eda\sim_lib\flex10ke_atoms.v

Table 3–5. Altera Gate-Level Timing Simulation Library Files (VHDL) (Part 1 of 2)

Device Simulation Model	Location in Quartus II Directory Structure
Arria GX (without transceiver block)	*<Quartus II installation directory>*\eda\sim_lib\arriagx_components.vhd *<Quartus II installation directory>*\eda\sim_lib\arriagx_atoms.vhd
Arria GX (with transceiver block)	*<Quartus II installation directory>*\eda\sim_lib\arriagx_hssi_components.vhd *<Quartus II installation directory>*\eda\sim_lib\arriagx_hssi_atoms.vhd
Arria II (without transceiver block)	*<Quartus II installation directory>*\eda\sim_lib\arriaii_components.vhd *<Quartus II installation directory>*\eda\sim_lib\arriaii_atoms.vhd
Arria II (with transceiver block)	*<Quartus II installation directory>*\eda\sim_lib\arriaii_hssi_components.vhd *<Quartus II installation directory>*\eda\sim_lib\arriaii_hssi_atoms.vhd
Arria II (with PCI Express)	*<Quartus II installation directory>*\eda\sim_lib\arriaii_pcie_components.vhd *<Quartus II installation directory>*\eda\sim_lib\arriaii_pcie_atoms.vhd
Stratix IV (without transceiver block)	*<Quartus II installation directory>*\eda\sim_lib\stratixiv_components.vhd *<Quartus II installation directory>*\eda\sim_lib\stratixiv_atoms.vhd

Table 3–5. Altera Gate-Level Timing Simulation Library Files (VHDL) (Part 2 of 2)

Device Simulation Model	Location in Quartus II Directory Structure
Stratix IV (with transceiver block)	<*Quartus II installation directory*>\eda\sim_lib\stratixiv_hssi_components.vhd <*Quartus II installation directory*>\eda\sim_lib\stratixiv_hssi_atoms.vhd
Stratix IV (with PCI Express)	<*Quartus II installation directory*>\eda\sim_lib\stratixiv_pcie_hip_components.vhd <*Quartus II installation directory*>\eda\sim_lib\stratixiv_pcie_hip_atoms.vhd
Stratix III	<*Quartus II installation directory*>\eda\sim_lib\stratixiii_components.vhd <*Quartus II installation directory*>\eda\sim_lib\stratixiii_atoms.vhd
Stratix II	<*Quartus II installation directory*>\eda\sim_lib\stratixii_components.vhd <*Quartus II installation directory*>\eda\sim_lib\stratixii_atoms.vhd
Stratix II GX (without transceiver block)	<*Quartus II installation directory*>\eda\sim_lib\stratixiigx_components.vhd <*Quartus II installation directory*>\eda\sim_lib\stratixiigx_atoms.vhd
Stratix II GX (with transceiver block)	<*Quartus II installation directory*>\eda\sim_lib\stratixiigx_hssi_components.vhd <*Quartus II installation directory*>\eda\sim_lib\stratixiigx_hssi_atoms.vhd
Stratix	<*Quartus II installation directory*>\eda\sim_lib\stratix_components.vhd <*Quartus II installation directory*>\eda\sim_lib\stratix_atoms.vhd
Stratix GX (without transceiver block)	<*Quartus II installation directory*>\eda\sim_lib\stratixgx_components.vhd <*Quartus II installation directory*>\eda\sim_lib\stratixgx_atoms.vhd
Stratix GX (with transceiver block)	<*Quartus II installation directory*>\eda\sim_lib\stratixgx_hssi_components.vhd <*Quartus II installation directory*>\eda\sim_lib\stratixgx_hssi_atoms.vhd
Cyclone III	<*Quartus II installation directory*>\eda\sim_lib\cycloneiii_components.vhd <*Quartus II installation directory*>\eda\sim_lib\cycloneiii_atoms.vhd
Cyclone II	<*Quartus II installation directory*>\eda\sim_lib\cycloneii_components.vhd <*Quartus II installation directory*>\eda\sim_lib\cycloneii_atoms.vhd
Cyclone	<*Quartus II installation directory*>\eda\sim_lib\cyclone.vhd <*Quartus II installation directory*>\eda\sim_lib\cyclone_atoms.vhd
MAX II	<*Quartus II installation directory*>\eda\sim_lib\maxii_components.vhd <*Quartus II installation directory*>\eda\sim_lib\maxii_atoms.vhd
MAX 7000 MAX 3000	<*Quartus II installation directory*>\eda\sim_lib\max_components.vhd <*Quartus II installation directory*>\eda\sim_lib\max_atoms.vhd
APEX II	<*Quartus II installation directory*>\eda\sim_lib\apexii_components.vhd <*Quartus II installation directory*>\eda\sim_lib\apexii_atoms.vhd
APEX 20K	<*Quartus II installation directory*>\eda\sim_lib\apex20k_components.vhd <*Quartus II installation directory*>\eda\sim_lib\apex20k_atoms.vhd
APEX 20KC APEX 20KE Excalibur	<*Quartus II installation directory*>\eda\sim_lib\apex20ke_components.vhd <*Quartus II installation directory*>\eda\sim_lib\apex20ke_atoms.vhd
HardCopy II	<*Quartus II installation directory*>\eda\sim_lib\hardcopyii_components.vhd <*Quartus II installation directory*>\eda\sim_lib\hardcopyii_atoms.vhd
FLEX 6000	<*Quartus II installation directory*>\eda\sim_lib\flex6000_components.vhd <*Quartus II installation directory*>\eda\sim_lib\flex6000_atoms.vhd
FLEX 10KE ACEX 1K	<*Quartus II installation directory*>\eda\sim_lib\flex10ke_components.vhd <*Quartus II installation directory*>\eda\sim_lib\flex10ke_atoms.vhd

Post-Synthesis Simulations (Verilog HDL)

The following VCS command shows the command-line syntax used to perform a post-synthesis simulation with the appropriate device family library listed in Table 3–4:

```
vcs -R <testbench> <post synthesis netlist> -v <Altera device family library> ↵
```

If you have already generated the option file (**simlib_comp.vcs**) from the EDA Simulation Library Compiler, type the following command:

```
vcs -f simlib_comp.vcs ↵
```

Be sure to include the post synthesis netlist file and testbench files in **simlib_comp.vcs**.

Alternatively, you can use the following commands to perform Post-Synthesis simulation for Verilog HDL designs:

```
# Create Libraries Directories
mkdir <Directory_to_store_compiled_library1>
mkdir <Directory_to_store_compiled_library2>
# Create Work Directory
mkdir <Directory_to_store_compiled_post_synthesis_netlist_and_testbench_files>
# Compilation
# (Before this step, make sure that mapped file ".synopsys_vss.setup" is created.)
# Libraries Compilation
vlogan -work  <library1_name> <library1>.v
vlogan -work  <library2_name> <library2>.v
# Design and Test bench files Compilation
vlogan -work <work_library_name> <post_synthesis_netlist>.vo <test_bench>.v
# Eleboration
vcs -debug_all <work_library_name>.<testbench_top_level_module>
# Run Simulation
simv -gui
```

The **.synopsys_vss.setup** file contains the following commands to map the libraries:

```
<library1_name> : <Directory_to_store_compiled_library1>
<library2_name> : <Directory_to_store_compiled_library2>
<work_library_name> :
   <Directory_to_store_compiled_post_synthesis_netlist_and_testbench_files>
```

Post-Synthesis Simulations (VHDL)

The following VCS-MX software commands show the command-line syntax used to perform a post-synthesis simulation with the appropriate device family library listed in Table 3–5:

```
# Create Libraries Directories
mkdir <Directory_to_store_compiled_library1>
mkdir <Directory_to_store_compiled_library2>
# Create Work Directory
mkdir <Directory_to_store_compiled_post_synthesis_netlist_and_testbench_files>
# Compilation
#(Before this step, make sure that mapped file ".synopsys_vss.setup" is created.)
# Libraries Compilation
vhdlan -work  <library1_name> <library1>.vhd
vhdlan -work  <library2_name> <library2>.vhd
# Design and Test bench files Compilation
vhdlan -work <work_library_name> <post_synthesis_netlist>.vho <test_bench>.vhd
# Eleboration
scs -debug_all <work_library_name>.<testbench_top_level_module>
# Run Simulation
scsim -gui
```

The **.synopsys_vss.setup** file contains the following commands to map the libraries:

```
<library1_name> : <Directory_to_store_compiled_library1>
<library2_name> : <Directory_to_store_compiled_library2>
<work_library_name> :
   <Directory_to_store_compiled_post_synthesis_netlist_and_testbench_files>
```

If you have generated the Altera libraries with EDA Simulation Library Compiler, ignore the steps **# Create Libraries Directories** and **# Libraries Compilation**.

Gate-Level Timing Simulation

A gate-level timing simulation verifies the functionality of the design after place-and-route. You can create a post-fit netlist in the Quartus II software and use this netlist to perform a gate-level timing simulation in the VCS or VCS-MX software.

Generating a Gate-Level Timing Simulation Netlist

To perform gate-level timing simulation, the VCS or VCS-MX software requires information about how the design was placed into device-specific architectural blocks. The Quartus II software provides this information in the form of a **.vo** file for Verilog HDL designs or a **.vho** file for VHDL designs. The accompanying timing information is stored in the Standard Delay Output (**.sdo**) file, which annotates the delay for the elements found in the **.vo** or **.vho** file.

To generate a gate-level timing simulation netlist in the Quartus II software, perform the following steps:

1. Perform a full compilation. On the Processing menu, click **Start Compilation**.
2. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
3. In the **Category** list, under **EDA Tool Settings**, select **Simulation**. The **Simulation** dialog box appears.
4. In the **Tool name** list, select **VCS** or **VCS MX**.
5. You can modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
6. Click **OK**.
7. In the **Settings** dialog box, click **OK**.
8. Run the EDA Netlist Writer. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer**.

During the EDA Netlist Writer stage, the Quartus II software produces a **.vo** file that can be used for post-synthesis simulations in the VCS software or a **.vho** file that can be used for post-synthesis simulations in the VCS-MX software. This netlist file is mapped to architecture-specific primitives. No timing information is included at this stage. The resulting netlist is located in the output directory you specified in the **Settings** dialog box, which defaults to the <*project directory*>**/simulation/vcs** directory for Verilog HDL designs or the <*project directory*>**/simulation/scsim** directory for VHDL designs.

Generating a Timing Netlist with Different Timing Models

In Stratix III and later devices (for example, Cyclone III and Stratix IV devices) you can specify different temperature and voltage parameters to generate the timing netlist. If you enable the Quartus II Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer when generating the simulation netlist files (***.vo** or ***.vho** and ***.sdo**), different timing models for different operating conditions are used by default. The multi-corner timing analysis is run by default during the full compilation.

Table 3–6 shows examples of default available operating conditions (model, voltage, and temperature) for Stratix III and Cyclone III devices.

Table 3–6. Default Available Operating Condition for Stratix III and Cyclone III Devices

Device Family	Model	Voltage	Temperature (C)
Stratix III	Slow	1100 mV	85°
	Slow	1100 mV	0°
	Fast	1100 mV	0°
Cyclone III	Slow	1200 mV	85°
	Slow	1200 mV	0°
	Fast	1200 mV	0°

If the multi-corner timing analysis is not run during full compilation, you should perform the following steps to manually generate the simulation netlist files (***.vo** or ***.vho** and ***.sdo**) for the three different operating conditions listed in Table 3–6:

1. Generate all the available corner models at all operating conditions. Type the following command at a command prompt:

```
quartus_sta <project name> --multicorner ↵
```

2. Generate the simulation output files for all three corners specified in Table 3–6. Perform steps 2 through 8 in “Generating a Gate-Level Timing Simulation Netlist” on page 3–14. The output files are generated in the simulation output directory.

The following example shows all of the generated simulation netlist files for the three operating conditions of the preceding steps, when **Verilog** is selected as the output netlist format:

First slow corner (slow, 1100 mV, 85° C):

.vo file— *<revision name>***.vo**

.sdo file— *<revision name>***_v.sdo**

The *<revision_name>***.vo** and *<revision name>***_v.sdo** are generated for backward compatibility in case there are existing scripts that still use them.

.vo file— *<revision name>*_*<speedgrade>***_1100mv_85c_slow.vo**

.sdo file— *<revision name>*_*<speedgrade>***_1100mv_85c_v_slow.sdo**

Second slow corner (slow, 1100 mV, 0° C):

.vo file— *<revision name>*_*<speedgrade>***_1100mv_0c_slow.vo**

.sdo file— *<revision name>*_*<speedgrade>***_1100mv_0c_v_slow.sdo**

Fast corner (fast, 1100 mV, 0° C):

.vo file— *<revision name>***_min_1100mv_0c_fast.vo**

.sdo file— *<revision name>***_min_1100mv_0c_v_fast.sdo**

For all other older device families (for example, Stratix II and Stratix devices), a slow-corner (worst case) timing model is used by default. There are only two timing models available, the slow-corner and fast-corner timing models. To generate the **.sdo** file using a different timing model, you must run the Quartus II Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer with a different timing model before you start the EDA Netlist Writer.

To run the Quartus II Classic Timing Analyzer with the best-case model, on the Processing menu, point to **Start** and click **Start Classic Timing Analyzer (Fast Timing Model)**. After the timing analysis is complete, the Compilation Report appears. You can also type the following command at a command prompt:

```
quartus_tan <project_name> --fast_model=on ↵
```

To run the Quartus II TimeQuest Timing Analyzer with a best-case model, use the **-fast_model** option after you create the timing netlist. The following command enables the fast timing models:

```
create_timing_netlist --fast_model ↵
```

After you run the Quartus II Classic Timing Analyzer or Quartus II TimeQuest Timing Analyzer, you can perform steps 2 through 8 in "Generating a Gate-Level Timing Simulation Netlist" on page 3–14 to generate the ***.sdo** file. For fast-corner timing models, the `-fast` post fix is added to the ***.vo** or ***.vho** and ***.sdo** files (for example, **my_project_fast.vo** or **my_project_fast.vho** and **my_project_fast.sdo**).

For more information about generating the timing netlist with different timing models, refer to the *Quartus II Classic Timing Analyzer* chapter or the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Gate-Level Timing Simulations (Verilog HDL)

For gate-level timing simulation, follow the steps in "Post-Synthesis Simulations (Verilog HDL)" on page 3–13.

You do not need to specify the ***.sdo** file because it is already specified in the ***.vo** file. However, the ***.sdo** file must be in same directory in which the `simv` command is running.

Gate-Level Timing Simulations (VHDL)

For gate-level timing simulation, follow the steps in "Post-Synthesis Simulations (VHDL)" on page 3–13.

For VHDL, the ***.sdo** file must be specified in the `scsim` command as follows:

```
scsim -gui -sdf typ:<design_instance>:<design>.sdo ↵
```

Disable Timing Violation on Registers

In certain situations, the timing violations can be ignored and you want to disable the timing violation on registers (for example, timing violations that occur in internal synchronization registers used for asynchronous clock-domain crossing).

By default, the **x_on_violation_option logic** option is **On**, which means simulation shows "**X**" whenever a timing violation occurs. To disable showing the timing violation on certain registers, you can set the **x_on_violation_option logic** option to **Off** for those registers. The following command is an example of the Quartus II Settings file (**.qsf**):

```
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to <register_name> ↵
```

Perform Timing Simulation Using the Post-Synthesis Netlist

You can perform a timing simulation using the post-synthesis netlist instead of using a gate-level netlist, and you can generate an **.sdo** file without running the fitter. In this case, the **.sdo** file includes all timing values for the device cells only. Interconnect delays are not included because fitting (placement and routing) has not been performed.

To generate the post-synthesis netlist and the **.sdo** file, type the following commands at a command prompt:

```
quartus_map <project name> -c <revision name> ↵
quartus_tan <project name> -c <revision name> --post_map \
--zero_ic_delays ↵
quartus_eda <project name> -c <revision name> --simulation \
--tool= <third-party EDA tool> --format=<HDL language> ↵
```

For more information about the `-format` and `-tool` options, type the following command:

```
quartus_eda -help=<options> command ↵
```

Common VCS and VCS-MX Software Compiler Options

The VCS software has options that help you simulate your design. Table 3–7 lists some of the available options.

Table 3–7. VCS Software Compiler Options (Part 1 of 2)

Library	Description
`-R`	Runs the executable file immediately.
`-RI`	After the compile has completed, instructs the VCS software to automatically launch VirSim.
`-v <library filename>`	Specifies a Verilog HDL library file (for example, **220model.v** or **altera_mf.v**). The VCS software looks in this file for module definitions that are found in the source code. Only the relevant library files are compiled based on the modules found.
`-y <library directory>`	Specifies a Verilog HDL library directory. The VCS software looks for library files in this folder that contain module definitions that are instantiated in the source code.
`+compsdf`	Indicates that the VCS software compiler includes the back-annotated Standard Delay File (**.sdf**) file in the compilation.
`+cli`	The VCS software enters Command-Line Interface (CLI) mode upon successful compilation completion.

Table 3–7. VCS Software Compiler Options (Part 2 of 2)

Library	Description
`+race`	Specifies that the VCS software generate a report that indicates all of the race conditions in the design. The default report name is race.out.
`-P`	Compiles user-defined Programming Language Interface (PLI) table files.
`-q`	Indicates the VCS software runs in quiet mode. All messages are suppressed.

For more information about VCS software options, refer to the *VCS User Guide*.

Using VirSim

VirSim is the graphical debugging system for the VCS software. This tool is included with the VCS software and can be run by using the `-RI` compile-time compiler option when compiling a design. The following VCS software command shows the command-line syntax for compiling and loading a timing simulation in VirSim:

```
vcs -RI <testbench>.v <design name>.vo -v <path to Quartus II installation> \
   \eda\sim_lib\<device family>_atoms.v +compsdf ↵
```

For more information about using VirSim, refer to the *VirSim User Guide* included in the VCS software installation.

Using DVE

DVE is the graphical debugging system for the VCS and VCS-MX software. This tool is included with the VCS-MX software. It can be run by using the `-gui` (simulating option) when running a simulation.

The following VCS or VCS-MX software commands show the command-line syntax for simulating in DVE:

`simv -gui` (for Verilog HDL simulation)

`scsim -gui` (for VHDL simulation)

However, to open the GUI with these commands, you must enable the use of UCLI and DVE when performing elaboration. To enable UCLI and DVE, enter the following commands:

`vcs -debug_all` (for Verilog HDL simulations)

`scs -debug_all` (for VHDL simulations)

For more information about using DVE, refer to the *DVE User Guide* included in the VCS-MX software installation.

Debugging Support Command-Line Interface

The Synopsys VCS software has an interactive non-graphical debugging capability that is very similar to other UNIX debuggers such as the GNU debugger (GDB). The VCS software CLI can be used to halt simulations at user-defined break points, force registers with values, and display register values.

Enable the non-graphical capability by using the `+cli` run-time option. Use the VCS software CLI to debug your Altera FPGA design by typing the following command:

```
vcs -R <testbench>.v <design name>.vo -v <path to Quartus II \
installation> \eda\sim_lib\<device family>_atoms.v +compsdf +cli ↵
```

The `+cli` command takes an optional number argument that specifies the level of debugging capability. As the optional debugging capability is increased, the overhead incurred by the simulation is increased, resulting in an increase in simulation time.

For more information about the `+cli` options, refer to the *VCS User Guide* included in the VCS software installation.

For the design examples to run gate-level timing simulation in VHDL or Verilog HDL language, refer to www.altera.com/support/examples/vcs/exm-vcs.html.

Simulating Designs that Include Transceivers

If your design includes a Stratix IV, Stratix II GX, Stratix GX, Arria II GX, or Arria GX transceiver, you must compile additional library files to perform functional RTL or gate-level timing simulations.

Stratix GX RTL Functional Simulation

To perform an RTL functional simulation of your design that instantiates the ALTGXB megafunction, enabling the gigabit transceiver block gigabit transceiver block on Stratix GX devices, you should compile the **stratixgx_mf** model file into the **altgxb** library.

The **stratixiigx_mf** model file references the **lpm** and **sgate** libraries; you must create these libraries to perform a simulation.

Compiling Library Files for Functional Stratix GX Simulation in Verilog HDL

To compile the libraries necessary for functional simulation of a Verilog HDL design targeting a Stratix GX device, at the VCS command prompt, type the following command:

```
vcs -R <testbench>.v <design files>.v -v stratixgx_mf.v -v sgate.v \
-v 220model.v -v altera_mf.v ↵
```

Stratix GX Gate-Level Timing Simulation

Perform a gate-level timing simulation of your design that includes a Stratix GX transceiver by compiling the **stratixgx_atoms** and **stratixgx_hssi_atoms** model files into the **stratixgx** and **stratixgx_gxb** libraries, respectively.

The **stratixgx_hssi_atoms** model file references the **lpm** and **sgate** libraries; you must create these libraries to perform a simulation.

Compiling Library Files for Timing Stratix GX Simulation in Verilog HDL

To compile the libraries necessary for timing simulation of a Verilog HDL design targeting a Stratix GX device, at the VCS command prompt, type the following command:

```
vcs -R <testbench>.v <gate-level netlist>.vo -v stratixgx_atoms.v -v \
stratixgx_hssi_atoms.v -v sgate.v -v 220model.v -v altera_mf.v +transport_int_delays \
+pulse_int_e/0 +pulse_int_r/0 +transport_path_delays +pulse_e/0 +pulse_r/0 ↵
```

Stratix II GX RTL Functional Simulation

Stratix II GX RTL functional simulation is similar to Arria GX RTL functional simulation. The following example shows only the RTL functional simulation for designs that include transceivers in Stratix II GX. To simulate the transceiver in Arria GX, you only have to replace the **stratixiigx_hssi** model file with the **arriagx_hssi** model file.

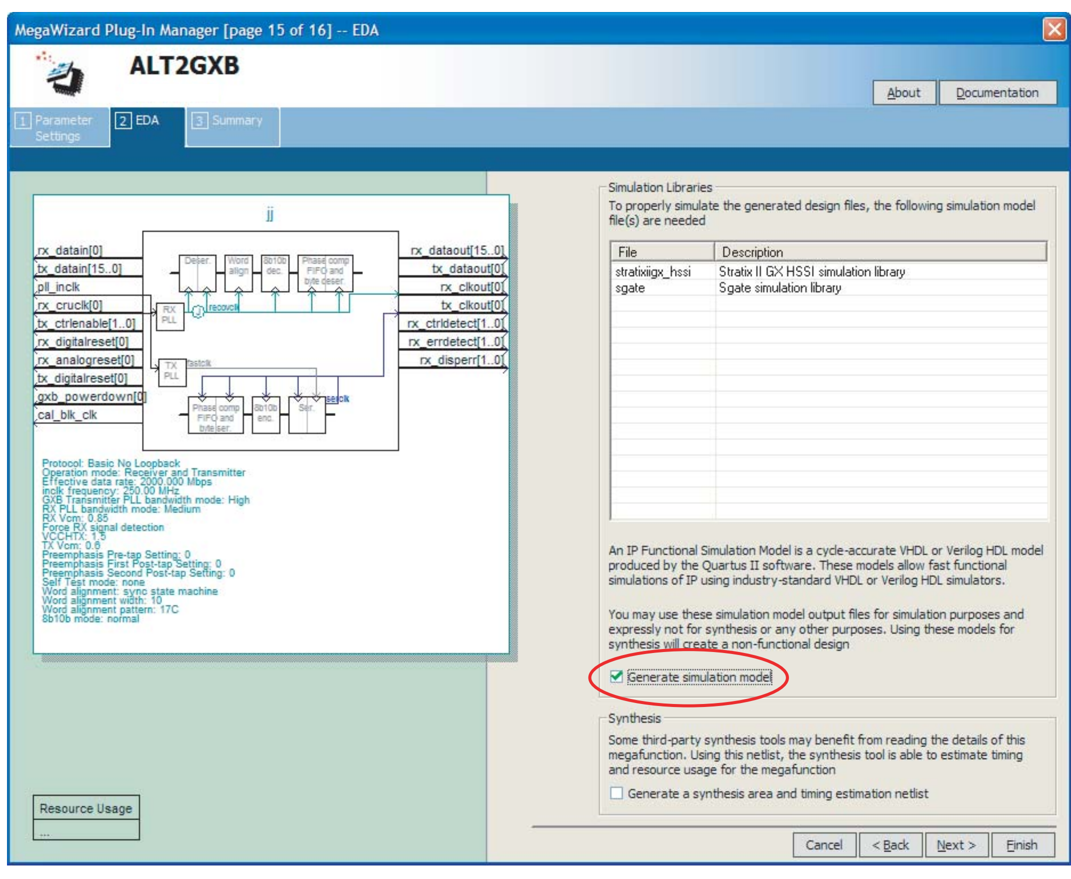
To perform an RTL functional simulation of your design that instantiates the ALT2GXB megafunction, enabling the gigabit transceiver block gigabit transceiver block on Stratix II GX devices, you should compile the **stratixiigx_hssi** model file into the **stratixiigx_hssi** library.

The **stratixiigx_hssi_atoms** model file references the **lpm** and **sgate** libraries; you must create these libraries to perform a simulation.

Generate a functional simulation netlist by turning on **Generate Simulation Model in the Simulation Library** in the ALT2GXB MegaWizard™ Plug-In Manager (Figure 3–3). The <*alt2gxb entity name*>**.vho** file or <*alt2gxb module name*>**.vo** file is generated in the current project directory.

The Quartus II-generated ALT2GXB functional simulation library file references **stratixiigx_hssi wysiwyg atoms**.

Figure 3–3. ALT2GXB MegaWizard Plug-In Manager, Generate Simulation Model



Compiling Library Files for Stratix II GX RTL Functional Simulation in Verilog HDL

To compile the libraries necessary for functional simulation of a Verilog HDL design targeting a Stratix II GX device, type the following command at the VCS command prompt:

```
vcs -R <testbench>.v <alt2gxb simulation netlist>.vo -v \
stratixgx_hssi_atoms.v -v sgate.v -v 220model.v -v altera_mf.v ↵
```

Stratix II GX Gate-Level Timing Simulation

Stratix II GX gate-level timing simulation is similar to Arria GX gate-level timing simulation. The following example shows only the gate-level timing simulation for designs that include transceivers in Stratix II GX. To simulate the transceiver in Arria GX, you only have to replace the **stratixiigx_hssi** model file with the **arriagx_hssi** model file.

To perform a gate-level timing simulation of your design that includes a Stratix II GX transceiver, compile `stratixiigx_atoms` and `stratixiigx_hssi_atoms` into the **stratixiigx** and **stratixiigx_hssi** libraries, respectively.

The **stratixiigx_hssi_atoms** model file references the **lpm** and **sgate** libraries; so you must create these libraries to perform a simulation.

Compiling Library Files for Stratix II GX Gate-Level Timing Simulation in Verilog HDL

To compile the libraries necessary for timing simulation of a Verilog HDL design targeting a Stratix II GX device, type the following command at the VCS command prompt:

```
vcs -R <testbench>.v <gate-level netlist>.vo -v stratixiigx_atoms.v -v \
stratixiigx_hssi_atoms.v -v sgate.v -v 220model.v -v altera_mf.v \
+transport_int_delays +pulse_int_e/0 +pulse_int_r/0 \
+transport_path_delays +pulse_e/0 +pulse_r/0 ↵
```

Transport Delays

By default, the VCS software filters out all pulses that are shorter than the propagation delay between primitives. Turning on the transport delay options in the VCS software prevents the simulation tool from filtering out these pulses. Use the following options to ensure that all signal pulses are seen in the simulation results.

+transport_path_delays

Use this option when the pulses in your simulation are shorter than the delay within a gate-level primitive. You must include the `+pulse_e/number` and `+pulse_r/number` options.

+transport_int_delays

Use this option when the pulses in your simulation are shorter than the interconnect delay between gate-level primitives. You must include the `+pulse_int_e/number` and `+pulse_int_r/number` options. The `+transport_path_delays` and `+transport_int_delays` options are also used by default in the NativeLink feature for gate-level timing simulation.

For more information about either of these options, refer to the *VCS User Guide* installed with the VCS software.

The following VCS software command shows the command-line syntax to perform a post-synthesis simulation with the device family library:

```
vcs -R <testbench>.v <gate-level netlist>.v -v <Altera device family library>.v \
+transport_int_delays +pulse_int_e/0 +pulse_int_r/0 +transport_path_delays \
+pulse_e/0 +pulse_r/0 ↵
```

Using NativeLink with the VCS or VCS-MX Software

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools and allows you to run VCS or VCS-MX within the Quartus II software.

Setting Up the EDA Tool Executable Location

Before using the NativeLink feature, you must specify the path to your simulation tool. To specify the path, perform the following steps:

1. On the Tools menu, click **Options.** The **Options** dialog box appears.

2. In the **Category** list, select **EDA Tool Options**. The **EDA Tool Options** dialog box appears.
3. Double-click the entry under the **Location of Executable** column.
4. Type or browse to the directory containing the executables of your EDA tool. For the VCS simulator tool, the executable path is at <*VCS_Installed_Directory*>**/bin**. For the VCS-MX simulator tool, the executable path is at <*VCS-MX_Installed_Directory*>**/bin**.
5. Click **OK**.

You can also specify the path to the simulator's executables by using the `set_user_option` Tcl command:

```
set_user_option -name EDA_TOOL_PATH_VCS <path to executables> ↵
```

Performing an RTL Simulation Using NativeLink

To run an RTL functional simulation automatically with the VCS or VCS-MX software in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, under **EDA Tool Settings**, select **Simulation**. The **Simulation** dialog box appears.
3. In the **Tool name** list, select **VCS or VCS MX**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
4. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.

   For more information about setting up a testbench with NativeLink, refer to "Setting Up a Testbench" on page 3–26.
5. (For VCS-MX only) If you have compiled libraries using the EDA Simulation Library Compiler, perform the following steps:
   a. On the **Simulation** page, click **More EDA Netlist Writer Settings**. The **More EDA Netlist Writer Settings** dialog box appears.
   b. Under **Existing option settings**, click **Location of User-Compiled Simulation Library**.
   c. Under **Setting**, type the path that contains the pre-compiled libraries that are generated from the EDA Simulation Library Compilation tool; for example, <*design_path*>**/simulation/scsim**.
6. Click **OK.**
7. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration** to perform an analysis and elaboration. This command collects all your file name information and builds your design hierarchy in preparation for simulation.
8. On the Tools menu, point to **Run EDA Simulation Tool** and click **EDA RTL Simulation** to automatically launch the VCS or VCS-MX software, compile all necessary design files, and complete a simulation.

9. If you want to generate only the Tcl script without using the GUI after the NativeLink process, perform the following steps:

   a. On the **Simulation** page, click **More NativeLink Settings**. The **More NativeLink Settings** dialog box appears.

   b. Under **Existing option settings**, click **Generate third-party EDA tool command scripts without running the EDA tool**.

   c. Under **Setting**, click **On**.

   For VCS software, if you turn on this option and run NativeLink, files such as **script_file.sh** and the option file (<*revision_name*>**_rtl.vcs**) are generated without showing the result in the GUI. You can then run the simulation by typing the following command:

   ```
   sh script_file.sh ↵
   ```

   For VCS-MX software, if you turn on this option and run NativeLink, the file <*project_name*>**_vcsmx_rtl_**<*vhdl/verilog*>**.tcl** is generated without showing the result in the GUI. You can then run the simulation by typing the following command:

   ```
   quartus_sh -t <project_name>_vcsmx_rtl_<vhdl/verilog>.tcl ↵
   ```

Performing a Gate-Level Timing Simulation Using NativeLink

To automatically run a gate-level timing simulation with the VCS or VCS-MX software in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears.
3. In the **Tool name** list, select **VCS** or **VCS MX**.
4. You can modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
5. To perform a gate level simulation after each full compilation, turn on **Run gate-level simulation automatically after compilation**.
6. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.
7. (For VCS-MX only) If you have compiled libraries using the EDA Simulation Library Compiler, perform the following steps:

   a. On the **Simulation** page, click **More EDA Netlist Writer Settings**. The **More EDA Netlist Writer Settings** dialog box appears.

   b. Under **Existing option settings**, click **Location of User-Compiled Simulation Library**.

   c. Under **Setting**, type the path that contains the pre-compiled libraries that are generated from the EDA Simulation Library Compilation tool; for example, <*design_path*>**/simulation/scsim**.

8. Click **OK**.
9. Perform a full compilation. On the Processing menu, click **Start Compilation**.

10. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer** to generate a simulation netlist of your design.

If you have run full compilation after you set the EDA Tool Settings, the **Start EDA Netlist Writer** step is not required.

11. On the Tools menu, point to **Run EDA Simulation Tool** and click **EDA Gate Level Simulation** to automatically launch the VCS or VCS-MX software, compile all necessary design files, and complete a simulation.

    If multi-corner Timing Analysis is performed, a dialog box appears, asking you to select the timing corner to run simulation. Select the timing corner and click **Run**.

12. If you want to run VCS or VCS-MX in command-line mode when running it automatically after full compilation, perform the following steps:
    a. On the Simulation page, click **More NativeLink Settings**. The **More NativeLink Settings** dialog box appears.
    b. Under **Existing option settings**, click **Launch third-party EDA tool in command-line mode**.
    c. Under Setting, click **On**.
    d. Click **OK**.
13. If you want to generate only the Tcl script without using the GUI after the NativeLink process, you can perform the following steps:
    a. On the **Simulation** page, click **More NativeLink Settings**. The **More NativeLink Settings** dialog box appears.
    b. Under **Existing option settings**, click **Generate third-party EDA tool command scripts without running the EDA tool**.
    c. Under **Setting**, click **On**.

    For VCS software, if you turn on this option and run NativeLink, files such as **script_file.sh** and the option file (<*revision_name*>**_gate.vcs**) are generated without showing the result in the GUI. You can then run the simulation by typing the following command:

    ```
    sh script_file.sh ↵
    ```

    For VCS-MX software, if you turn on this option and run NativeLink, the file <*project_name*>**_vcsmx_gate_**<*vhdl/verilog*>**.tcl** is generated without showing the result in the GUI. You can then run the simulation by typing the following command:

    ```
    quartus_sh -t <project_name>_vcsmx_gate_<vhdl/verilog>.tcl ↵
    ```

Setting Up a Testbench

You can automatically launch your EDA simulator tool, compile your design files and testbench files, and perform a simulation automatically using the NativeLink feature.

To set up NativeLink with a testbench, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. Click the "+" icon to expand **EDA Tool Settings** and select **Simulation**. The **Simulation** page appears.
3. Under **NativeLink settings**, select **None** or **Compile test bench**. Table 3–8 shows the NativeLink settings.

Table 3–8. NativeLink Settings

Settings	Description
None	Compile simulation models and design files.
Compile test bench	NativeLink compiles simulation models, design files, testbench files, and starts simulation.

4. If you select **Compile test bench**, select your testbench setup from the **Compile test bench** list. You can use different testbench setups to specify different testbench files for different test scenarios. If there are no testbench setups entered, create a testbench setup by performing the following steps:
   a. Click **Test Benches**. The **Test Benches** dialog box appears.
   b. Click **New**. The **New Test Bench Settings** dialog box appears.
   c. In the **Test bench name** box, type in the testbench setup name, which is used to identify the different testbench setups.
   d. In the **Top level module** box, type in the top-level entity name. For example, for a Quartus II generated Verilog HDL testbench, type <*Vector Waveform File name*>`_vlg_vec_tst`.
   e. In the **Instance** box, type the full instance path to the top level of your FPGA design. For example, for a Quartus II generated Verilog HDL testbench, type `i1`.
   f. Under **Simulation period**, select **Run simulation until all vector stimuli are used**. If you select **End simulation at**, specify the simulation end time and the time unit.
   g. Under **Test bench files**, browse and add all your testbench files in the **File name** box. Use the **Up** and **Down** buttons to reorder your files. The script used by NativeLink compiles the files in order from top to the bottom.

You can also specify the library name and the HDL version to compile the testbench file. NativeLink compiles the testbench to the library name of the HDL specified version.

   h. Click **OK**.
   i. In the **Test Benches** dialog box, click **OK**.

Creating a Testbench

In the Quartus II software, you can create a Verilog HDL or VHDL testbench from a Vector Waveform File (**.vwf**). The generated testbench includes the behavior of the input stimulus and applies it to your instantiated top-level FPGA design. To create a Verilog HDL or VHDL testbench, perform the following steps:

1. On the File menu, click **Open**. The **Open** dialog box appears.
2. Click the **Files of type** arrow and select **Waveform/Vector Files**. Select your file.
3. Click **Open**.
4. On the File menu, click **Export**. The **Export** dialog box appears.
5. Click the **Save as type** arrow and select **VHDL Test Bench File (*.vht)** or **Verilog Test Bench File (*.vt)**.
6. You can turn on **Add self-checking code to file** to check your simulation results against your **.vwf** file.
7. Click **Export**.

Generating a Simulation Script from the EDA Netlist Writer

Beginning in the Quartus II software version 9.0, you can generate the simulation script (such as the DO file, Tcl script, and option file) when you run the EDA Netlist Writer. You can use the simulation script to run your simulation at the stated simulator in stand-alone.

For generating the simulation script via EDA Netlist Writer, perform the following steps:

1. Click **More EDA Netlist Writer Settings**. The **More EDA Netlist Writer Settings** dialog box appears.
2. In the **Existing options** list, click **Generate third-party EDA tool command script for RTL simulation** (if you want to generate an RTL simulation script) or click **Generate third-party EDA tool command script for Gate-Level simulation** (if you want to generate a Gate-Level simulation script).
3. Under **Setting**, click **On**.
4. Click **OK**.
5. Follow the steps in "Generating a Post-Synthesis Simulation Netlist for VCS" on page 3–29.

To generate the simulation script with the EDA Netlist Writer, type the following command in the command line:

```
quartus_eda --simulation --tools=<Your decided tool> --format=<vhdl or verilog>
--gen_script=<rtl or gate_level> <Project Name> -c <Revision name> ↵
```

Generating a Timing .vcd File for PowerPlay

To generate a timing Value Change Dump (***.vcd**) file for PowerPlay, you must first generate a VCD script in the Quartus II software, and then run the VCD script from the VCS software. This timing **.vcd** file can then be used by PowerPlay for power estimation.

Perform the following steps to generate timing VCD scripts in the Quartus II software:

1. In the Quartus II software, on the Assignments menu, click **Settings**. The Settings database appears.
2. In the **Settings** dialog box, on the **Simulator Settings** page, in the **Tool Name** list, choose the appropriate third-party simulation tool (for example, VCS), and turn the **Generate Value Change Dump File Script** option on.
3. To generate the VCD script file, perform a full compilation.

Perform the following steps to generate a timing **.vcd** file in the VCS software:

1. In the VCS software, before compiling and simulating your design, include the script in your testbench file where the design under test (DUT) is instantiated:

```
include <revision_name>_dump_all_vcd_nodes.v ↵
```

Include the script within the testbench module block. If you include the script outside of the testbench module block, syntax errors occur during compilation.

2. Run the simulation using the VCS command as usual. Exit the VCS software when the simulation is done and the *<revision_name>***.vcd** file is generated in the simulation directory.

For more detailed information about using the timing **.vcd** file for power estimation, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Viewing a Waveform from a .vpd or .vcd File

A **.vpd** file is automatically generated when your simulation is done. The **.vpd** file is not readable. It is used for generating the waveform view through VirSim or DVE. You can view your waveform result in VirSim or DVE if you have created a **.vpd** or **.vcd** file.

To view a waveform from a **.vpd** file through VirSim, perform the following steps:

1. Type `virsim` on a command line. The **VirSim Hierarchy** dialog box appears.
2. On the File menu, click **Open**. The **Open File** dialog box appears.
3. In the **Directories** field, browse to the directory that contains your **.vpd** file (for example, **inter.vpd**).
4. Double-click the **.vpd** file.
5. On the Window menu, click **Waveform**. The **VirSim Waveform** dialog box appears.

6. Move the signals that you want to observe from the **VirSim Hierarchy** dialog box to the **VirSim Waveform** dialog box. View the waveform.

To view a waveform from a **.vpd** file through DVE, perform the following steps:

1. Type `dve` on a command line. The **DVE** dialog box appears.
2. On the File menu, click **Open Database**. The **Open Database** dialog box appears.
3. Browse to the directory that contains your **.vpd** file (for example, **inter.vpd**).
4. Double-click the **.vpd** file.
5. In the **DVE** dialog box, select the signals that you want to observe from the Hierarchy.
6. On the Signal menu, click **Add To Waves**.
7. Click **New Wave View**. The waveform appears.

You cannot view a waveform from a **.vcd** file in VirSim or DVE directly. The **.vcd** file must first be converted to a **.vpd** file. Perform the following steps:

1. Use the `vcd2vpd` command to convert the file. For example, type the following on a command-line:

```
vcd2vpd <example>.vcd <example>.vpd ↵
```

2. After you convert the **.vcd** file to a **.vpd** file, follow the procedures for viewing a waveform from a **.vpd** file through VirSim or DVE.

You can also convert your **.vpd** file to a **.vcd** file by using the `vpd2vcd` command.

Scripting Support

You can run procedures and create settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For detailed information about scripting command options, refer to the Qhelp utility.

To start the **Qhelp** utility, type this command:

```
quartus_sh --qhelp ↵
```

Generating a Post-Synthesis Simulation Netlist for VCS

You can use the Quartus II software to generate a post-synthesis simulation netlist with Tcl commands or with a command at a command prompt.

Tcl Commands

Type the following Tcl commands to generate a post-synthesis simulation netlist:

```
set_global_assignment -name EDA_SIMULATION_TOOL "VCS" ↵
set_global_assignment -name EDA_GENERATE_FUNCTIONAL_NETLIST ON ↵
```

Command Prompt

Type the following command to generate a simulation output file for the VCS software simulator; specify VHDL or Verilog HDL for the format:

```
quartus_eda <project name> --simulation=on --format=<format> \
--tool=vcs --functional ↵
```

Generating a Gate-Level Timing Simulation Netlist for VCS

You can use the Quartus II software to generate a gate-level timing simulation netlist with Tcl commands or with a command at a command prompt.

Tcl Commands

Type the following Tcl command to generate a gate-level timing simulation netlist:

```
set_global_assignment -name EDA_SIMULATION_TOOL "VCS" ↵
```

Command Prompt

Type the following command to generate a simulation output file for the VCS software simulator. Specify VHDL or Verilog HDL for the format.

```
quartus_eda <project name> --simulation=on --format=<format> --tool=vcs ↵
```

Conclusion

You can use the Synopsys VCS or VCS-MX software in your Altera FPGA design flow to easily and accurately perform RTL functional simulations, post-synthesis simulations, and gate-level functional timing simulations. The seamless integration of the Quartus II software and VCS or VCS-MX software make this simulation flow an ideal method for fully verifying an FPGA design.

Referenced Documents

This chapter references the following documents:

- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Installation and Licensing for UNIX and Linux Workstation Manual*
- *Quartus II Installation and Licensing for Windows Manual*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *VCS User Guide*

Document Revision History

Table 3–9 shows the revision history for this chapter.

Table 3–9. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Added support for Synopsys VCS-MX software. ■ Changed chapter title to "Synopsys VCS and VCS-MX Support". ■ Major revision to "Compile Libraries Using the EDA Simulation Library Compiler" on page 3–4. ■ Major revision to "RTL Functional Simulations" on page 3–7. ■ Added Table 3–4 on page 3–10 and Table 3–5 on page 3–11. ■ Added new section "Using DVE" on page 3–18. ■ Added new section "Generating a Simulation Script from the EDA Netlist Writer" on page 3–27. ■ Added new section "Viewing a Waveform from a .vpd or .vcd File" on page 3–28.	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	■ Added "Compile Libraries Using the EDA Simulation Library Compiler" on page 3–3. ■ Added information about the `--simlib_comp` utility. ■ Updated entire chapter using 8½" × 11" chapter template. ■ Minor editorial updates.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Updated Table 3–1. ■ Updated Figure 3–1. ■ Updated Table 3–3. ■ Updated "Generating a Timing Netlist with Different Timing Models" on page 3–7. ■ Added "Disable Timing Violation on Registers" on page 3–8. ■ Updated "Simulating Designs that Include Transceivers" on page 3–10. ■ Updated "Performing a Gate-Level Timing Simulation Using NativeLink" on page 3–15. ■ Added "Generating a Timing VCD File for PowerPlay" on page 3–17.	Updated for the Quartus II software version 8.0.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

4. Cadence NC-Sim Support

QII53003-9.0.0

Introduction

This chapter is a "getting started" guide to using the Cadence Incisive verification platform software in Altera® FPGA design flows. The Incisive verification platform software includes NC-Sim, NC-Verilog, NC-VHDL, Verilog HDL, and VHDL desktop simulators. This chapter provides step-by-step explanations of the basic NC-Sim, NC-Verilog, and NC-VHDL functional, post-synthesis, and gate-level timing simulations. It also describes the location of the simulation libraries and how to automate simulations.

This chapter contains the following topics:

- "Software Requirements"
- "Simulation Flow Overview" on page 4–2
- "RTL Functional Simulation" on page 4–7
- "Post-Synthesis Simulation" on page 4–17
- "Gate-Level Timing Simulation" on page 4–19
- "Simulating Designs that Include Transceivers" on page 4–24
- "Using the NativeLink Feature with NC-Sim" on page 4–29
- "Generating a Timing VCD File for PowerPlay" on page 4–35
- "Viewing a Waveform from a .trn File" on page 4–36
- "Scripting Support" on page 4–37

Software Requirements

You must first install the Quartus® II software before using it with the Cadence Incisive verification platform software. The Cadence interface is installed automatically when you install the Quartus II software on your computer.

Table 4–1 shows the Cadence NC simulator versions compatible with specific Quartus II software versions.

Table 4–1. Compatibility Between Software Versions (Part 1 of 2)

Quartus II Software	Cadence NC Simulators (UNIX)	Cadence NC Simulators (PC)	Cadence NC Simulators (Linux)
Version 9.0	Version 6.2	—	Version 6.2
Version 8.1	Version 6.2	—	Version 6.2
Version 8.0	Version 6.2	—	Version 6.2
Version 7.2	Version 6.1 p001	—	Version 6.1 p001
Version 7.1	Version 5.83 p003	—	Version 5.83 p003
Version 7.0	Version 5.82 p001	—	Version 5.82 p001
Version 6.1	Version 5.82 p001	Version 5.4 s011	Version 5.82 p001

Table 4–1. Compatibility Between Software Versions (Part 2 of 2)

Quartus II Software	Cadence NC Simulators (UNIX)	Cadence NC Simulators (PC)	Cadence NC Simulators (Linux)
Version 6.0	Version 5.5 s012	Version 5.4 s011	Version 5.5 s012
Version 5.1	Version 5.4 s011	Version 5.4 s011	Version 5.4 s011
Version 5.0	Version 5.4 s004	Version 5.4 p001	Version 5.4 s004
Version 4.2	Version 5.1 s017	Version 5.1 s017	Version 5.1 s017
Version 4.1	Version 5.1 s012	Version 5.1 s010	Version 5.0 p001
Version 4.0	Version 5.0 s005	Version 5.0 s006	Version 5.0 p001

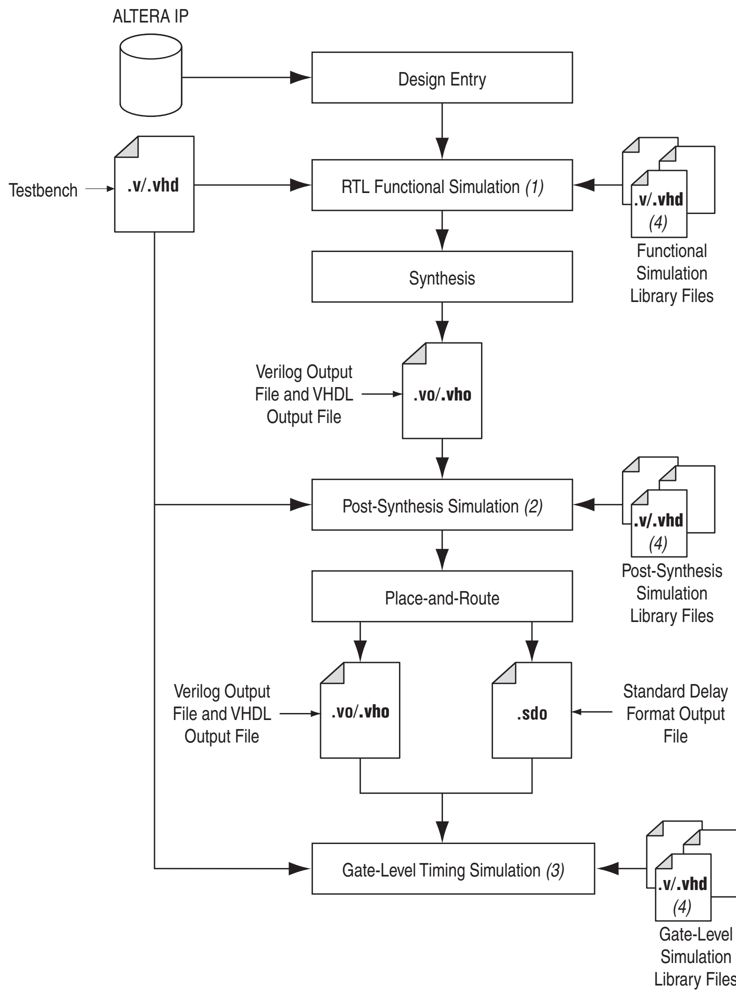
Simulation Flow Overview

The Incisive platform software supports the following simulation flows:

- RTL Functional Simulation
- Post-Synthesis Simulation
- Gate-Level Timing Simulation
- Using the NativeLink Feature with NC-Sim

Figure 4–1 shows the Quartus II software and Cadence design flow.

Figure 4–1. Quartus II Software Design Flow with Cadence NC Simulators *(Note 1)*, *(2)*, *(3)*, *(4)*



Notes to Figure 4–1:

(1) RTL functional simulation is performed before a gate-level timing simulation or post-synthesis simulation. RTL functional simulation verifies the functionality of the design before synthesis and place-and-route. If you are performing a functional simulation with NativeLink, you must complete analysis and elaboration first.

(2) A post-synthesis simulation verifies the functionality of a design after synthesis has been performed.

(3) Gate-level timing simulation is a post place-and-route simulation to verify the operation of the design after the worst-case timing delays have been calculated.

(4) All of these simulation libraries are located at *<quartus installation directory>***eda\sim_lib**.

RTL functional simulation verifies the functionality of your design. When you perform an RTL functional simulation with Cadence Incisive simulators, you use your design files (Verilog HDL or VHDL) and the models provided with the Quartus II software. These Quartus II models are required if your design uses the library of parameterized modules (LPM) functions or Altera-specific megafunctions. Refer to "RTL Functional Simulation" on page 4–7 for more information about how to perform this simulation.

A post-synthesis simulation verifies the functionality of a design after synthesis has been performed. You can create a post-synthesis netlist (**.vo** or **.vho**) in the Quartus II software and use this netlist to perform a post-synthesis simulation with the Incisive simulator. Refer to "Post-Synthesis Simulation" on page 4–17 for more information about how to perform this simulation.

After performing place-and-route, the Quartus II software generates a Verilog Output File (**.vo**) or VHDL Output File (**.vho**) and a Standard Delay Output file (**.sdo**) for gate-level timing simulation. The netlist files map your design to architecture-specific primitives. The **.sdo** file contains the delay information of each architecture primitive and routing element specific to your design. Together, these files provide an accurate simulation of your design with the selected Altera FPGA architecture. Refer to "Gate-Level Timing Simulation" on page 4–19 for more information about how to perform this simulation.

Operation Modes

In the NC simulators, you can use either the GUI mode or the command-line mode to simulate your design.

To start the Incisive simulators in GUI mode in a PC or a UNIX environment, at a command prompt, type the following:

```
nclaunch ↵
```

To simulate in command-line mode, use the programs shown in Table 4–2.

Table 4–2. Command-Line Programs

Program	Function
ncvlog or **ncvhdl**	NC-Verilog (**ncvlog**) compiles your Verilog HDL code into a Verilog Syntax Tree (**.vst**) file. **ncvlog** also performs syntax and static semantics checks. NC-VHDL (**ncvhdl**) compiles your VHDL code into a VHDL Syntax Tree (**.ast**) file. **ncvhdl** also performs syntax and static semantics checks.
ncelab	NC-Elab (**ncelab**) elaborates the design. **ncelab** constructs the design hierarchy and establishes signal connectivity. This program also generates a Signature File (**.sig**) and a Simulation SnapShot File (**.sss**).
ncsim	NC-Sim (**ncsim**) performs mixed-language simulation. This program is the simulation kernel that performs event scheduling and executes the simulation code.

Quartus II Software and NC Simulation Flow Overview

This section provides an overview of the Quartus II software and Cadence NC simulation flow. More detailed information is provided in "RTL Functional Simulation" on page 4–7, "Post-Synthesis Simulation" on page 4–17, and "Gate-Level Timing Simulation" on page 4–19.

For high-speed simulation, you must select **ps** in the **Resolution** list for your simulator resolutions. If you choose slower than **ps**, the high-speed simulation might fail.

Perform the following steps:

1. Set up your working environment (UNIX only).

   You must set several environment variables in UNIX to establish an environment that facilitates entering and processing designs.

2. Create user libraries.

   Create a file that maps logical library names to their physical locations. These library mappings include your working directory and any design-specific libraries; for example, Altera LPM functions or megafunctions.

3. Compile source code and testbenches.

   Compile your design files at the command-line using **ncvlog** (Verilog HDL files) or **ncvhdl** (VHDL files) or, on the Tools menu, click **Verilog Compiler** or **VHDL Compiler** in NCLaunch. During compilation, the NC software performs syntax and static semantic checks. If no errors are found, compilation produces an internal representation for each HDL design unit in the source files. By default, these intermediate objects are stored in a single, packed, library database file in your working directory.

4. Elaborate your design.

   Before you can simulate your model, you must define the design hierarchy in a process called "elaboration". Use **ncelab** in command-line mode or, on the Tools menu in NCLaunch, click **Elaborator**.

5. Add signals to your waveform.

   Before simulating, specify which signals to view in your waveform using a simulation history manager (SHM) database.

6. Simulate your design.

   Run the simulator with the **ncsim** program (command-line mode) or by clicking **Run** in the SimVision Console window.

Compile Libraries Using the EDA Simulation Library Compiler

The EDA Simulation Library Compiler is used to compile Verilog HDL and VHDL simulation libraries for all Altera devices and supported third-party simulators. You can use this tool to compile all libraries required by RTL and gate-level simulation.

When the compilation targets third-party simulation tools such as ModelSim, VCS-MX, Active-HDL, Riviera-PRO, and NC-Sim, the compiled libraries are kept in either the directory you set or the default directory. When you perform the simulation using these simulators, you can use or re-use the compiled libraries and avoid the overhead associated with redundant library compilations.

However, if the Synopsys VCS software performs the compilation while running the EDA Simulation Library Compiler, the option files (**simlib_comp.vcs**) are generated after compilation. You can then include your design and testbench files in the option files and invoke them with the `vcs` command.

Before using this option, ensure the appropriate simulation tools are already installed and their paths are specified. To specify the path, refer to "Setting Up NativeLink" on page 4–29.

Run the EDA Simulation Library Compiler through the GUI

Starting with the Quartus II software 9.0 release, the EDA Simulation Library Compiler contains a GUI. To compile libraries with the EDA Simulation Library Compiler GUI, perform the following steps:

1. On the Tools menu, click **EDA Simulation Library Compiler**. The **EDA Simulation Library Compiler** dialog box appears.
2. In the **Tool name** entry box under **EDA simulation tool**, select a simulation tool.

   The **Executable location** box displays the location of the simulation tool you specified. This location must be set before running the EDA Simulation Library Compiler.
3. Under **Library families**, select one or more device families for your design compilation and move them to **Selected families** box.
4. Under **Library language**, select **VHDL**, **Verilog**, or both.
5. In the **Output directory** field, specify a location in which to store the compiled libraries or option files.
6. Click **Start Compilation**.

For example, if you want to simulate a Verilog HDL design on a Stratix® II device in the NC-Sim simulator, do the following:

- Select **NC-Sim** in the **Tool name** field.
- Move **Stratix II** from the **Available families** list to the **Selected families** list.
- Select **Verilog** in the **Library language** field.
- Specify a location in the **Output directory** field in which to keep the option file.
- Click **Start Compilation**.

When the EDA Simulation Library Compiler finishes, all required libraries are compiled and stored in the output location you specified. The next time you perform simulation in NC-Sim, you only have to compile your design and testbench files. You do not have to compile the Altera libraries again.

If you use NativeLink to run the simulation, refer to "Using the NativeLink Feature with NC-Sim" on page 4–29.

Run EDA Simulation Library Compiler In Command Line

You can run the EDA Simulation Library Compiler in the command line. Type the following command:

```
quartus_sh --simlib_comp -family <device> -tool <simulation tool name>
-language <language> -directory <directory> ↵
```

For more information about the command's options and how to define them, type the following command:

```
quartus_sh --help=simlib_comp ↵
```

RTL Functional Simulation

The following sections provide detailed instructions for performing RTL functional simulation using the Quartus II software and the Cadence Incisive platform software tools.

Create Libraries

Before simulating with the Incisive simulator, you must set up libraries with a file named **cds.lib**. The **cds.lib** file is an ASCII text file that maps logical library names—for example, your working directory or the location of resource libraries such as models for LPM functions—to their physical directory paths. When you run the Incisive simulator, the tool reads **cds.lib** to determine which libraries are accessible and where they are located. There is also a default **cds.lib** file, which you can modify for your project settings.

You can use more than one **cds.lib** file. For example, you can have a project-wide **cds.lib** file that contains library settings specific to a project, such as technology or cell libraries and a user **cds.lib** file.

The following sections describe how to create and edit a **cds.lib** file:

- "Basic Library Setup" on page 4–7
- "LPM Functions, Altera Megafunctions, and Altera Primitive Library Setup" on page 4–9

Basic Library Setup

You can create a **cds.lib** file with any text editor. The following examples show how you use the `DEFINE` statement to bind a library name to its physical location. The logical and physical names can be the same or you can select different names. The `DEFINE` statement usage is:

```
DEFINE <library name> <physical directory path>
```

A simple **cds.lib** file for Verilog HDL contains the lines shown in Example 4–1.

Example 4–1. cds.lib File for Verilog HDL

```
DEFINE lib_std /usr1/libs/std_lib
DEFINE worklib ../worklib
```

Using Multiple cds.lib Files

Use the `INCLUDE` or `SOFTINCLUDE` statement to reference another **cds.lib** file within a **cds.lib** file. The syntax is:

```
INCLUDE <path to another cds.lib>
```

or

```
SOFTINCLUDE <path to another cds.lib>
```

For VHDL or mixed-language simulation, in addition to the `DEFINE` statements, you must include the default **cds.lib** file (included with NC-Sim). The syntax for including the default **cds.lib** file is:

```
INCLUDE <path to NC installation>/tools/inca/files/cds.lib
```

or

```
INCLUDE $CDS_INST_DIR/tools/inca/files/cds.lib
```

The default **cds.lib** file, provided with NC tools, contains a `SOFTINCLUDE` statement to include other **cds.lib** files, such as **cdsvhdl.lib** and **cdsvlog.lib**. These files contain library definitions for IEEE libraries and Synopsys libraries.

Create a cds.lib File in Command-Line Mode

To create a **cds.lib** file at the command prompt, perform the following steps:

1. Create a directory for the work library and any other libraries you require by typing the following at a command prompt:

   ```
   mkdir <library name> ↵
   ```

   For example: `mkdir worklib` ↵

2. Using a text editor, create a **cds.lib** file and add the following line to it:

   ```
   DEFINE <library name> <physical directory path>
   ```

   For example: `DEFINE worklib ./worklib` ↵

Create a cds.lib File in GUI Mode

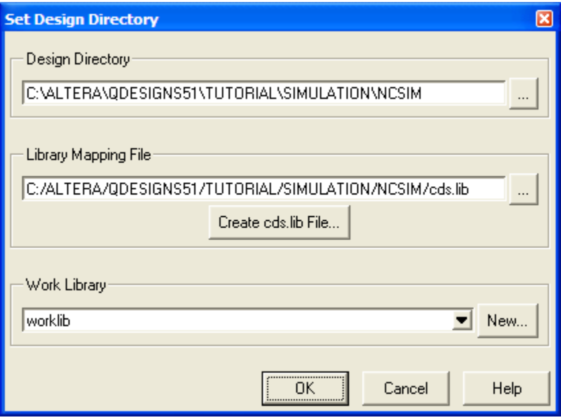
To create a **cds.lib** file using the GUI, perform the following steps:

1. To run the GUI, at the command line, type the following command:

   ```
   nclaunch ↵
   ```

2. If the NCLaunch window is not in multiple step mode, on the File menu, click **Switch to Multiple Step**.
3. Change your design directory. On the File menu, click **Set Design Directory**. The **Set Design Directory** dialog box appears (Figure 4–2).

Figure 4–2. Creating a Work Directory in GUI Mode



4. To navigate to your design directory, click **Browse**.
5. Click **Create cds.lib File**. In the **New cds.lib File** dialog box, click **Save** and choose the libraries to include.
6. Under **Work Library**, click **New**.
7. Enter your new work library name; for example, `worklib`.

8. Click **OK**. The new library is displayed under **Work Library**. Figure 4–2 shows an example using the directory name **worklib**.
9. Repeat steps 7 and 8 for each functional simulation library. For example: `lpm`, `altera_mf`, `altera`.
10. In the **Set Design Directory** dialog box, click **OK**.

You can edit your libraries by editing the **cds.lib** file. To edit the **cds.lib** file, in the right side of the window, right-click the **cds.lib** filename and choose **Edit**.

LPM Functions, Altera Megafunctions, and Altera Primitive Library Setup

Altera provides behavioral descriptions for LPM functions, Altera-specific megafunctions, and Altera primitives. You can implement the megafunctions in a design using the Quartus II MegaWizard™ Plug-In Manager, or by instantiating them directly from your design file. If your design uses LPM functions, Altera megafunctions, or Altera primitives, you must set up resource libraries so you can simulate your design in the Incisive simulator.

Many LPM functions and Altera megafunctions use memory files. You must convert the memory files into a format the Incisive tools can read before simulating. To covert the memory files, follow the instructions in “Compile Source Code and Testbenches” on page 4–11.

Altera provides megafunction behavioral descriptions in the files shown in Table 4–3. These library files are located in the <*path to Quartus II installation*>/**eda**/**sim_lib** directory.

For more information about LPM functions and Altera megafunctions, refer to the Quartus II Help.

Table 4–3. Megafunction Behavioral Description Files (Part 1 of 2)

Library Description	Verilog HDL	VHDL
LPM	**220model.v**	**220model.vhd** *(1)* **220model_87.vhd** *(2)* **220pack.vhd**
Altera megafunction	**altera_mf.v**	**altera_mf.vhd** *(1)* **altera_mf_87.vhd** *(2)* **altera_components.vhd**
Altera primitives	**altera_primitives.v**	**altera_primitives.vhd** *(1)* **altera_primitives_components.vhd**
IP functional simulation models	**sgate.v**	**sgate.vhd** **sgate_pack.vhd**
ALTGXB	**stratixgx_mf.v** *(3)*	**stratixgx_mf.vhd** *(3)* **stratixgx_mf_components.vhd** *(3)*

Table 4–3. Megafunction Behavioral Description Files (Part 2 of 2)

Library Description	Verilog HDL	VHDL
ALT2GXB	**stratixiigx_hssi_atoms.v**, **arriagx_hssi_atoms.v** *(3)*, *(4)*	**stratixiigx_hssi_atoms.vhd** **stratixiigx_hssi_components.vhd** **arriagx_hssi_atoms.vhd** **arriagx_hssi_components.vhd** *(3)*, *(4)*
ALTGX	**stratixiv_hssi_atoms.v**	**stratixiv_hssi_atoms.vhd** **stratixiv_hssi_components.vhd**

Notes to Table 4–3:

(1) Use this model with VHDL 93.

(2) Use this model with VHDL 87.

(3) The ALT2GXB and ALTGXB library files require the **lpm** and **sgate** libraries.

(4) You must generate a functional simulation netlist for simulation.

If an `lpm` library does not exist, set up a library for LPM functions. Create a new directory and add the following line to your **cds.lib** file:

```
DEFINE lpm <path>/<directory name>
```

If an `altera_mf` library does not exist, set up a library for Altera megafunctions. Add the following line to your **cds.lib** file:

```
DEFINE altera_mf <path>/<directory name>
```

Megafunctions Requiring Atom Libraries

The following Altera megafunctions require device atom libraries to perform a functional simulation in a third-party simulator:

- ALTCLKBUF
- ALTCLKCTRL
- ALTDQ
- ALTDQS
- ALTDDIO_IN
- ALTDDIO_OUT
- ALTDDIO_BIDIR
- ALTUFM_NONE
- ALTUFM_PARALLEL
- ALTUFM_SPI
- ALTMEMMULT
- ALTREMOTE_UPDATE

For more information about these megafunctions, refer to the Literature: Megafunctions section of the Altera website.

The device atom library files are located in the following directory:

*<path to Quartus II installation>***/eda/sim_lib**

Compile Source Code and Testbenches

Compile your testbench and design files with **ncvlog** (for Verilog HDL files) and **ncvhdl** (for VHDL files). Both **ncvlog** and **ncvhdl** perform syntax checks and static semantic checks. A successful compilation produces an internal representation for each HDL design unit in the source files. By default, these intermediate objects are stored in a single, packed, library database file in your work library directory.

Compilation in Command-Line Mode

To compile from the command line, use one of the following commands:

- Verilog HDL:

  ncvlog <*options*> -work <*library name*> <*design files*> ↵

- VHDL:

  ncvhdl <*options*> -work <*library name*> <*design files*> ↵

You must create a work library before compiling.

If your design uses LPM, Altera megafunctions, or Altera primitives, you must also compile the Altera-provided functional models. The commands in Example 4–2 and Example 4–3 show an example of each.

Example 4–2. Creating a Work Directory in GUI Mode in Verilog HDL

```
ncvlog -WORK lpm 220model.v
ncvlog -WORK altera_mf altera_mf.v
ncvlog -WORK altera altera_primitives.v
```

Example 4–3. ncvlog -WORK altera altera_primitives.v in VHDL

```
ncvhdl -V93 -WORK lpm 220pack.vhd
ncvhdl -V93 -WORK lpm 220model.vhd
ncvhdl -V93 -WORK altera_mf altera_mf_components.vhd
ncvhdl -V93 -WORK altera_mf altera_mf.vhd
ncvhdl -V93 -WORK altera altera_primitives_components.vhd
ncvhdl -V93 -WORK altera altera_primitives.vhd
```

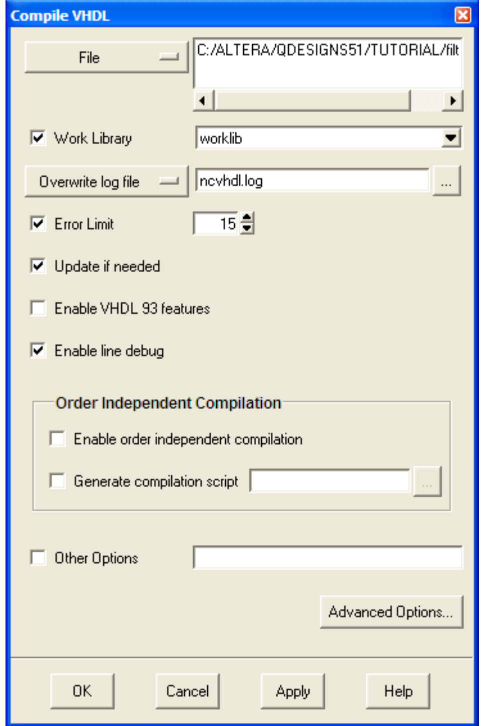
Compilation in GUI Mode

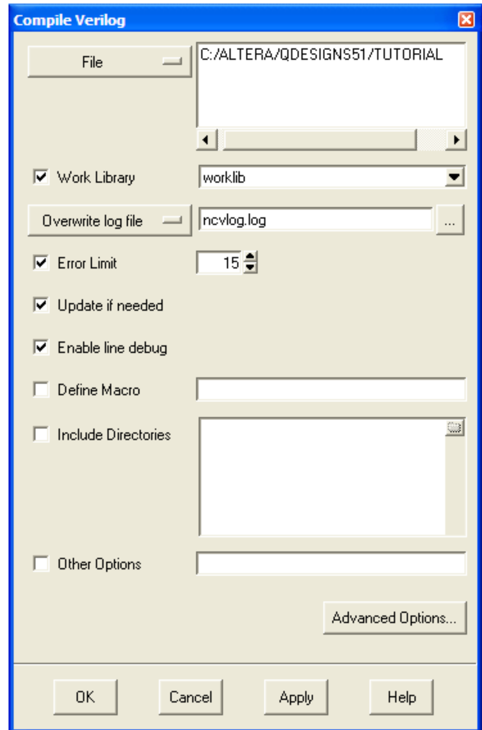
To compile using the NCLaunch GUI, perform the following steps:

1. In the NCLaunch window, right-click a library filename and click **NCVlog** (Verilog HDL) or **NCVhdl** (VHDL).

   Alternatively, on the Tools menu, click **Verilog Compiler** or **VHDL Compiler**. Figure 4–3 shows the **Compile Verilog** and **Compile VHDL** dialog boxes.

Figure 4–3. Compiling Verilog HDL and VHDL Files





2. To begin compilation, in the **Compile Verilog** or **Compile VHDL** dialog box, select the file. Click **OK**. The dialog box closes and returns you to **NCLaunch**.

The command-line equivalent argument is shown at the bottom of the **NCLaunch** window.

Elaborate Your Design

Before you can simulate your design, you must define the design hierarchy in a process called elaboration. When you use the Incisive simulator, you use the language-independent **ncelab** program to elaborate your design. The **ncelab** program constructs a design hierarchy based on the design's instantiation and configuration information, establishes signal connectivity, and computes initial values for all objects in the design. The elaborated design hierarchy is stored in a simulation snapshot, which is the representation of your design that the simulator uses to run the simulation. The snapshot is stored in the library database file, along with the other intermediate objects generated by the compiler and elaborator.

If you are running the NC-Verilog simulator with the single-step invocation method (`ncverilog`), and want to compile your source files and elaborate the design with one command, use the `+elaborate` option to stop the simulator after elaboration; for example:

```
ncverilog +elaborate test.v ↵
```


© March 2009 Altera Corporation

Elaboration in Command-Line Mode

To elaborate your Verilog HDL or VHDL design from the command line, use the following command:

```
ncelab [options][<library>.]<cell>[:<view>] ↵
```

You can set your simulation timescale using the `-TIMESCALE` <*arguments*> option. The following example elaborates a dual-port RAM with the time scale option:

```
ncelab -TIMESCALE 1ps/1ps worklib.lpm_ram_dp_test:entity ↵
```

If you specified a timescale of 1 ps in the Verilog HDL testbench, the `TIMESCALE` option is not necessary. Using a ps resolution ensures the correct simulation of your design.

If your design includes high speed signals, you might have to add the following pulse reject options with the `ncelab` command.

```
ncelab -TIMESCALE 1ps/1ps worklib.mydesign:entity -PULSE_R 0 -PULSE_INT_R 0 ↵
```

For more information about the pulse reject options, refer to the *SDF Annotate Guide* from Cadence.

To list the elements in your library and the available views, use the **ncls** program. The following command displays all of the cells and their views in your current **worklib** directory:

```
ncls -library worklib ↵
```

For more information about the **ncls** program, refer to the Cadence NC-Verilog Simulator Help or Cadence NC-VHDL Simulator Help.

Elaboration in GUI Mode

To elaborate using the GUI, perform the following steps:

1. In the right side of the NCLaunch window, expand your current work library.
2. Select and (if necessary) expand the entity or module name you want to elaborate.
3. Right-click the view you want to display. Click **NCElab**. The **Elaborate** dialog box appears. Optionally, on the Tools menu, click **Elaborator**.
4. In the **Other Options** box, set the simulation timescale by typing the following command :

   ```
   -TIMESCALE 1ps/1ps ↵
   ```

5. To begin elaboration, in the **Elaborate** dialog box, click **OK**. The dialog box closes and returns you to **NCLaunch**.

Add Signals to View

The SHM database is a Cadence proprietary waveform database that stores selected signals for viewing. To specify which signal to view, you must create a database. To create this database, add commands to your code, or create a Verilog Value Change Dump File (**.vcd**) to store the simulation history.

For more information about using a **.vcd** file, refer to the *Cadence NC-Sim User Manual* from Cadence included in the installation package.

Adding Signals in Command-Line Mode

To create an SHM database, specify the system tasks described in Table 4–4 in your Verilog HDL code.

For VHDL, you can use the Tcl command interface or C function calls to add signals to a database. Refer to the Cadence documentation included in the installation package for details.

Table 4–4. SHM Database System Tasks

System Task	Description
`$shm_open("<filename>.shm");`	Opens a database. If you do not specify a filename, the default **waves.shm** database opens. If a database with the specified name does not exist, it is created for you.
`$shm_probe("[A|S|C]");`	Probe signals. You can specify the signals to probe; if you do not specify signals, the default is all ports in the current scope. `A` probes all nodes in the current scope. `S` probes all nodes below the current scope. `C` probes all nodes below the current scope and in libraries.
`$shm_save;`	Saves the database.
`$shm_close;`	Closes the database.

Example 4–4 shows an example of how to add signals to an SHM database.

Example 4–4. Elaborating the Design

```
initial
   begin
      $shm_open ("waves.shm");
      $shm_probe ("AS");
    end
```

You can insert this code sample into your Verilog HDL file. It is applicable only for Verilog HDL files. For more information about these system tasks, refer to the *Cadence NC-Sim User Manua*l included in the installation.

Adding Signals in GUI Mode

To add signals in GUI mode, perform the following steps:

1. In the NC-Sim software, load the design.

   a. In the **NCLaunch** window, click the "**+**" icon to expand the **Snapshots** directory.

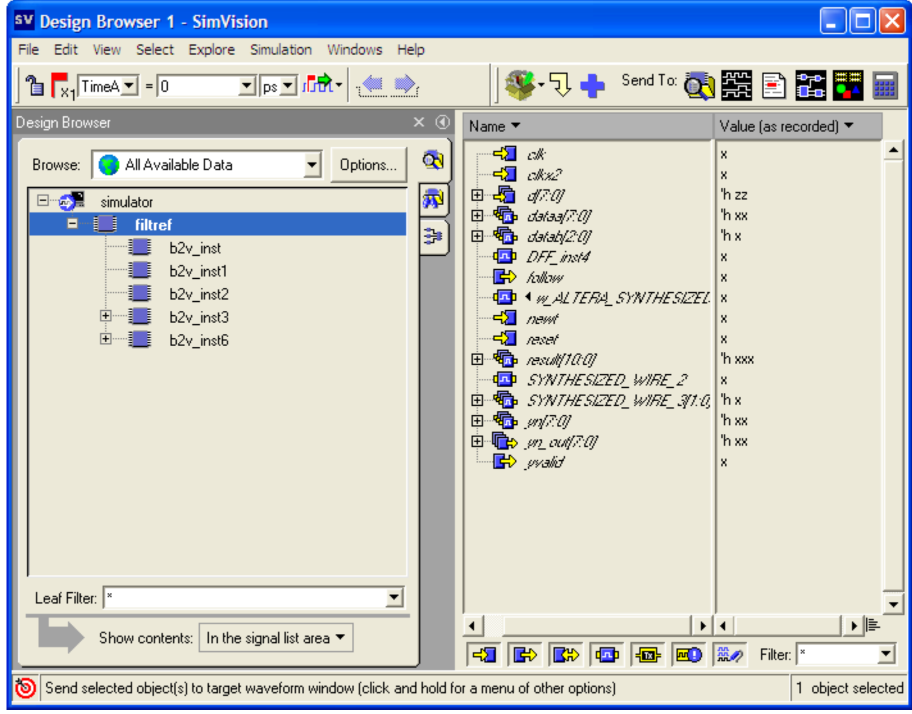
   b. Right-click the **lib.cell:view** you want to simulate. Click **NCSim**.

   c. In the **Simulate** dialog box, click **OK**.

   After you load the design, the SimVision Console and SimVision Design Browser windows appear. Figure 4–4 shows the SimVision Design Browser window.

2. To display the signal names, in the Design Browser window, on the left side of the window select a module (Figure 4–4).

Figure 4–4. SimVision Design Browser



3. To send the selected signals to the Waveform Viewer, perform one of the following steps:

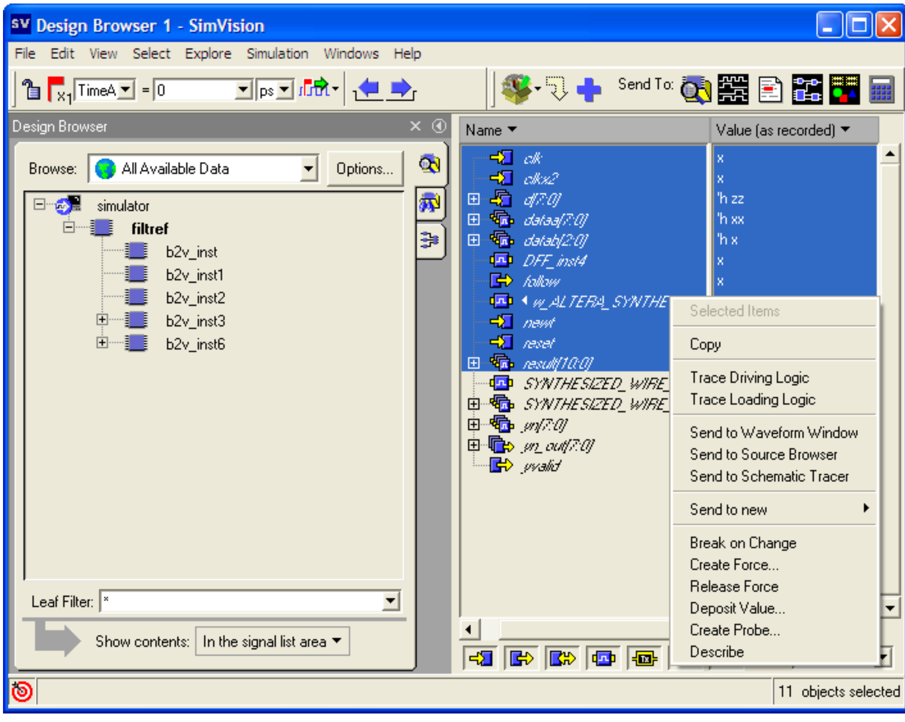
   Select a group of signals from the right side of the Design Browser window. In the **Send To** toolbar (the upper-right area of the Design Browser window), click the Send to Waveform Viewer icon.

   or

   Right-click the signals and click **Send to Waveform Window** (Figure 4–5).

   A waveform window showing all of your signals appears. You are now ready to simulate your testbench and design.

Figure 4–5. Selecting Signals in the Design Browser Window



Simulate the Design

After you have compiled and elaborated your design, you can simulate it using **ncsim**. The **ncsim** program loads the file or snapshot generated by **ncelab** as its primary input. It then loads other intermediate objects referenced by the snapshot. If you enable interactive debugging, it might also load HDL source files and script files. The simulation output is controlled by the model or debugger. The output can include result files generated by the model, SHM database, or **.vcd** file.

RTL Functional Simulation in Command-Line Mode

To perform RTL functional simulation of your Verilog HDL or VHDL design at the command line, type the following command:

```
ncsim [options][<library>.]<cell>[:<view>] ↵
```

For example:

```
ncsim worklib.lpm_ram_dp:syn ↵
```

Table 4–5 shows some of the options you can use with **ncsim**.

Table 4–5. ncsim Options

Options	Description
`-gui`	Launch GUI mode
`-batch`	Used for non-interactive mode
`-tcl`	Used for interactive mode (not required when using `–gui`)

RTL Functional Simulation in GUI Mode

You can run and step through simulation of your Verilog HDL or VHDL design in the GUI. In the Design Browser window, on the Simulation menu, click **Run** to begin the simulation.

You must load the design before simulating. If you have not done so, refer to step 1 in "Adding Signals in GUI Mode" on page 4–14 for instructions.

Post-Synthesis Simulation

The following sections provide detailed instructions for performing post-synthesis simulation using Quartus II output files, simulation libraries, and the Incisive platform software.

Quartus II Simulation Output Files

After performing synthesis with either a third-party synthesis tool or with Quartus II integrated synthesis, you must generate a simulation netlist for functional simulations. To generate a simulation netlist for functional simulation, perform the following steps in the Quartus II software:

1. Perform Analysis and Synthesis. On the Processing menu, point to **Start** and click **Start Analysis & Synthesis**.
2. Turn on the **Generate Netlist for Functional Simulation Only** option by performing the following steps:
   a. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
   b. In the **Category** list, click the "+" symbol to expand the **EDA Tool Settings** list.
   c. Under **EDA Tool Settings**, click **Simulation**.
   d. In the **Tool name** list, from the pull-down menu, select **NCSim**.
   e. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**.

      You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
   f. Click the **More EDA Netlist Writer Settings** button. The **More EDA Netlist Writer Settings** dialog box appears.
   g. In the **Existing options settings list**, click **Generate Netlist for Functional Simulation Only**.
   h. From the **Setting** list under **Option**, from the pull-down list, click **On**.
   i. Click **OK**.
   j. In the **Settings** dialog box, click **OK**.
3. Run the EDA Netlist Writer. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer**.

During the EDA Netlist Writer stage, the Quartus II software produces a **.vo** file or **.vho** file that can be used for post-synthesis simulations in the NC-Sim software. This netlist file is mapped to architecture-specific primitives. No timing information is included at this stage. The resulting netlist is located in the output directory you specified in the **Settings** dialog box, which defaults to the *<project directory>***/simulation/NCSim** directory.

Create Libraries

Create the following libraries for your simulation:

- Work library
- Device family library targeting your design targets using the following files in the *<path to Quartus II installation>***/eda/sim_lib** directory:
  - *<device_family>***_atoms.v**
  - *<device_family>***_atoms.vhd**
  - *<device_family>***_components.vhd**

Compile Project Files and Libraries

Compile the project files and libraries into your work directory using the **ncvlog** or **ncvhdl** programs or the GUI. Include the following files:

- Testbench file
- The Quartus II software functional output netlist file (**.vo** file or **.vho** file)
- Atom library file for the device family *<device family>***_atoms.***<v | vhd>*
- For VHDL, *<device family>***_components.vhd**

Refer to the section "Compile Source Code and Testbenches" on page 4–11 for instructions about compiling.

Elaborate Your Design

Elaborate your design using the **ncelab** program as described in "Elaboration in GUI Mode" on page 4–13.

Add Signals to the View

Refer to the section "Add Signals to View" on page 4–13 for information about adding signals to the view.

Simulate Your Design

Simulate your design using the **ncsim** program as described in "Simulate the Design" on page 4–16.

Gate-Level Timing Simulation

The following sections provide detailed instructions for performing timing simulation using the Quartus II output files, simulation libraries, and Cadence NC tools.

Generating a Gate-Level Timing Simulation Netlist

To perform gate-level timing simulation, the NC-Sim software requires information about how the design was placed into device-specific architectural blocks. The Quartus II software provides this information in the form of a **.vo** file for Verilog HDL designs and a **.vho** file for VHDL designs. The accompanying timing information is stored in the **.sdo** file, which annotates the delay for the elements found in the **.vo** file or **.vho** file.

To generate the **.vo** or **.vho** file and the **.sdo** file, perform the following steps:

1. Perform a full compilation. On the Processing menu, click **Start Compilation**.
2. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
3. In the **Category** list, select **Simulation**. The **Simulation** page appears.
4. In the **Tool name** list, select **NCSim**.
5. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
6. Click **OK**.
7. Run the EDA Netlist Writer. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer**.

During the EDA Netlist Writer stage, the Quartus II software produces a **.vo** file, **.vho** file, and an **.sdo** file used for gate-level timing simulations in the NC-Sim software. This netlist file is mapped to architecture-specific primitives. The timing information for the netlist is included in the **.sdo** file. The resulting netlist is located in the output directory you specified in the **Settings** dialog box, which defaults to the <*project directory*>**/simulation/ncsim** directory.

Generating a Timing Netlist with Different Timing Models

To generate the timing netlist in Stratix III and later devices (for example, Cyclone® III and Stratix IV), you can specify different temperature and voltage parameters. Using the Quartus II Classic or Quartus II TimeQuest Timing Analyzer when generating the simulation netlist files (***.vo** or ***.vho** and ***.sdo**), produces different timing models for different operating conditions. The multi-corner timing analysis is run by default during the full compilation.

Table 4–6 shows the example of default available operation conditions (model, voltage, and temperature) for Stratix III and Cyclone III devices.

Table 4–6. Default Available Operating Condition for Stratix III and Cyclone III Devices

Device Family	Model	Voltage	Temperature (°C)
Stratix III	Slow	1100 mV	85°
	Slow	1100 mV	0°
	Fast	1100 mV	0°
Cyclone III	Slow	1200 mV	85°
	Slow	1200 mV	0°
	Fast	1200 mV	0°

If the multi-corner timing analysis is not run during full compilation, you should perform the following steps to manually generate the simulation netlist files (***.vo** or *.**vho** and *.**sdo**) for the three different operating conditions listed in Table 4–6:

1. Generate all the available corner models at all operating conditions. Type the following command at a command prompt:

```
quartus_sta <project name> --multicorner ↵
```

2. Generate the simulation output files for all three corners specified in Table 4–6. Perform steps 2 through 7 in “Generating a Gate-Level Timing Simulation Netlist” on page 4–19. The output files are generated in the simulation output directory.

The following example shows all generated simulation netlist files for the three operating conditions of the preceding steps, when selecting **Verilog** as the output netlist format:

First slow corner (slow, 1100 mV, 85° C)

.vo file— <*revision name*>**.vo**

.sdo file— <*revision name*>**_v.sdo**

The <*revision_name*>**.vo** and <*revision name*>**_v.sdo** are generated for backward compatibility in case there are existing scripts that still use them.

.vo file— <*revision name*>_<*speedgrade*>**_1100mv_85c_slow.vo**

.sdo file— <*revision name*>_<*speedgrade*>**_1100mv_85c_v_slow.sdo**

Second slow corner (slow, 1100 mV, 0° C)

.vo file— <*revision name*>_<*speedgrade*>**_1100mv_0c_slow.vo**

.sdo file— <*revision name*>_<*speedgrade*>**_1100mv_0c_v_slow.sdo**

Fast corner (fast, 1100 mV, 0° C)

.vo file— <*revision name*>**_min_1100mv_0c_fast.vo**

.sdo file— <*revision name*>**_min_1100mv_0c_v_fast.sdo**

For all other older families, a slow-corner (worst case) timing model is used by default. There are only two timing models available: slow-corner and fast-corner. To generate the **.sdo** file using a different timing model, you must run the Quartus II Classic or the Quartus II TimeQuest Timing Analyzer with a different timing model before you start the EDA Netlist Writer.

To run the Quartus II Classic Timing Analyzer with the best-case model, on the Processing menu, point to **Start** and click **Start Classic Timing Analyzer (Fast Timing Model)**. After the timing analysis is complete, the Compilation Report appears. You can also type the following command at a command prompt:

```
quartus_tan <project_name> --fast_model=on ↵
```

To run the Quartus II TimeQuest Timing Analyzer with a best-case model, use the **-fast_model** option after you create the timing netlist. The following command enables the fast timing models:

```
create_timing_netlist --fast_model ↵
```

After you run the Quartus II Classic or Quartus II TimeQuest Timing Analyzer, you can perform steps 2 through 7 in "Generating a Gate-Level Timing Simulation Netlist" on page 4–19 to generate the ***.sdo** file. For fast corner timing models, the **-fast** post fix is added to the ***.vo** or ***.vho** and ***.sdo** files (for example, **my_project_fast.vo** or **my_project_fast.vho**, and **my_project_fast.sdo**).

For more information about generating a timing netlist with different timing models, refer to the *Quartus II Classic Timing Analyzer* or the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Disable Timing Violation on Registers

In certain situations, the timing violations can be ignored and you can disable the timing violation on registers. For example, timing violations that occur in internal synchronization registers used for asynchronous clock-domain crossing.

By default, the **x_on_violation_option logic** option is **On**, which means the simulation shows "**X**" whenever a timing violation occurs. To disable showing the timing violation on certain registers, you can set the **x_on_violation_option logic** option to **Off** for those registers. The following command is the Quartus II Tcl command to disable timing violation on registers. This Tcl command is also stored in the **.qsf** file.

```
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to \
<register_name>
```

Perform Timing Simulation Using Post-Synthesis Netlist

You can perform a timing simulation using the post-synthesis netlist instead of using a gate-level netlist and you can generate a **.sdo** file without running the Fitter. In this case, the **.sdo** file includes all timing values for the device cells only. Interconnect delays are not included because fitting (placement and routing) has not been performed.

To generate the post-synthesis netlist and the **.sdo** file, type the following at a command prompt:

```
quartus_map <project name> -c <revision name> ↵
quartus_tan <project name> -c <revision name> --post_map --zero_ic_delays ↵
quartus_eda <project name> -c <revision name> --simulation --tool=<3rd party EDA tool> \
--format=<HDL language> ↵
```

For more information about the `-format` and `-tool` options, type the following command at a command prompt:

```
quartus_eda -help=<options> command ↵
```

Quartus II Timing Simulation Libraries

Altera device simulation library files are provided in the <*Quartus II installation*>/**eda**/**sim_lib** directory. The **.vo** file or **.vho** file requires the library for the device your design targets. For example, the Stratix device family requires the following library files:

- **stratix_atoms.v**
- **stratix_atoms.vhd**
- **stratix_components.vhd**

If your design targets a Stratix device, you must set up the appropriate mappings in your **cds.lib** file. Refer to "Create Libraries" for more information.

Create Libraries

Create the following libraries for your simulation:

- Work library
- Device family libraries targeting using the following files in the <*path to Quartus II installation*>/**eda**/**sim_lib** directory:
  - <*device_family*>**_atoms.v**
  - <*device_family*>**_atoms.vhd**
  - <*device_family*>**_components.vhd**

For instructions for creating libraries, refer to "Basic Library Setup" on page 4–7 and "LPM Functions, Altera Megafunctions, and Altera Primitive Library Setup" on page 4–9.

Compile the Project Files and Libraries

Compile the project files and libraries into your work directory using the **ncvlog** or **ncvhdl** programs or the GUI. Include the following files:

- Testbench file
- The Quartus II software functional output netlist file (**.vo** file or **.vho** file)
- Atom library file for the device family <*device family*>**_atoms.**<*v* | *vhd*>
- For VHDL, <*device family*>**_components.vhd**

For instructions about compiling, refer to "Compile Source Code and Testbenches" on page 4–11.

Elaborate Your Design

When performing elaboration with the Quartus II-generated Verilog HDL netlist file, the Standard Delay Format Output File is read automatically. When you run **ncelab**, it recognizes the embedded system task `$sdf_annotate` and automatically compiles and annotates the Standard Delay Format Output File (runs **ncsdfc** automatically).

The Standard Delay Format Output File should be located in the same directory where you invoke an elaboration or simulation, because the `$sdf_annotate` task references the Standard Delay Format Output File without using a full path. If you are invoking an elaboration or simulation from a different directory, you can either comment out the `$sdf_annotate` and annotate the Standard Delay Format Output File with the GUI, or add the full path of the Standard Delay Format Output File.

Refer to "Elaborate Your Design" on page 4–12 for instructions for elaboration.

For VHDL, the Quartus II software-generated VHDL netlist file does not contain system task calls to locate your **.sdf** file; therefore, you must compile the Standard Delay Format Output File manually. For information about compiling the Standard Delay Format Output File, refer to "Compiling the Standard Delay Output File (VHDL Only) in Command-Line Mode" and "Compiling the Standard Delay Output File (VHDL Only) in GUI Mode".

Compiling the Standard Delay Output File (VHDL Only) in Command-Line Mode

To annotate the Standard Delay Format Output File timing data from the command line, perform the following steps:

1. Compile the Standard Delay Format Output File using the **ncsdfc** program by typing the following command at the command prompt:

   ```
   ncsdfc <project name>_vhd.sdo -output <output name> ↵
   ```

   The **ncsdfc** program generates an *<output name>***.sdf.X** compiled **.sdo** file.

   If you do not specify an output name, **ncsdfc** uses *<project name>***.sdo.X**.

2. Specify the compiled **.sdf** file for the project by adding the following command to an ASCII SDF command file for the project:

   ```
   COMPILED_SDF_FILE = "<project name>.sdf.X" SCOPE = <instance path>
   ```

Example 4–5 shows an example of an SDF command file.

Example 4–5. SDF Command File

```
// SDF command file sdf_file
COMPILED_SDF_FILE = "lpm_ram_dp_test_vhd.sdo.X",
SCOPE = :tb,
MTM_CONTROL = "TYPICAL",
SCALE_FACTORS = "1.0:1.0:1.0",
SCALE_TYPE = "FROM_MTM";
```

After you compile the **.sdf** file, type the following command to elaborate the design:

```
ncelab worklib.<project name>:entity -SDF_CMD_FILE <SDF Command File> ↵
```

Compiling the Standard Delay Output File (VHDL Only) in GUI Mode

To annotate the **.sdo** file timing data in the GUI, in the **NCLaunch** window, perform the following steps:

1. On the Tools menu, click **SDF Compiler**. The **Compile SDF** dialog box appears.
2. In the **SDF File** box, type the name of the **.sdo** file for the project.

3. Click **OK**.

   When the **.sdo** file compilation is complete, you can elaborate the design. Refer to "Elaboration in GUI Mode" on page 4–13 for instructions.

   If you perform a VHDL gate-level timing simulation, you must create an SDF command file before you begin elaboration. To create the SDF command file, continue the procedure.

4. On the Tools menu, click **Elaborator**. The **Elaborate** dialog box appears.
5. Click **Advanced Options**.
6. Click **Annotation**.
7. Turn on **Use SDF File**.
8. Click **Edit**.
9. Browse to the location of the SDF command file name.
10. Click **Add** and browse to the location of the **.sdo** file in the **Compiled SDF File** box and click **OK**.
11. Click **OK** to save and exit the **SDF Command File** dialog box.

Add Signals to View

Refer to the section "Add Signals to View" on page 4–13 for information about adding signals to view.

Simulate Your Design

Simulate your design using the **ncsim** program as described in "Simulate the Design" on page 4–16.

For the design examples to run gate-level timing simulation, refer to the Cadence NC-Sim Simulation Design Example web page.

Simulating Designs that Include Transceivers

If your design includes a Stratix IV, Stratix II GX, Stratix GX, Arria® II GX, or Arria GX transceiver, you must compile additional library files to perform RTL functional or gate-level timing simulations.

For high-speed simulation, you must select **ps** in the **Resolution** list for your simulator resolutions. If you choose slower than **ps**, the high-speed simulation might fail.

Stratix GX RTL Functional Simulation

To perform an RTL functional simulation of your design that instantiates the ALTGXB megafunction, enabling the gigabit transceiver block (GXB) on Stratix GX devices, compile the **stratixgx_mf** model file into the **altgxb** library.

The **stratixgx_mf** model file references the **lpm** and **sgate** libraries, so you must create these libraries to perform a simulation.

Compiling Library Files for Functional Stratix GX Simulation in VHDL

To compile the libraries necessary for functional simulation of a VHDL design targeting a Stratix GX device, type the commands shown in Example 4–6 at the NC-Sim command prompt.

Example 4–6. Compile Libraries Commands for Functional Simulation in VHDL

```
ncvhdl -work lpm 220pack.vhd 220model.vhd
ncvhdl -work altera_mf altera_mf_components.vhd altera_mf.vhd
ncvhdl -work sgate sgate_pack.vhd sgate.vhd
ncvhdl -work altgxb stratixgx_mf.vhd stratixgx_mf_components.vhd
ncsim work.<my design>
```

Compiling Library Files for Functional Stratix GX Simulation in Verilog HDL

To compile the libraries necessary for a functional simulation of a Verilog HDL design targeting a Stratix GX device, type the commands shown in Example 4–7 at the NC-Sim command prompt.

Example 4–7. Compile Libraries Commands for Functional Simulation in Verilog HDL

```
ncvlog -work lpm 220model.v
ncvlog -work altera_mf altera_mf.v
ncvlog -work sgate sgate.v
ncvlog -work altgxb stratixgx_mf.v
ncsim work.<my design>
```

Stratix GX Gate-Level Timing Simulation

To perform a gate-level timing simulation of your design that includes a Stratix GX transceiver, compile the **stratixgx_atoms** and **stratixgx_hssi_atoms** model files into the **stratixgx** and **stratixgx_gxb** libraries, respectively.

You must create these libraries to perform a simulation because the **stratixgx_hssi_atoms** model file references the **lpm** and **sgate** libraries.

Compiling Library Files for Timing Stratix GX Simulation in VHDL

To compile the libraries necessary for timing simulation of a VHDL design targeting a Stratix GX device, type the commands shown in Example 4–8 at the NC-Sim command prompt.

Example 4–8. Compile Libraries Commands for Timing Simulation in VHDL

```
ncvhdl -work lpm 220pack.vhd 220model.vhd
ncvhdl -work altera_mf altera_mf_components.vhd altera_mf.vhd
ncvhdl -work sgate sgate_pack.vhd sgate.vhd
ncvhdl -work stratixgx stratixgx_atoms.vhd stratixgx_components.vhd
ncvhdl -work stratixgx_gxb stratixgx_hssi_atoms.vhd \
stratixgx_hssi_components.vhd
ncelab work.<my design> -TIMESCALE 1ps/1ps -PULSE_R 0 -PULSE_INT_R 0
```

Compiling Library Files for Timing Stratix GX Simulation in Verilog HDL

To compile the libraries necessary for timing simulation of a Verilog HDL design targeting a Stratix GX device, type the commands shown in Example 4–9 at the NC-Sim command prompt.

Example 4–9. Compile Libraries Commands for Timing Simulation in Verilog HDL

```
ncvlog -work lpm 220model.v
ncvlog -work altera_mf altera_mf.v
ncvlog -work sgate sgate.v
ncvlog -work stratixgx stratixgx_atoms.v
ncvlog -work stratixgx_gxb stratixgx_hssi_atoms.v
ncelab work.<my design> -TIMESCALE 1ps/1ps -PULSE_R 0 -PULSE_INT_R 0
```

Stratix II GX RTL Functional Simulation

Stratix II GX functional simulation is similar to Arria GX functional simulation. Example 4–10 shows only the RTL functional simulation for designs that include transceivers in Stratix II GX. To simulate transceivers in Arria GX, replace the **stratixiigx_hssi** model file with the **arriagx_hssi** model file.

To perform an RTL functional simulation of your design that instantiates the ALT2GXB megafunction, edit your **cds.lib** file so all of the libraries point to the work library, and compile the **stratixiigx_hssi** model file into the **stratixiigx_hssi** library. When compiling the library files, you can safely ignore the following warning message:

```
"Multiple logical libraries mapped to a single location"
```

Example 4–10 shows the **cds.lib** file.

Example 4–10. cds.lib File

```
SOFTINCLUDE ${CDS_INST_DIR}/tools/inca/files/cdsvhdl.lib
SOFTINCLUDE ${CDS_INST_DIR}/tools/inca/files/cdsvlog.lib
DEFINE work ./ncsim_work
DEFINE stratixiigx_hssi ./ncsim_work
DEFINE stratixiigx ./ncsim_work
DEFINE lpm ./ncsim_work
DEFINE sgate ./ncsim_work
```

The **stratixiigx_hssi_atoms** model file references the **lpm** and **sgate** libraries, so you must create these libraries to perform a simulation.

Generate a functional simulation netlist by turning on **Create a simulation library for this design** in the last page of the ALT2GXB MegaWizard Plug-In Manager (Figure 4–6). The *<alt2gxb entity name>***.vho** or *<alt2gxb module name>***.vo** is generated in the current project directory.

Figure 4–6. ALT2GXB MegaWizard Plug-In Manager: Generate Simulation Model

The Quartus II generated ALT2GXB functional simulation library file references **stratixiigx_hssi** WYSIWYG atoms.

Compiling Library Files for Functional Stratix II GX Simulation in VHDL

To compile the libraries necessary for functional simulation of a VHDL design targeting a Stratix II GX device, type the commands shown in Example 4–11 at the NC-Sim command prompt.

Example 4–11. Compile Libraries Commands for Functional Simulation in VHDL

```
ncvhdl -work lpm 220pack.vhd 220model.vhd
ncvhdl -work altera_mf altera_mf_components.vhd altera_mf.vhd
ncvhdl -work sgate sgate_pack.vhd sgate.vhd
ncvhdl -work stratixiigx_hssi stratixiigx_hssi_components.vhd \
stratixiigx_hssi_atoms.vhd
ncvhdl -work work <alt2gxb entity name>.vho
ncelab work.<my design>
```

Compiling Library Files for Functional Stratix II GX Simulation in Verilog HDL

To compile the libraries necessary for functional simulation of a Verilog HDL design targeting a Stratix II GX device, type the commands shown in Example 4–12 at the NC-Sim command prompt.

Example 4–12. Compile Libraries Commands for Functional Simulation in Verilog HDL

```
ncvlog -work lpm 220model.v
ncvlog -work altera_mf altera_mf.v
ncvlog -work sgate sgate.v
ncvlog -work stratixiigx_hssi stratixiigx_hssi_atoms.v
ncvlog -work work <alt2gxb module name>.vo
ncelab work.<my design>
```

Stratix II GX Gate-Level Timing Simulation

Stratix II GX functional simulation is similar to Arria GX functional simulation. Example 4–13 shows only the gate-level timing simulation for designs that include transceivers in Stratix II GX. To simulate transceivers in Arria GX, replace the **stratixiigx_hssi** model file with the **arriagx_hssi** model file.

To perform a post-fit timing simulation of your design that includes the ALT2GXB megafunction, edit your **cds.lib** file so that all the libraries point to the work library and compile `stratixiigx_atoms` and `stratixiigx_hssi_atoms` into the **stratixiigx** and **stratixiigx_hssi** libraries, respectively. When compiling the library files, you can safely ignore the following warning message:

```
"Multiple logical libraries mapped to a single location"
```

For an example of a **cds.lib** file, refer to "Stratix II GX RTL Functional Simulation" on page 4–26.

The **stratixiigx_hssi_atoms** model file references the **lpm** and **sgate** libraries, so you must create these libraries to perform a simulation.

Compiling Library Files for Timing Stratix II GX Simulation in VHDL

To compile the libraries necessary for timing simulation of a VHDL design targeting a Stratix II GX device, type the commands shown in Example 4–13 at the NC-Sim command prompt.

Example 4–13. Compile Libraries Commands for Timing Simulation in VHDL

```
ncvhdl -work lpm 220pack.vhd 220model.vhd
ncvhdl -work altera_mf altera_mf_components.vhd altera_mf.vhd
ncvhdl -work sgate sgate_pack.vhd sgate.vhd
ncvhdl -work stratixiigx stratixiigx_atoms.vhd \
stratixiigx_components.vhd
ncvhdl -work stratixiigx_hssi stratixiigx_hssi_components.vhd \
stratixiigx_hssi_atoms.vhd
ncvhdl -work work <alt2gxb>.vho
ncelab work.<my design> -TIMESCALE 1ps/1ps -PULSE_R 0 -PULSE_INT_R 0
```

Compiling Library Files for Timing Stratix II GX Simulation in Verilog HDL

To compile the libraries necessary for timing simulation of a Verilog HDL design targeting a Stratix II GX device, type the commands shown in Example 4–14 at the NC-Sim command prompt.

Example 4–14. Compile Libraries Commands for Timing Simulation in Verilog HDL

```
ncvlog -work lpm 220model.v
ncvlog -work altera_mf altera_mf.v
ncvlog -work sgate sgate.v
ncvlog -work stratixiigx stratixiigx_atoms.v
ncvlog -work stratixiigx_hssi stratixiigx_hssi_atoms.v
ncelab work.<my design> -TIMESCALE 1ps/1ps -PULSE_R 0 -PULSE_INT_R 0
```

Pulse Reject Delays

By default, the NC-Sim software filters out all pulses that are shorter than the propagation delay between primitives. Setting the pulse reject delays (similar to transport delays) options in the NC-Sim software prevents the simulation tool from filtering out these pulses. Use the following options to ensure that all signal pulses are seen in the simulation results.

-PULSE_R

Use this option when the pulses in your simulation are shorter than the delay within a gate-level primitive. The argument is the percentage of delay for pulse reject limit for the path.

-PULSE_INT_R

Use this option when the pulses in your simulation are shorter than the interconnect delay between gate-level primitives. The argument is the percentage of delay for pulse reject limit for the path. The `-PULSE_R` and `-PULSE_INT_R` options are also used by default in the NativeLink feature for gate-level timing simulation.

The following NC-Sim software command describes the command-line syntax to perform a gate-level timing simulation with the device family library:

```
ncelab worklib.<project name>:entity -SDF_CMD_FILE <SDF Command File> \
-TIMESCALE 1ps/1ps -PULSE_R 0 -PULSE_INT_R 0
```

Using the NativeLink Feature with NC-Sim

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools and allows you to run NC-Sim within the Quartus II software.

Setting Up NativeLink

To run NC-Sim automatically from the Quartus II software using the NativeLink feature, you must specify the path to your simulation tool by performing the following steps:

1. On the Tools menu, click **Options**. The **Options** dialog box appears.
2. In the **Category** list, select **EDA Tool Options**. The **EDA Tool Options** page appears.

3. Double-click the entry under the **Location of executable** column beside the name of your **EDA Tool**, and type or browse to the directory containing the executables of your EDA tool.

   For example, if NCSim is installed in the **/tools/ncsim** directory, enter the location as **/tools/ncsim/tools.lnx86/bin**.

4. Click **OK**.

You can also specify the path to the simulator's executables by using the `set_user_option` Tcl command:

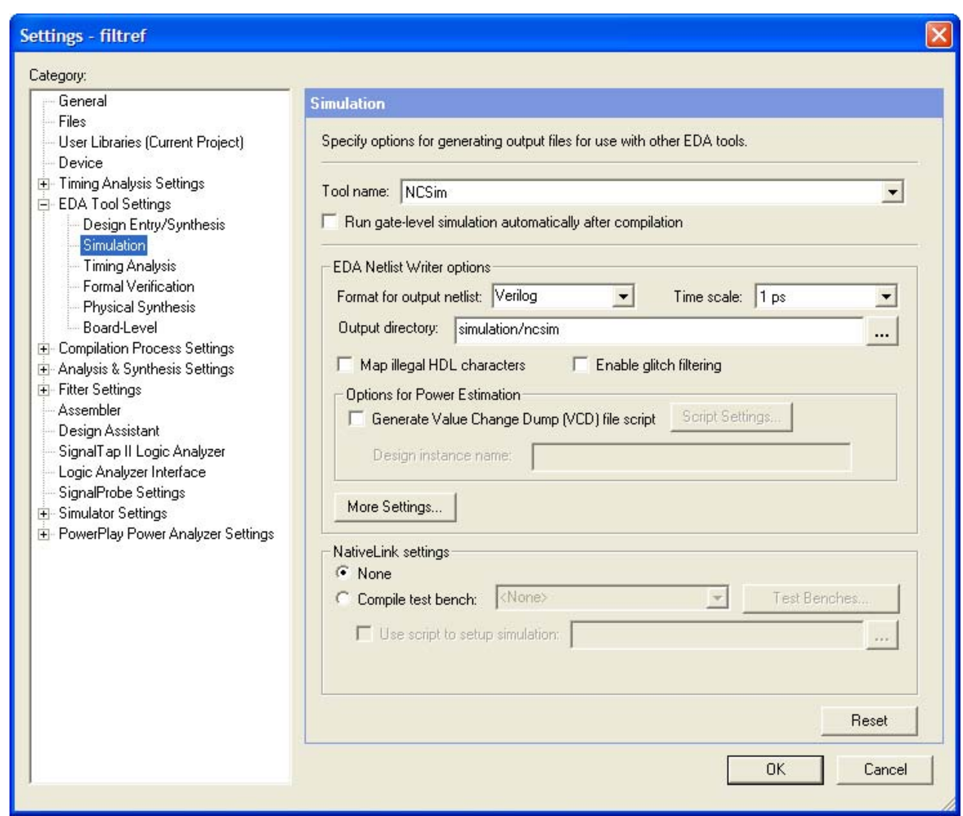
```
set_user_option -name EDA_TOOL_PATH_NCSIM <path to executables>
```

Performing an RTL Simulation Using NativeLink

To run an RTL functional simulation with the NC-Sim software automatically in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears (Figure 4–7).

Figure 4–7. Simulation Page in the Settings Dialog Box



3. In the **Tool name** pull-down list, select **NCSim**.

4. If your design is written entirely in Verilog HDL or in VHDL, the NativeLink feature automatically chooses the correct language and Altera simulation libraries. If your design is written with mixed languages, the NativeLink feature uses the default language specified in the **Format for output netlist** list. To change the default language when there is a mixed language design, under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. Table 4–7 shows the design languages for output netlists and simulation models.

Table 4–7. NativeLink Design Languages

Design File	Format for Output Netlist	Simulation Models Used
Verilog HDL	Any	Verilog HDL
VHDL	Any	VHDL
Mixed	Verilog HDL	Verilog HDL
Mixed	VHDL	VHDL

For mixed-language simulation, choose the same language that was used to generate the megafunctions to ensure correct parameter passing between the megafunctions and the Altera libraries. For example, if your ALTSYNCRAM megafunction was generated in VHDL, choose VHDL as the format for output netlist.

For mixed-language simulations, it is important to be aware of the following conditions:

- VHDL designs instantiating Verilog HDL user-defined primitives (UDPs) are not supported.
- Parameters cannot be passed in Verilog HDL modules that instantiate VHDL components.

5. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.

   For more information about setting up a testbench with NativeLink, refer to the section “Setting Up a Testbench” on page 4–33.

6. If you have compiled libraries using the EDA Simulation Library Compiler, perform the following steps:

   a. On the **Simulation** page, click **More EDA Netlist Writer Settings**. The **More EDA Netlist Writer Settings** dialog box appears.

   b. Under **Existing option settings**, click **Location of User-Compiled Simulation Library**.

   c. Under **Setting**, type the path that contains the user-compiled libraries that are generated from the EDA Simulation Library Compiler; for example, **c:**<*design_path*>**/simulation/ncsim**.

7. Click **OK.**

8. On the Processing menu, point to **Start** and click **Start Analysis and Elaboration** to perform an analysis and elaboration. This command collects all your file name information and builds your design hierarchy in preparation for simulation.

9. To automatically run NC-Sim, compile all necessary design files, and complete a simulation, on the Tools menu, point to **Run EDA Simulation Tool** and click **EDA RTL Simulation**.

Performing a Gate-Level Simulation Using NativeLink

To run a gate-level timing simulation with the NC-Sim software in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears (Figure 4–7 on page 4–30).
3. In the **Tool name** list, select **NCSim**.
4. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, choose **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
5. To perform a gate level simulation after each full compilation, turn on **Run Gate Level Simulation automatically after compilation**.
6. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.
7. If you have compiled libraries using the EDA Simulation Library Compiler, perform the following steps:
   a. On the **Simulation** page, click **More EDA Netlist Writer Settings**. The **More EDA Netlist Writer Settings** dialog box appears.
   b. Under **Existing option settings**, click **Location of User-Compiled Simulation Library**.
   c. Under **Setting**, type the path that contains the pre-compiled libraries that are generated from the EDA Simulation Library Compiler; for example, **c:**<*design_path*>**/simulation/modelsim**.
8. Click **OK**.
9. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer** to generate a simulation netlist of your design.

☞ If you have run full compilation after you set the **EDA Tool Settings**, the **Start EDA Netlist Writer** step is not required.

10. To automatically run NC-Sim, compile all necessary design files, and complete a simulation, on the Tools menu, point to **Run EDA Simulation Tool** and click **EDA Gate Level Simulation**.

☞ If multi-corner Timing Analysis is performed, a dialog box appears asking you to select the timing corner to run simulation. Select the timing corner and click **Run**.

A Tcl File (***.tcl**) is generated in the <*project_directory*>**\simulation\ncsim** directory when you run NativeLink. This **.tcl** file enables you to simulate the design without using NativeLink, with the following command:

```
quartus_sh -t <project_directory>\simulation\ncsim\<generated_do_file>.tcl ↵
```

1. To generate only the Tcl script without using the GUI after the NativeLink process, perform the following steps:
   a. On the **Simulation** page, click **More NativeLink Settings**. The **More NativeLink Settings** dialog box appears.
   b. Under **Existing option settings**, click **Generate third-party EDA tool command scripts without running the EDA tool**.
   c. Under **Setting**, click **On**.

   If you turn this option on and run NativeLink, the **.tcl** file for the simulation process will be generated without showing the results in the GUI. You can then run the simulation by typing the following command:

```
quartus_sh -t <your_design_name>_ncsim_gate_level_<verilog/vhdl>.tcl
```

Setting Up a Testbench

You can compile your design files and testbench files, and run EDA simulation tools to perform a simulation automatically using the NativeLink feature.

To setup NativeLink with a testbench, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears.
3. Under **NativeLink settings**, select **None** or **Compile test bench** (Table 4–8).

Table 4–8. NativeLink Settings

Settings	Description
None	Compile simulation models and design files.
Compile test bench	NativeLink compiles simulation models, design files, testbench files, and starts simulation.

4. If you select **Compile test bench**, select your testbench setup from the **Compile test bench** list. You can use different testbench setups to specify different testbench files for different test scenarios. If there are no testbench setups entered, create a testbench setup by performing the following steps:

   a. Click **Test Benches**. The **Test Benches** dialog box appears.

   b. Click **New**. The **New Test Bench Settings** dialog box appears.

   c. In the **Test Bench name** box, type in the testbench setup name which is used to identify the different testbench setups.

   d. In the **Test level module** box, type in the top-level entity name. For example, for a Quartus II generated VHDL testbench, type `filtref_vhd_vec_tst`.

   e. In the **Instance** box, type in the full instance path to the top level of your FPGA design. For example, for a Quartus II generated VHDL testbench, type `i1`.

   f. Under **Simulation period**, select **Run simulation until all vector stimuli are used**. If you select **End simulation at**, specify the simulation end time and the time unit.

   g. Under **Test bench files**, browse and add all your testbench files in the **File name** box. Use the **Up** and **Down** button to reorder your files. The script used by NativeLink compiles the files in order from top to the bottom.

   You can also specify the library name and the HDL version to compile the testbench file. NativeLink compiles the testbench to the library name of the HDL specified version.

   h. Click **OK**.

   i. In the **Test Benches** dialog box, click **OK**.

5. Under **NativeLink settings**, you can turn on **Use script to setup simulation** and browse to your script. You can write a script to set up your waveforms before running the simulation.

The script should be a valid NC-Sim Tcl script. NativeLink passes the script to **ncsim** command with command-line arguments to set up and run simulation.

Creating a Testbench

In the Quartus II software, you can create a Verilog HDL or VHDL testbench from a Vector Waveform File (**.vwf**). The generated testbench includes the behavior of the input stimulus and applies it to your instantiated top-level FPGA design.

1. On the File menu, click **Open**. The **Open** dialog box appears.
2. In the Files of type list, **Files of type** arrow and select **Waveform/Vector Files**. Select your file.
3. Click **Open**.
4. On the File menu, click **Export**. The **Export** dialog box appears.
5. In the **Save as type** list, select **VHDL Test Bench File (*.vht)** or **Verilog Test Bench File (*.vt)**.

6. You can turn on **Add self-checking code to file** to check your simulation results against your Vector Waveform File.
7. Click **Export**.

Generate Simulation Script from EDA Netlist Writer

In the Quartus II software version 9.0 and later, you can generate the simulation script (including the **.do** file, Tcl script, and option file) when you run the EDA Netlist Writer. You can use the simulation script to run your simulation in the preferred simulator in stand-alone mode.

To generate a simulation script with the EDA Netlist Writer, perform the following steps:

1. On the **Simulation** page, click **More EDA Netlist Writer Settings**. The **More EDA Netlist Writer Settings** dialog box appears.
2. Under **Existing option settings**, click **Generate third-party EDA tool command script for RTL simulation** (if you want to generate an RTL simulation script) or click **Generate third-party EDA tool command script for Gate-Level simulation** (if you want to generate a Gate-Level simulation script).
3. Under **Setting**, click **On**.
4. Click **OK**.
5. Follow the steps in "Quartus II Simulation Output Files" on page 4–17.

In the command-line, to generate the simulation script with the EDA Netlist Writer, type the following command:

```
quartus_eda --simulation --tools=<Your decided tool> --format=<vhdl or verilog>
    --gen_script=<rtl or gate_level> <Project Name> -c <Revision name> ↵
```

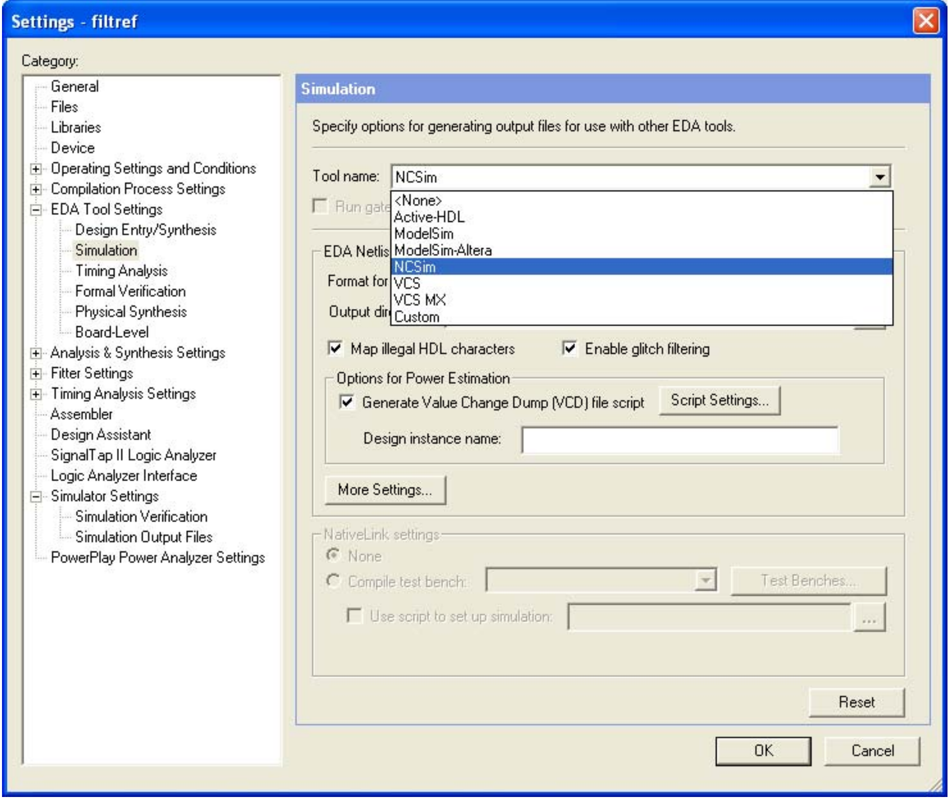
Generating a Timing VCD File for PowerPlay

To generate a timing **.vcd** file for PowerPlay, you must first generate a VCD script in the Quartus II software and run the VCD script from the NC-Sim software to generate a timing **.vcd** file. This timing **.vcd** file can then be used by the PowerPlay Early Power Estimator for power estimation. The following instructions show you how to generate a timing **.vcd** file.

Perform the following steps to generate timing VCD scripts in the Quartus II software:

1. In the Quartus II software, on the Assignments menu, click **Settings**. The **Settings** dialog box appears (Figure 4–8).
2. In the **Category** list, click the "+" icon to expand **EDA Tool SEttings**.
3. Click **Simulation**.
4. From the **Tool Name** pull-down list, click your required third-party simulation tool.
5. Turn on the **Generate Value Change Dump (VCD) File Script** option.

Figure 4–8. Simulation Settings Dialog Box



6. Click **OK**.
7. To generate the VCD script file, perform a full compilation.

Perform the following steps to generate a timing **.vcd** file in NC-Sim:

1. In the NC-Sim software, before simulating your design, source the *<revision_name>***_dump_all_vcd_nodes.tcl** script. To source the **.tcl** script, you should use the **–input** switch while running the **nssim** command. For example:

```
ncsim -input <revision_name>_dump_all_vcd_nodes.tcl <my design>
```

2. Continue to run the simulation until simulation finishes. Exit the **ncsim** and the *<revision_name>***.vcd** is generated in the simulation directory.

For more detailed information about using the timing **.vcd** file for power estimation, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Viewing a Waveform from a .trn File

A **.trn** file is automatically generated when your simulation is done. The **.trn** file is not readable. It is used for generating the waveform view through SimVision.

To view a waveform from a **.trn** file through SimVision, perform the following steps:

1. Type `simvision` on a command line. The **Design Browser** dialog box appears.
2. On the File menu, click **Open Database**. The **Open File** dialog box appears.

3. In the **Directories** field, browse to the directory that contains your **.trn** file.
4. Double-click the **.trn** file.
5. In the **Design Browser** dialog box, select the signals that you want to observe from the Hierarchy.
6. Right-click the selected signals and click **Send to Waveform Window**.

You cannot view a waveform from a **.vcd** file in SimVision and the **.vcd** file cannot be converted to a **.trn** file.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt.

For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser.

To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF format.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For information about all settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Manual*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Generate NC-Sim Simulation Output Files

You can generate **.vo** files and **.sdo** simulation output files with Tcl commands or at a command prompt.

For more information about generating **.vo** simulation output files and **.sdo** file simulation output files, refer to "Quartus II Simulation Output Files" on page 4–17.

Tcl Commands

The following three assignments cause a Verilog HDL netlist to be written out when you run the Quartus II netlist writer. The netlist has a 1 ps timing resolution for the NC-Sim simulation software.

```
set_global_assignment -name EDA_OUTPUT_DATA_FORMAT VERILOG -section_id eda_simulation
set_global_assignment -name EDA_TIME_SCALE "1 ps" -section_id eda_simulation
set_global_assignment -name EDA_SIMULATION_TOOL "NC-Verilog (Verilog)"
```

Use the following Tcl command to run the Quartus II Netlist Writer:

```
execute_module -tool eda
```

Command Prompt

Use the following command to generate a simulation output file for the Cadence NC-Sim software simulator. Specify Verilog HDL or VHDL for the format.

```
quartus_eda <project name> --simulation --format=<verilog|vhdl> --tool=ncsim ↵
```

Conclusion

The Cadence NC family of simulators work within an Altera FPGA design flow to perform RTL functional, post-synthesis, and gate-level timing simulation, easily and accurately.

Altera provides functional models of LPM and Altera-specific megafunctions that you can compile with your testbench or design. For timing simulation, use the atom netlist file generated by Quartus II compilation.

The seamless integration of the Quartus II software and Cadence NC tools make this simulation flow an ideal method for fully verifying an FPGA design.

Referenced Documents

This chapter references the following documents:

- Cadence NC-Sim Simulation Design Example
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Manual*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *SDF Annotate Guide* and *Cadence NC-Sim User Manual* from Cadence
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 4–9 shows the revision history for this chapter.

Table 4–9. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Removed "Compile Libraries Using the Altera Simulation Library Compiler". ■ Added "Compile Libraries Using the EDA Simulation Library Compiler" on page 4–5. ■ Added "Generate Simulation Script from EDA Netlist Writer" on page 4–35. ■ Added "Viewing a Waveform from a .trn File" on page 4–36. ■ Minor editorial updates.	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	■ Added "Compile Libraries Using the Altera Simulation Library Compiler" on page 4–5. ■ Added information about the `--simlib_comp` utility. ■ Minor editorial updates. ■ Updated entire chapter using 8½" × 11" chapter template.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0.0	■ Updated Table 4–1. ■ Updated Figure 4–1. ■ Updated "Compilation in Command-Line Mode" on page 4–9. ■ Updated "Generating a Timing Netlist with Different Timing Models" on page 4–18. ■ Added "Disable Timing Violation on Registers" on page 4–20. ■ Updated "Simulating Designs that Include Transceivers" on page 4–23. ■ Updated "Performing a Gate Level Simulation Using NativeLink" on page 4–30. ■ Added "Generating a Timing VCD File for PowerPlay" on page 4–33. ■ Added hyperlinks to referenced documents throughout the chapter. ■ Minor editorial updates.	Updated for the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

5. Aldec Active-HDL Support

QII53023-9.0.0

Introduction

This chapter provides an overview of how to use the Active-HDL software to simulate designs that target Altera® FPGAs. It provides a step-by-step explanation of how to perform register transfer level (RTL) functional simulations, post-synthesis simulations, and gate-level timing simulations for Verilog HDL or VHDL designs using Active-HDL software. This chapter also describes the location of the simulation library files, how to compile these library files, and how to load the design and testbench.

The Quartus® II software version 9.0 and later supports Aldec's Riviera-PRO tool. For more information about this simulator, refer to the manual that is provided with installation. The manual is located at <*Riviera-PRO Installation Directory*>**/help/riviera.pdf**.

This chapter contains the following topics:

- "Software Compatibility"
- "Using Active-HDL Software in Quartus II Design Flows" on page 5–2
- "Simulation Libraries" on page 5–3
- "Simulation Netlist Files" on page 5–6
- "Perform Simulation Using the Active-HDL Software (GUI Mode)" on page 5–12
- "Perform Simulation Using the Active-HDL Software (Batch Mode)" on page 5–28
- "Simulating Designs that Include Transceivers" on page 5–39
- "Using the NativeLink Feature in Active-HDL Software" on page 5–45
- "Generating VCD Files for PowerPlay" on page 5–51
- "Scripting Support" on page 5–52

Software Compatibility

To simulate your design using the Active-HDL software, you must first set up the Altera libraries. These libraries are installed with the Quartus II software. Table 5–1 shows the compatibility between versions of the Quartus II software and the Aldec Active-HDL software.

Table 5–1. Supported Quartus II and Active-HDL Software Version Compatibility

Aldec	Altera
Active-HDL software version 8.1	Quartus II software version 9.0
Active-HDL software version 7.3sp1	Quartus II software version 8.1
Active-HDL software version 7.3	Quartus II software version 8.0
Active-HDL software version 7.3	Quartus II software version 7.2

For more information about installing the software and directories created during the Quartus II software installation, refer to the *Quartus II Installation and Licensing for Windows* manual or the *Quartus II Installation and Licensing for UNIX and Linux Workstations* manual.

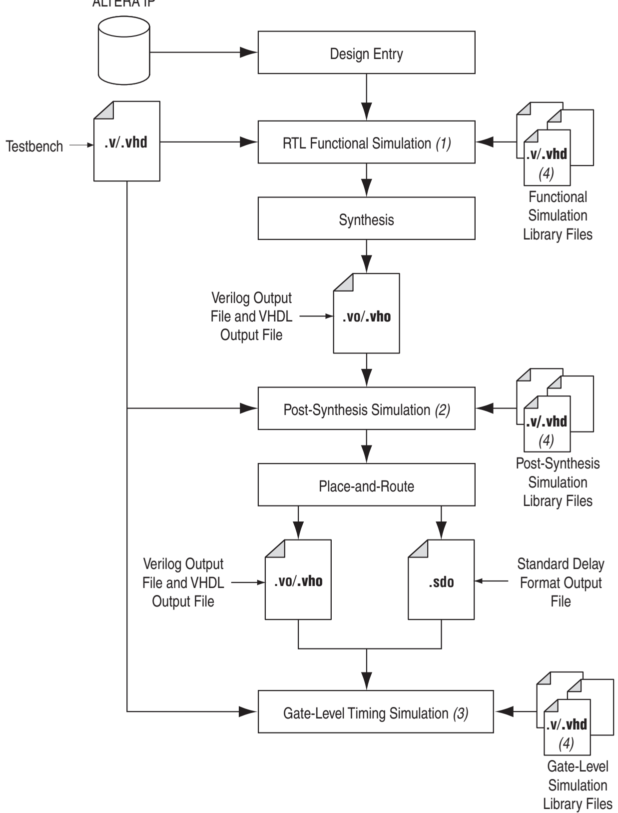
Using Active-HDL Software in Quartus II Design Flows

You can perform the following types of simulations using the Active-HDL software:

- RTL Functional Simulation
- Post-synthesis Simulation
- Gate-level Timing Simulation

Figure 5–1 illustrates the Active-HDL software used in the Altera Quartus II design flow.

Figure 5–1. Using the Active-HDL Software in an Altera Quartus II Design



Notes to Figure 5–1:

(1) RTL functional simulation is performed before a gate-level timing simulation or post-synthesis simulation. RTL functional simulation verifies the functionality of the design before synthesis and place-and-route. If you are performing an RTL functional simulation through NativeLink, you must complete analysis and elaboration first.

(2) A post-synthesis simulation verifies the functionality of a design after synthesis has been performed.

(3) Gate-level timing simulation is a post place-and-route simulation to verify the operation of the design after the worst-case timing delays have been calculated.

(4) In Active-HDL software, you must compile the simulation library files to perform RTL functional, post-synthesis, and gate-level timing simulation. All these simulation libraries are located at *<quartus installation directory>***eda\sim_lib**. If you already have the precompiled simulation libraries that come with the Active-HDL software, compiling the simulation library files is not required.

Simulation Libraries

Simulation model libraries are required to run a simulation whether you are running an RTL functional simulation, post-synthesis simulation, or gate-level timing simulation. In general, running an RTL functional simulation requires the RTL functional simulation model libraries and running a post-synthesis or gate-level timing simulation requires the gate-level timing simulation model libraries. You must compile the necessary library files before you can run the simulation.

However, there are a few exceptions where you are also required to compile gate-level timing simulation library files to perform RTL functional simulation. For example, the following are some of the Altera megafunctions gate-level libraries required to perform an RTL functional simulation in third-party simulators:

- ALTCLKBUF
- ALTCLKCTRL
- ALTDQS
- ALTDQ
- ALTDDIO_IN
- ALTDDIO_OUT
- ALTDDIO_BIDIR
- ALTUFM_NONE
- ALTUFM_PARALLEL
- ALTUFM_SPI
- ALTMEMMULT
- ALTREMOTE_UPDATE

To identify what type of simulation libraries are required to run the simulation for a certain Altera megafunction, refer to the last page in the Altera MegaWizard™ Plug-In Manager. It tells you what simulation library files are required to perform an RTL functional simulation for that particular megafunction.

Simulating the transceiver megafunction (for example, ALT2GXB) is also another exception that requires the gate-level libraries to perform RTL functional simulation and vice-versa. For detailed, step-by-step instructions about how to simulate the transceiver megafunction, refer to "Simulating Designs that Include Transceivers" on page 5–39.

Simulation Library Files in the Quartus II Software

In Active-HDL software, you must compile the necessary libraries to perform RTL functional, post-synthesis functional, or gate-level timing simulation. The following sections show the location of these library files in the Quartus II directory structure. You can refer to these library files for any particular simulation model that you are looking for.

RTL Functional Simulation Library Files

Table 5–2 shows the VHDL RTL simulation model location in the Quartus II directory.

Table 5–2. RTL Functional Simulation Library Files in the Quartus II Software (VHDL)

RTL Simulation Model	Location in Quartus II Directory Structure
LPM	*<Quartus II installation directory>***\eda\sim_lib\220pack.vhd** *<Quartus II installation directory>***\eda\sim_lib\220model.vhd** *<Quartus II installation directory>***\eda\sim_lib\220model_87.vhd** *(1)*
Altera Megafunction	*<Quartus II installation directory>***\eda\sim_lib\altera_mf_components.vhd** *<Quartus II installation directory>***\eda\sim_lib\altera_mf.vhd** *<Quartus II installation directory>***\eda\sim_lib\altera_mf_87.vhd** *(1)*
Low-Level Primitives	*<Quartus II installation directory>* **\eda\sim_lib\altera_primitives_components.vhd** *<Quartus II installation directory>***\eda\sim_lib\altera_primitives.vhd**
ALTGXB Megafunction (Stratix® GX)	*<Quartus II installation directory>* **\eda\sim_lib\stratixgx_mf_components.vhd** *<Quartus II installation directory>***\eda\sim_lib\stratixgx_mf.vhd**

Note to Table 5–2:

(1) Simulating a design that uses VHDL-1987.

Table 5–3 shows the Verilog RTL simulation model location in the Quartus II directory.

Table 5–3. RTL Functional Simulation Library Files in the Quartus II Software (Verilog)

RTL Simulation Model	Location in Quartus II Directory Structure
LPM	*<Quartus II installation directory>***\eda\sim_lib\220model.v**
Altera Megafunction	*<Quartus II installation directory>***\eda\sim_lib\altera_mf.v**
Low-Level Primitives	*<Quartus II installation directory>***\eda\sim_lib\altera_primitives.v**
ALTGXB Megafunction (Stratix GX)	*<Quartus II installation directory>***\eda\sim_lib\stratixgx_mf.v**

Gate-Level Timing Simulation Library Files

Table 5–4 shows the VHDL gate-level timing simulation model location in the Quartus II directory.

Table 5–4. Gate-Level Timing Simulation Library Files in the Quartus II Software (VHDL) (Part 1 of 2)

Device Simulation Model	Location in Quartus II Directory Structure
Arria® GX (without transceiver block)	*<Quartus II installation directory>***\eda\sim_lib\arriagx_components.vhd** *<Quartus II installation directory>***\eda\sim_lib\arriagx_atoms.vhd**
Arria GX (with transceiver block)	*<Quartus II installation directory>***\eda\sim_lib\arriagx_hssi_components.vhd** *<Quartus II installation directory>***\eda\sim_lib\arriagx_hssi_atoms.vhd**
Stratix IV	*<Quartus II installation directory>***\eda\sim_lib\stratixiv_components.vhd** *<Quartus II installation directory>***\eda\sim_lib\stratixiv_atoms.vhd**

© March 2009 Altera Corporation

Table 5–4. Gate-Level Timing Simulation Library Files in the Quartus II Software (VHDL) (Part 2 of 2)

Device Simulation Model	Location in Quartus II Directory Structure
Stratix IV (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiv_hssi_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixiv_hssi_atoms.vhd**
Stratix IV (with PCI Express)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiv_pcie_hip_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixiv_pcie_hip_atoms.vhd**
Stratix III	<*Quartus II installation directory*>**\eda\sim_lib\stratixiii_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixiii_atoms.vhd**
Stratix II	<*Quartus II installation directory*>**\eda\sim_lib\stratixii_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixii_atoms.vhd**
Stratix II GX (without transceiver blocK)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiigx_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixiigx_atoms.vhd**
Stratix II GX (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiigx_hssi_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixiigx_hssi_atoms.vhd**
Stratix	<*Quartus II installation directory*>**\eda\sim_lib\stratix_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratix_atoms.vhd**
Stratix GX	<*Quartus II installation directory*>**\eda\sim_lib\stratixgx_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixgx_atoms.vhd**
Stratix GX (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixgx_hssi_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\stratixgx_hssi_atoms.vhd**
Cyclone® II	<*Quartus II installation directory*>**\eda\sim_lib\cycloneii_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\cycloneii_atoms.vhd**
Cyclone	<*Quartus II installation directory*>**\eda\sim_lib\cyclone.vhd** <*Quartus II installation directory*>**\eda\sim_lib\cyclone_atoms.vhd**
MAX® II	<*Quartus II installation directory*>**\eda\sim_lib\maxii_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\maxii_atoms.vhd**
MAX 7000 and MAX 3000	<*Quartus II installation directory*>**\eda\sim_lib\max_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\max_atoms.vhd**
APEX™ II	<*Quartus II installation directory*>**\eda\sim_lib\apexii_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\apexii_atoms.vhd**
APEX 20K	<*Quartus II installation directory*>**\eda\sim_lib\apex20k_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\apex20k_atoms.vhd**
APEX 20KC, APEX 20KE, and Excalibur™	<*Quartus II installation directory*>**\eda\sim_lib\apex20ke_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\apex20ke_atoms.vhd**
FLEX® 10KE and ACEX® 1K	<*Quartus II installation directory*>**\eda\sim_lib\flex10ke_components.vhd** <*Quartus II installation directory*>**\eda\sim_lib\flex10ke_atoms.vhd**

Table 5–5 shows the Verilog gate-level timing simulation model location in the Quartus II directory.

Table 5–5. Gate-Level Timing Simulation Library Files in the Quartus II Software (Verilog)

Device Simulation Model	Location in Quartus II Directory Structure
Arria GX (without transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\arriagx_atoms.v**
Arria GX (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\arriagx_hssi_atoms.v**
Stratix IV	<*Quartus II installation directory*>**\eda\sim_lib\stratixiv_atoms.v**
Stratix IV (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiv_hssi_atoms.v**
Stratix IV (with PCI Express)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiv_pcie_hip_atoms.v**
Stratix III	<*Quartus II installation directory*>**\eda\sim_lib\stratixiii_atoms.v**
Stratix II	<*Quartus II installation directory*>**\eda\sim_lib\stratixii_atoms.v**
Stratix II GX (without transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiigx_atoms.v**
Stratix II GX (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixiigx_hssi_atoms.v**
Stratix	<*Quartus II installation directory*>**\eda\sim_lib\stratix_atoms.v**
Stratix GX	<*Quartus II installation directory*>**\eda\sim_lib\stratixgx_atoms.v**
Stratix GX (with transceiver block)	<*Quartus II installation directory*>**\eda\sim_lib\stratixgx_hssi_atoms.v**
Cyclone II	<*Quartus II installation directory*>**\eda\sim_lib\cycloneii_atoms.v**
Cyclone	<*Quartus II installation directory*>**\eda\sim_lib\cyclone_atoms.v**
MAX II	<*Quartus II installation directory*>**\eda\sim_lib\maxii_atoms.v**
MAX 7000 and MAX 3000	<*Quartus II installation directory*>**\eda\sim_lib\max_atoms.v**
APEX II	<*Quartus II installation directory*>**\eda\sim_lib\apexii_atoms.v**
APEX 20K	<*Quartus II installation directory*>**\eda\sim_lib\apex20k_atoms.v**
APEX 20KC, APEX 20KE, and Excalibur	<*Quartus II installation directory*>**\eda\sim_lib\apex20ke_atoms.v**
FLEX 10KE and ACEX 1K	<*Quartus II installation directory*>**\eda\sim_lib\flex10ke_atoms.v**

Simulation Netlist Files

Simulation netlist files are required to perform post-synthesis simulation or gate-level timing simulation. These simulation netlist files are generated from the EDA Netlist Writer.

If you are performing post-synthesis simulation, the Verilog Output File (***.vo**) or VHDL Output File (***.vho**) is required. The section "Generate Post-Synthesis Simulation Netlist Files" on page 5–7 shows you the steps to generate post-synthesis simulation netlist files for ***.vo** or ***.vho** files.

If you are performing gate-level timing simulation, not only is the ***.vo** file or ***.vho** file required, but the Standard Delay Format Output File (*.**sdo**) is also required. The ***.sdo** file is used to annotate the delay for the elements found in the ***.vo** or ***.vho** file. The section "Generate Gate-Level Timing Simulation Netlist Files" on page 5–8 shows you how to generate gate-level timing simulation netlist files (that is, ***.vo** or ***.vho** files and ***.sdo** files).

Generate Post-Synthesis Simulation Netlist Files

The following steps describe the process for generating post-synthesis simulation netlist files in the Quartus II software:

1. Perform Analysis and Synthesis. On the Processing menu, point to **Start** and click **Start Analysis and Synthesis** (you can also do this after step 2).
2. Turn on the **Generate Netlist for Functional Simulation Only** option by performing the following steps:
   a. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
   b. In the **Category** list, click the "+" icon to expand **EDA Tools Settings** and select **Simulation**. The **Simulation** page appears.
   c. In the **Tool name** list, select **Active-HDL**.
   d. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
   e. Click **More EDA Netlist Writer Settings**. The **More EDA Netlist Writer Settings** dialog box appears.
   f. In the **Existing option settings** list, click **Generate netlist for functional simulation only**.
   g. Under **Options**, from the **Setting** list, select **On**.
   h. Click **OK**.
   i. In the **Settings** dialog box, click **OK**.
3. Run the EDA Netlist Writer. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer**.

During the EDA Netlist Writer stage, the Quartus II software produces a **.vo** or **.vho** file that can be used for post-synthesis simulations in the Active-HDL software. This netlist file is mapped to architecture-specific primitives. No timing information is included at this stage. The resulting netlist is located in the output directory you specified in the **Settings** dialog box, which defaults to the *<project directory>***/simulation/activehdl** directory.

If you want to generate a post-synthesis simulation netlist with just the cell delays, you can generate the ***.sdo** file without running the Fitter. In this case, the ***.sdo** file includes all timing values for only the device cells. Interconnect delays are not included because fitting (placement and routing) has not been performed. To generate the post-synthesis netlist and ***.sdo** file, type the following commands at a command prompt:

```
quartus_map <project name> -c <revision name> ↵
quartus_sta <project name> -c <revision name> --post_map --zero_ic_delays ↵
```

or

```
quartus_tan <project name> -c <revision name> --post_map --zero_ic_delays ↵
quartus_eda <project name> -c <revision name> --simulation --tool= <3rd party EDA tool> \
--format=<HDL language> ↵
```

For more information about the **–format** and **–tool** options, type the following command at a command prompt:

```
quartus_eda -help=<options> ↵
```

Generate Gate-Level Timing Simulation Netlist Files

To perform gate-level timing simulation, the Active-HDL software requires information about how the design was placed into device-specific architectural blocks. The Quartus II software provides this information in the form of a ***.vo** file for Verilog HDL designs and a ***.vho** file for VHDL designs. The accompanying timing information is stored in the **.sdo** file, which annotates the delay for the elements found in the Verilog Output file or VHDL Output file.

Perform he following steps describe to generate a gate-level timing simulation netlist file in the Quartus II software:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the "+" icon to expand **EDA Tool Settings** and select **Simulation**. The **Simulation** page appears.
3. In the **Tool name** list, select **Active-HDL**.
4. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
5. Click **OK**.
6. In the **Settings** dialog box, click **OK**.
7. If you have not run a full compilation, perform a full compilation. On the Processing menu, click **Start Compilation**.
8. If you have already run a full compilation, run the EDA Netlist Writer. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer**.

   During the full compilation or EDA Netlist Writer stage, the Quartus II software produces a ***.vo** file, a ***.vho** file, and a ***.sdo** file used for gate-level timing simulations in the Active-HDL software. This netlist file is mapped to architecture-specific primitives. The timing information for the netlist is included in the *.**sdo** file. The resulting netlist is located in the output directory you specified in the **Settings** dialog box, which defaults to *<project directory>***/simulation/activehdl**.

Generating Different Timing Models

In Stratix III and later devices (for example, Cyclone III and Stratix IV), you can specify different temperature and voltage parameters to generate the timing netlist. If you enable the Quartus II Classic or Quartus II TimeQuest Timing Analyzer when generating the simulation netlist files (***.vo** or ***.vho** and ***.sdo**), different timing models for different operating conditions are used by default. Multicorner timing analysis is run by default during the full compilation.

Table 5–6 shows an example of available default operation conditions (model, voltage, and temperature) for Stratix III and Cyclone III devices.

Table 5–6. Default Available Operating Condition for Stratix III and Cyclone III Devices

Device Family	Model	Voltage	Temperature (°C)
Stratix III	Slow	1100 mV	85°
	Slow	1100 mV	0°
	Fast	1100 mV	0°
Cyclone III	Slow	1200 mV	85°
	Slow	1200 mV	0°
	Fast	1200 mV	0°

If multicorner timing analysis is not run during full compilation, perform the following steps to manually generate the simulation netlist files (***.vo** or ***.vho** and ***.sdo**) for the three different operating conditions listed in Table 5–6:

1. Generate all the available corner models at all operating conditions. Type the following command at a command prompt:

   ```
   quartus_sta <project name> --multicorner ↵
   ```

2. Generate the simulation output files for all three corners specified in Table 5–6. Perform steps 2 through 8 in "Generate Gate-Level Timing Simulation Netlist Files" on page 5–8. The output files are generated in the simulation output directory.

The following examples show all the simulation netlist files generated for the three operating conditions in Table 5–6, when **Verilog** is selected as the output netlist format:

First Slow Corner (Slow, 1100 mV, 85° C):

.vo file— <*revision name*>**.vo**
.sdo file— <*revision name*>**_v.sdo**

The <*revision_name*>**.vo** and <*revision name*>**_v.sdo** are generated for backward compatibility issues in case there are existing scripts that still use them.

.vo file— <*revision name*>_<*speedgrade*>_1100mv_85c_slow**.vo**
.sdo file— <*revision name*>_<*speedgrade*>_1100mv_85c_v_slow**.sdo**

Second Slow Corner (Slow, 1100 mV, 0° C):

.vo file— <*revision name*>_<*speedgrade*>_1100mv_0c_slow**.vo**
.sdo file— <*revision name*>_<*speedgrade*>_1100mv_0c_v_slow**.sdo**

Fast Corner (Fast, 1100 mV, 0° C):

.vo file— <*revision name*>_min_1100mv_0c_fast**.vo**
.sdo file— <*revision name*>_min_1100mv_0c_v_fast**.sdo**

For older devices, a slow-corner (worst case) timing model is used by default. There are only two timing models available: the slow-corner and the fast-corner timing model. To generate the ***.sdo** file using a different timing model, you must run the Quartus II Classic or the Quartus II TimeQuest Timing Analyzer with a different timing model before you start the EDA Netlist Writer.

To run the Quartus II Classic Timing Analyzer with the best-case model, on the Processing menu, point to **Start** and click **Start Classic Timing Analyzer (Fast Timing Model)**. After timing analysis is complete, the Compilation Report appears. You can also type the following command at a command prompt:

```
quartus_tan <project_name> --fast_model=on ↵
```

To run the Quartus II TimeQuest Timing Analyzer with a best-case model, use the **-fast_model** option after you create the timing netlist. The following command enables fast timing models:

```
create_timing_netlist --fast_model ↵
```

After you run the Quartus II Classic or Quartus II TimeQuest Timing Analyzer, perform steps 2 through 8 in "Generate Gate-Level Timing Simulation Netlist Files" on page 5–8 to generate the ***.sdo** file. For fast corner timing models, the **–fast** post fix is added to the ***.vo** or ***.vho**, and ***.sdo** file (for example, **my_project_fast.vo** or **my_project_fast.vho**, and **my_project_fast.sdo**).

For more information about generating the timing netlist with a different timing model, refer to the *Quartus II Classic Timing Analyzer* or the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Disable Timing Violation on Registers

In certain situations, the timing violation can be ignored and you can disable the timing violation on registers; for example, timing violations that occur in internal synchronization registers used for asynchronous clock-domain crossing.

By default, the **x_on_violation_option logic** option is **On**, which means the simulation shows "x" whenever a timing violation occurs. To disable showing the timing violation on certain registers, set the **x_on_violation_option logic** option to **Off** on those registers. The following command is an example of the QSF file:

```
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF -to <register_name>
```

Compile Libraries Using the EDA Simulation Library Compiler

The EDA Simulation Library Compiler is used to compile Verilog HDL and VHDL simulation libraries for all Altera devices and supported third-party simulators. You can use this tool to compile all libraries required by RTL and gate-level simulation.

When the compilation targets third-party simulation tools such as ModelSim, VCS-MX, Active-HDL, Riviera-PRO, and NC-Sim, the compiled libraries are kept in either the directory you set or the default directory. When you perform the simulation using these simulators, you can use or re-use the compiled libraries and avoid the overhead associated with redundant library compilations.

However, if the Synopsys VCS software performs the compilation while running the EDA Simulation Library Compiler, the option files (**simlib_comp.vcs**) are generated after compilation. You can then include your design and test bench files in the option files and invoke them with the `vcs` command.

Before using this option, ensure the appropriate simulation tools are already installed and their paths are specified. To specify the path, refer to "Setting Up NativeLink" on page 5–45.

Run the EDA Simulation Library Compiler through the GUI

Starting with the Quartus II software 9.0 release, the EDA Simulation Library Compiler contains a GUI. To compile libraries with the EDA Simulation Library Compiler GUI, perform the following steps:

1. On the Tools menu, click **EDA Simulation Library Compiler**. The **EDA Simulation Library Compiler** dialog box appears.
2. In the **Tool name** entry box under **EDA simulation tool**, select a simulation tool.

   The **Executable location** box displays the location of the simulation tool you specified. This location must be set before running the EDA Simulation Library Compiler.
3. Under **Library families**, select one or more device families for your design compilation and move them to **Selected families** box.
4. Under **Library language**, select **VHDL**, **Verilog**, or both.
5. In the **Output directory** field, specify a location in which to store the compiled libraries or option files.
6. Click **Start Compilation**.

For example, if you want to simulate a Verilog HDL design on a Stratix II device in the Active-HDL simulator, do the following:

- Select **Active-HDL** in the **Tool name** field.
- Move **Stratix II** from the **Available families** list to the **Selected families** list.
- Select **Verilog** in the **Library language** field.
- Specify a location in the **Output directory** field in which to keep the option file.
- Click **Start Compilation**.

When the EDA Simulation Library Compiler finishes, all required libraries are compiled and stored in the output location you specified. The next time you perform simulation in Active-HDL, you only have to compile your design and testbench files. You do not have to compile the Altera libraries again.

Run EDA Simulation Library Compiler In Command Line

You can run the EDA Simulation Library Compiler in the command line. Type the following command:

```
quartus_sh --simlib_comp -family <device> -tool <simulation tool name>
-language <language> -directory <directory> ↵
```

For more information about the command’s options and how to define them, type the following command:

```
quartus_sh --help=simlib_comp ↵
```

Perform Simulation Using the Active-HDL Software (GUI Mode)

Perform simulation of Verilog HDL or VHDL designs with Active-HDL software at various levels to verify designs from different aspects. There are three types of simulation:

- RTL functional simulation
- Post-synthesis simulation
- Gate-level timing simulation

Simulation helps you verify your designs and debug them against any possible errors the designs may have. The following sections provide step-by-step instructions to perform the simulation through the GUI.

For high-speed simulation, you must select **ps** in the **Resolution** list for your simulator resolutions. If you choose slower than **ps**, the high-speed simulation may fail.

Workspace creation is the mandatory first step to start working in the Active-HDL GUI. You must create a new workspace to add the simulation model files, design files, and testbench file before you can compile them.

Simulating VHDL Designs

Simulation of a VHDL design using the Active-HDL GUI is user friendly. You do not have to remember the commands to compile the libraries or load and simulate the VHDL design files. You can use the Active-HDL GUI to perform the RTL functional simulation, post-synthesis simulation, and gate-level timing simulation. The following sections show you how to perform simulation at various levels through the Active-HDL GUI.

Perform RTL Functional Simulation

RTL functional simulation is typically performed to verify the syntax of the code and to check the functionality of the design. The following sections show how to perform RTL functional simulation in the Active-HDL software for VHDL designs.

If your Active-HDL software comes with precompiled simulation libraries, creating or compiling simulation libraries is not required and you can skip the following section.

Create and Compile Simulation Libraries

Simulation libraries are required to simulate a design that contains an Altera primitive, lpm function, or Altera megafunction. Different designs require different simulation libraries, and you must create these required simulation libraries before you can run the simulation.

For example, the library name for Altera megafunctions should be **altera_mf** and the library name for LPM should be **lpm**. For a list of all functional simulation library files, refer to "RTL Functional Simulation Library Files" on page 5–4.

Before you start creating simulation libraries (for example, **altera_mf** and **lpm**) ensure that the library with the same name is not present in the Active-HDL software. Perform the following steps:

1. In the Active-HDL software, on the View menu, click **Library Manager.** The Library Manager window appears.
2. Check to see if the simulation libraries (for example, **altera_mf** and **lpm**) that you are going to create are not already present.
3. If the simulation libraries are already present, you must detach them. To detach the simulation libraries, right-click on the library and select **Detach**.

To create and compile the simulation libraries, you must create a new workspace. Perform the following steps to create a new workspace, create a new library, compile the library, and register the library in the **Library Manager**.

1. In the Active-HDL software, on the File menu, point to **New** and click **Design**. The **New Design Wizard** appears.
2. Select **Create an Empty Design** and keep the **Create New Workspace** option selected.
3. Click **Next**. The **Property** page appears.
4. In the **Property** page, click **Next** to proceed to the **Design name** and **Library name** fields.
5. Type the design name (for example, `altera_mf` or `lpm`), select the location of your RTL design in the **Design folder** field, and click **Next**. For simplicity, keep the design name and the library name the same.
6. Click **Finish** to complete the wizard.
7. On the Design menu, click **Add files to Design**.
8. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** and add the necessary simulation model files.

   Compile the **altera_mf_components.vhd** and **altera_mf.vhd** model files into the **altera_mf** library. Compile the **220pack.vhd** and **220model.vhd** model files into the **lpm** library.
9. On the Design menu, click **Compile All** to compile all the files (for example, **altera_mf_components.vhd** and **altera_mf.vhd**) in the design library.
10. On the File menu, click **Close Workspace**.
11. You must register the created library in the Active-HDL software. On the View menu, click **Library Manager**. The Library Manager window appears.
12. On the Library menu, click **Attach Library**.
13. Locate the **.lib** file (for example, **altera_mf.lib** or **lpm.lib**) from the design directory that you created in the previous steps and click **Open**. This action attaches the simulation library as a global library inside your library manager and make it visible for any design in Active-HDL.
14. Repeat this procedure to create and compile another simulation library.

Compile Testbench and Design Files into the Work Library

To compile design files and the testbench into a work library, you must create a new workspace. Perform the following steps to create a new workspace and compile your testbench and design files into the work library:

1. In the Active-HDL software, on the File menu, point to **New** and click **Design**. The **New Design Wizard** appears.
2. Select **Create an Empty Design** and keep the **Create New Workspace** option selected.
3. Click **Next**. The **Property** page appears.
4. In the **Property** page, click **Next** to reach the design name and library name options.
5. Type `work` for the design name and select the location of your RTL design. For simplicity, keep the design name and the library name the same.
6. Click **Finish** to complete the wizard.
7. On the Design menu, click **Add files to Design.**
8. Browse to the RTL design directory and add the testbench and RTL design files.
9. On the Design menu, click **Compile All** to compile the testbench and RTL design files.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following steps:

1. In the **Design Browser**, click the **Top-level Selection** pull-down menu. Select the top-level entity, which is your testbench with corresponding architecture.
2. On the Simulation menu, select **Initialize Simulation**. This loads the simulation.
3. The **Design Browser** automatically switches to the **Structure** tab and shows you the design tree.

Running the Simulation

To run a simulation, perform the following steps:

1. On the File menu, point to **New** and click **Waveform**.
2. Drag signals of interest from the **Design Browser** (in the **Structure** tab) to the Waveform window.
3. On the Simulation menu, click **Run Until**. A pop-up window appears.
4. Specify how long you want your simulation to run (for example, 500 ns).

Perform Post-Synthesis Simulation

Perform post-synthesis simulation to verify that functionality of the design is not lost after synthesis. You can create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the Active-HDL software.

Before you run post-synthesis simulation, generate post-synthesis simulation netlist files. Refer to "Generate Post-Synthesis Simulation Netlist Files" on page 5–7.

Use the instructions in the following section to perform a post-synthesis simulation for VHDL designs in the Active-HDL software.

If your Active-HDL software comes with precompiled simulation libraries, creating or compiling simulation libraries is not required, and you can skip the following section.

Create and Compile Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-synthesis primitives. Depending on the device family you are using, you must create the required simulation libraries before running the simulation.

Performing post-synthesis simulation requires a gate-level timing simulation library. For example, the library name for the Stratix III family should be **stratixiii**. For a list of all gate-level timing simulation library files, refer to "Gate-Level Timing Simulation Library Files" on page 5–4.

Before you create simulation libraries (for example, **stratixiii**) ensure that a library with the same name is not present in the Active-HDL software. Perform the following steps:

1. In the Active-HDL software, on the View menu, click **Library Manager.** The Library Manager window appears.
2. Check that the simulation libraries (for example, **stratixiii**) you are going to create are not already present.
3. If these simulation libraries are present, you must detach them. To detach these simulation libraries, right-click the library and from the menu list, select **Detach**.

To create and compile the simulation libraries, you must create a new workspace. Perform the following steps to create a new workspace, create a new library, and compile and register the library in the **Library Manager**. Perform the following steps:

1. In the Active-HDL software, on the File menu, point to **New** and click **Design**. The **New Design Wizard** appears.
2. Select **Create an Empty Design** and keep the **Create New Workspace** option selected.
3. Click **Next**. The **Property** page appears.
4. In the **Property** page, click **Next** to reach the design name and library name options.
5. Type the design name (for example, `stratixiii`) and select the location of your RTL design. For simplicity, keep the design name and the library name the same.
6. Click **Finish** to complete the wizard.
7. On the Design menu, click **Add files to Design.**
8. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** and add the necessary simulation model files.

Compile the **stratixiii_atoms_components.vhd** and **stratixiii_atoms.vhd** model files into the **stratixiii** library.

9. On the Design menu, click **Compile All** to compile all the files (for example, **altera_mf_components.vhd** and **altera_mf.vhd**) to the design library.
10. On the File menu, click **Close Workspace**.
11. To register the created library in the Active-HDL software, on the View menu, click **Library Manager**. The Library Manager window appears.
12. On the Library menu, click **Attach Library.**
13. Locate the **.lib** file (for example, **stratixiii.lib**) from the design directory that you created in the previous steps and click **Open**. This attaches the simulation library as a global library inside your library manager and makes it visible for any design in the Active-HDL software.
14. Repeat this procedure to create and compile another simulation library, if necessary.

Compile the Testbench and Design File into the Work Library

To compile design files and the testbench into a work library, you must create a new workspace. Perform the following steps to create a new workspace, and compile your testbench and ***.vho** file into the work library:

1. In the Active-HDL software, on the File menu, point to **New** and click **Design**. The **New Design Wizard** appears.
2. Select **Create an Empty Design** and keep the **Create New Workspace** option selected.
3. Click **Next**. The **Property** page appears.
4. In the **Property** page, click **Next** to reach the design name and library name options.
5. Type `work` for the design name and select the location of your RTL design. For simplicity, keep the design name and the library name the same.
6. Click **Finish** to complete the wizard.
7. On the Design menu, click **Add files to Design**.
8. Browse to the VHDL output file directory (for example, *<project directory>***/simulation/activehdl**) and add the VHDL output file (***.vho**). Browse to the testbench file directory and add the testbench file.
9. On the Design menu, click **Compile All** to compile the testbench and VHDL output netlist files.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following steps:

1. In the **Design Browser**, click the **Top-level Selection** list. Select the top-level entity, which is your testbench with corresponding architecture.

2. On the Simulation menu, click **Initialize Simulation**. This loads the simulation.
3. The **Design Browser** automatically switches to the **Structure** tab and displays the design tree.

Running the Simulation

To run the simulation, perform the following steps:

1. On the File menu, point to **New** and click **Waveform**.
2. Drag signals of interest from the **Design Browser** (in the **Structure** tab) to the Waveform window.
3. On the Simulation menu, click **Run Until**.
4. In the pop-up window, specify how long you want your simulation to run (for example, 500 ns).

Perform Gate-Level Timing Simulation

Gate-level timing simulation is a very important step in ensuring that the FPGA's functionality is still correct and meets all the timing requirements after the design was placed and routed. You can create the gate-level netlist in the Quartus II software and use the netlist to perform gate-level simulation with the Active-HDL software.

Before you run gate-level timing simulation, generate gate-level timing simulation netlist files. Refer to "Generate Gate-Level Timing Simulation Netlist Files" on page 5–8.

Use the following instructions to perform a gate-level timing simulation for VHDL designs in the Active-HDL software.

If your Active-HDL software comes with precompiled simulation libraries, creating or compiling simulation libraries is not required, and you can skip the next section.

Create and Compile Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-fitting primitives. Depending on the device family you are using, you must create the required simulation libraries before running the simulation.

Performing gate-level timing simulation requires a gate-level timing simulation library. For example, the library name for the Stratix III family should be **stratixiii**. For a list of all gate-level timing simulation library files, refer to "Gate-Level Timing Simulation Library Files" on page 5–4.

Before you create simulation libraries (for example, **stratixiii**), ensure that a library with the same name is not present in the Active-HDL software. Perform the following steps:

1. In the Active-HDL software, on the **View** menu, Click **Library Manager.** The Library Manager window appears.
2. Check to see that the simulation libraries (for example, **stratixiii**) you are going to create are not already present.
3. If these simulation libraries are present, you must detach them. Right-click the library and from the menu list, select **Detach**.

To create and compile simulation libraries, you must create a new workspace. Perform the following steps to create a new workspace, create a new library, and compile and register the library in the **Library Manager**. Do this by performing the following steps:

1. In the Active-HDL software, on the File menu, point to **New** and click **Design**. The **New Design Wizard** appears.
2. Select **Create an Empty Design** and keep the **Create New Workspace** option selected.
3. Click **Next**. The **Property** page appears.
4. In the **Property** page, click **Next** to reach the design name and library name options.
5. Type the design name (for example, `stratixiii`) and select the location of your RTL design. For simplicity, keep the design name and the library name the same.
6. Click **Finish** to complete the wizard.
7. On the Design menu, click **Add files to Design**.
8. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** and add the necessary simulation model files.

   Compile the **statixiii_atoms_components.vhd** and **stratixiii_atoms.vhd** model files into the **stratixiii** library.
9. On the Design menu, click **Compile All** to compile all the files (for example, **altera_mf_components.vhd** and **altera_mf.vhd**) into the design library.
10. On the File menu, click **Close Workspace**.
11. To register the created library in the Active-HDL software, on the **View** menu, click **Library Manager**. The Library Manager window appears.
12. On the Library menu, click **Attach Library**.
13. Locate the **.lib** file (for example, **stratixiii.lib**) from the design directory that you created in the previous steps and click **Open**. This attaches the simulation library as a global library inside your library manager and makes it visible to any design in the Active-HDL software.
14. Repeat this procedure to create and compile another simulation library, if necessary.

Compile the Testbench and Design File into the Work Library

To compile design files and the testbench into a work library, you must create a new workspace. Perform the following steps to create a new workspace and compile your design files into the work library:

1. In the Active-HDL software, on the File menu, point to **New** and click **Design**. The **New Design Wizard** appears.
2. Select **Create an Empty Design** and keep the **Create New Workspace** option selected.
3. Click **Next**. The **Property** page appears.

4. In the **Property** page, click **Next** to reach the design name and library name options.
5. Type `work` for the design name and select the location of your RTL design. For simplicity, keep the design name and the library name the same.
6. Click **Finish** to complete the wizard.
7. On the Design menu, click **Add files to Design**.
8. Browse to the VHDL output file directory (for example, <*project directory*>**/simulation/activehdl**) and add the VHDL output file (***.vho**) and standard output file (***.sdo**). Browse to the testbench file directory and add the testbench file.
9. On the Design menu, click **Compile All** to compile the testbench and VHDL output netlist files.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following steps:

1. On the Design menu, click **Settings.** The Design Settings window appears. Expand the **Simulation** category and click **SDF**.

   If there are no **.sdo** files listed, no **.sdo** files have been added to the design. You must add the **.sdo** file before running the timing simulation.

2. In the **Files-Region** dialog box, select the level of the design hierarchy to which the **.sdo** file should be bonded to. For example, if your design under test is instantiated in the testbench file under the **i1** label, the region should be '**i1/**'.
3. In the **SDF** settings box, set **Value** to **Average** and set **Load** to **Yes** so that the simulator loads this file upon simulation start.

You do not have to set the Value (Minimum, Average, Maximum) for the ***.sdo** file, because the Quartus II EDA Netlist Writer generates the ***.sdo** file using the same value for the triplet (minimum, average, and maximum timing values).

4. Click **OK** to close the Design Settings window.
5. In the **Design Browser**, click the **Top-level Selection** list. Select the top-level entity, which is your testbench with corresponding architecture.
6. On the Simulation menu, click **Initialize Simulation**. This loads the simulation.
7. The **Design Browser** automatically switches to the **Structure** tab and shows you the design tree.

Running the Simulation

To run the simulation, perform the following steps:

1. On the File menu, point to **New** and click **Waveform**.
2. Drag signals of interest from the **Design Browser** (in the **Structure** tab) to the Waveform window.

3. On the Simulation menu, click **Run Until**.
4. In the pop-up window, specify how long you want your simulation to run (for example, 500 ns).

Simulating Verilog HDL Designs

Simulation of Verilog HDL designs using the Active-HDL GUI is user friendly. You do not have to remember the commands to compile the libraries or to load and simulate the Verilog HDL design files. You can use the Active-HDL GUI to perform RTL functional simulation, post-synthesis simulation, and gate-level simulation. The following sections show you how to perform simulation at various levels through the Active-HDL GUI.

Perform RTL Functional Simulation

RTL functional simulation is typically performed to verify the syntax of the code and to check the functionality of the design. The following sections show how to perform RTL functional simulation in the Active-HDL software for Verilog HDL designs.

If your Active-HDL software comes with precompiled simulation libraries, creating or compiling simulation libraries is not required, and you can skip the next section.

Create and Compile Simulation Libraries

Simulation libraries are required to simulate a design that contains an Altera primitive, LPM function, or Altera megafunction. Different designs require different simulation libraries, and you must create these required simulation libraries before you can run the simulation.

For example, the library name for Altera megafunctions should be **altera_mf_ver** and the library name for LPM should be **lpm**. For a list of all functional simulation library files, refer to "RTL Functional Simulation Library Files" on page 5–4.

Before you start creating simulation libraries (for example, **altera_mf_ver** or **lpm_ver**), ensure that the library with the same name is not present in the Active-HDL software.

1. In the Active-HDL software, on the View menu, click **Library Manager.** The Library Manager window appears.
2. Check to see if the simulation libraries that you are going to create are not already present.
3. If the simulation libraries are already present, you must detach them. Right-click the library and from the menu list, select **Detach**.

To create and compile simulation libraries, you must create a new workspace. Perform the following steps to create a new workspace, create a new library, compile the library, and register the library in the **Library Manager**.

1. In the Active-HDL software, on the File menu, point to **New** and click **Design**. The **New Design Wizard** appears.
2. Select **Create an Empty Design** and keep the **Create New Workspace** option selected.
3. Click **Next**. The **Property** page appears.

4. In the **Property** page, click **Next** to proceed to the **Design name** and **Library name** fields.
5. Type the design name (for example, `altera_mf_ver` or `lpm_ver`), select the location of your RTL design in the **Design folder** field, and click **Next**. For simplicity, keep the design name and the library name the same.
6. Click **Finish** to complete the wizard.
7. On the Design menu, click **Add files to Design**.
8. Browse to the <*Quartus II installation directory*>**/eda/sim_lib** and add the necessary simulation model files.

   For example, compile the **altera_mf_.v** model files into the **altera_mf_ver** library, and compile the **220model.v** model files into the **lpm_ver** library.
9. On the Design menu, click **Compile All** to compile all the files (for example, **altera_mf.v**) into the design library.
10. On the File menu, click **Close Workspace**.
11. On the View menu, click **Library Manager**. The Library Manager window appears.
12. On the Library menu, click **Attach Library.**
13. Locate the **.lib** file (for example, **altera_mf_ver.lib**) from the design directory that you created in the previous steps and click **Open**. This attaches the simulation library as a global library inside your library manager and makes it visible to any design in Active-HDL.
14. Repeat this procedure to create and compile another simulation library.

Compile the Testbench and Design Files into the Work Library

To compile design files and the testbench into the work library, you must create a new workspace. Perform the following steps to create a new workspace and compile your testbench and design files into the work library:

1. In the Active-HDL software, on the File menu, point to **New** and click **Design**. The **New Design Wizard** appears.
2. Select **Create an Empty Design** and keep the **Create New Workspace** option selected.
3. Click **Next**. The **Property** page appears.
4. In the **Property** page, click **Next** to reach the design name and library name options.
5. Type `work` for the design name and select the location of your RTL design. For simplicity, keep the design name and the library name the same.
6. Click **Finish** to complete the wizard.
7. On the Design menu, click **Add files to Design**.
8. Browse to the RTL design directory and add the testbench and RTL design files.

9. Your design may require simulation libraries that you created previously to compile successfully. For example, the **altera_mf_ver** library is required for compiling designs that use Altera megafunctions. If your design requires these simulation libraries, perform the following steps:

   a. On the Design menu, click **Settings**. The Design Settings window appears. Expand the **Compilation** category and click **Verilog**.

   b. To add the **Verilog Library** settings, click the **Add library** icon (upper-right icon) and click **OK** to insert the required Verilog HDL simulation libraries.

   These simulation libraries are the simulation libraries that you compiled or installed previously (for example, **altera_mf_ver** and **lpm_ver**, **altera_ver**).

10. On the Design menu, click **Compile All** to compile the testbench and RTL design files.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following steps:

1. In the **Design Browser**, click the **Top-level Selection** list. Select the top-level module, which is your testbench.
2. To add the required simulation library, on the Design menu, click **Settings.** The Design Settings window appears.

   a. Expand the **Simulation** category and click **Verilog**.

   b. In the **Verilog Libraries** window, click the **Add library** icon (top-right icon) and click **OK** to insert the required Verilog HDL simulation libraries (for example, **altera_mf_ver, lpm_ver**, and **altera_ver**).

3. On the Simulation menu, click **Initialize Simulation**. This loads the simulation.
4. The **Design Browser** automatically switches to the **Structure** tab and shows you the design tree.

Running the Simulation

Perform the following steps to run the simulation:

1. On the File menu, point to **New** and click **Waveform**.
2. Drag signals of interest from the **Design Browser** (in the **Structure** tab) to the Waveform window.
3. On the Simulation menu, click **Run Until.**
4. In the pop-up window, specify how long you want your simulation to run (for example, 500 ns).

Perform Post-Synthesis Simulation

Perform post-synthesis simulation to verify that functionality of the design is not lost after synthesis. You can create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the Active-HDL software.

Before you run post-synthesis simulation, generate post-synthesis simulation netlist files. Refer to "Generate Post-Synthesis Simulation Netlist Files" on page 5–7.

Use the instructions in the following section to perform a post-synthesis simulation for Verilog HDL designs in the Active-HDL software.

If your Active-HDL software comes with precompiled simulation libraries, creating or compiling simulation libraries is not required and you can skip the following section.

Create and Compile Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-synthesis primitives. Depending on the device family you are using, you must create the required simulation libraries before running the simulation.

Performing post-synthesis simulation requires a gate-level timing simulation library. For example, the library name for the Stratix III family should be **stratixiii_ver**. For a list of all gate-level timing simulation library files, refer to "Gate-Level Timing Simulation Library Files" on page 5–4.

Before you create simulation libraries (for example, **stratixiii_ver**), ensure that a library with the same name is not present in the Active-HDL software. Perform the following steps:

1. In the Active-HDL software, on the View menu, click **Library Manager.** The Library Manager window appears.
2. Check to see that the simulation libraries you are going to create are not already present.
3. If these simulation libraries are present, you must detach them. Right-click the library and from the menu list, select **Detach**.

To create and compile the simulation libraries, you must create a new workspace. Perform the following steps to create a new workspace, create a new library, and compile and register the library in the **Library Manager**. Perform the following steps:

1. In the Active-HDL software, on the File menu, point to **New** and click **Design**. The **New Design Wizard** appears.
2. Select **Create an Empty Design** and keep the **Create New Workspace** option selected.
3. Click **Next**. The **Property** page appears.
4. In the **Property** page, click **Next**.
5. Type the design name (for example, `stratixiii_ver`) and select the location of your RTL design. For simplicity, keep the design name and the library name the same.
6. Click **Finish** to complete the wizard.
7. On the Design menu, click **Add files to Design**.
8. Browse to <*Quartus II installation directory*>**/eda/sim_lib** and add the necessary simulation model files.

   For example, compile the **statixiii.v** model file into the **stratixiii_ver** library.

9. On the Design menu, click **Compile All** to compile all the files (for example, **stratixiii.v**) to the design library.
10. On the File menu, click **Close Workspace**.
11. On the View menu, click **Library Manager**. The **Library Manager** window appears.
12. On the Library menu, click **Attach Library**.
13. Locate the **.lib** file (for example, **stratixiii.lib**) from the design directory that you created in the previous steps and click **Open**. This attaches the simulation library as a global library inside your library manager and makes it visible to any design in the Active-HDL software.
14. Repeat this procedure to create and compile another simulation library, if necessary.

Compile the Testbench and Design File into the Work Library

To compile design files and the testbench into a work library, you must create a new workspace. Perform the following steps to create a new workspace, and compile your testbench and ***.vo** file into the work library:

1. In the Active-HDL software, on the File menu, point to **New** and click **Design**. The **New Design Wizard** appears.
2. Select **Create an Empty Design** and keep the **Create New Workspace** option selected.
3. Click **Next**. The **Property** page appears.
4. In the **Property** page, click **Next** to reach the design name and library name options.
5. Type `work` for the design name and select the location of your RTL design. For simplicity, keep the design name and the library name the same.
6. Click **Finish** to complete the wizard.
7. On the Design menu, click **Add files to Design**.
8. Browse to the Verilog HDL output file directory (for example, *<project directory>***/simulation/activehdl**) and add the Verilog output file (***.vo**). Browse to the testbench file directory and add the testbench file.
9. Your design may require simulation libraries that you created previously to compile successfully. For example, the **stratixiii** library is required for compiling designs that use Stratix III devices. If your design requires these simulation libraries, perform the following steps:
    a. On the Design menu, click **Settings.** The Design Settings window appears. Expand the **Compilation** category and click **Verilog**.
    b. To add the **Verilog Library** settings, click the **Add library** icon (upper-right icon), and click **OK** to insert the required Verilog HDL simulation libraries.

    ☞ These simulation libraries are the simulation libraries you compiled or installed previously.

10. On the Design menu, click **Compile All** to compile the testbench and ***.vo** files.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following steps:

1. In the **Design Browser**, click the **Top-level Selection** list. Select the top-level module, which is your testbench.
2. To add the required simulation library, on the Design menu, click **Settings.** The Design Settings window appears.
   a. Expand the **Simulation** category and click **Verilog**.
   b. In the **Verilog Libraries** window, click the **Add library** icon (top-right icon) and click **OK** to insert the required Verilog HDL simulation libraries (for example, **stratixiii_ver**).
3. On the Simulation menu, click **Initialize Simulation**. This loads the simulation.
4. The **Design Browser** automatically switches to the **Structure** tab and shows you the design tree.

Running the Simulation

Perform the following steps to run a simulation:

1. On the File menu, point to **New** and click **Waveform**.
2. Drag signals of interest from the **Design Browser** (in the **Structure** tab) to the **Waveform** window.
3. On the Simulation menu, click **Run Until**.
4. In the pop-up window, specify how long you want your simulation to run (for example, 500 ns).

Perform Gate-Level Timing Simulation

Gate-level timing simulation is a very important step in ensuring that the FPGA's functionality is still correct and meets all of the timing requirements after the design was placed and routed. You can create the gate-level netlist in the Quartus II software and use the netlist to perform gate-level simulation with the Active-HDL software.

Before you run the gate-level timing simulation, generate gate-level timing simulation netlist files. Refer to "Generate Gate-Level Timing Simulation Netlist Files" on page 5–8.

Use the following instructions to perform a gate-level timing simulation for Verilog HDL designs in the Active-HDL software.

If your Active-HDL software comes with precompiled simulation libraries, creating or compiling simulation libraries is not required, and you can skip the next section.

Create and Compile Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-fitting primitives. Depending on the device family you are using, you must create the required simulation libraries before running the simulation.

Performing gate-level timing simulation requires a gate-level timing simulation library. For example, the library name for the Stratix III family should be **stratixiii_ver**. For a list of all gate-level timing simulation library files, refer to "Gate-Level Timing Simulation Library Files" on page 5–4.

Before you create simulation libraries (for example, **stratixiii_ver**), ensure that a library with the same name is not present in the Active-HDL software. Perform the following steps:

1. In the Active-HDL software, on the View menu, click **Library Manager.** The **Library Manager** window appears.
2. Check to see that the simulation libraries (for example, **stratixiii_ver**) you are going to create are not already present.
3. If these simulation libraries are present, you must detach them. Right-click the library and from the menu list, click **Detach**.

To create and compile simulation libraries, you must create a new workspace. Perform the following steps to create a new workspace and a new library, and compile and register the library in the **Library Manager**:

1. In the Active-HDL software, on the File menu, point to **New** and click **Design**. The **New Design Wizard** appears.
2. Select **Create an Empty Design** and keep the **Create New Workspace** option selected.
3. Click **Next**. The **Property** page appears.
4. In the **Property** page, click **Next** to reach the design name and library name options.
5. Type the design name (for example, `stratixiii_ver`) and select the location of your RTL design. For simplicity, keep the design name and the library name the same.
6. Click **Finish** to complete the wizard.
7. On the Design menu, click **Add files to Design**.
8. Browse to <*Quartus II installation directory*>**/eda/sim_lib** and add the necessary simulation model files.

   For example, compile the **statixiii_atoms.v** model files into the **stratixiii_ver** library.
9. On the Design menu, click **Compile All** to compile all the files (for example, **statixiii_atoms.v**) to the design library.
10. On the File menu, click **Close Workspace**.
11. You must now register the created library in the Active-HDL software. On the **View** menu, click **Library Manager**. The Library Manager window appears.
12. On the Library menu, click **Attach Library**.

13. Locate the **.lib** file (for example, **stratixiii.lib**) in your design directory that you created in the previous steps and click **Open**. This attaches the simulation library as a global library inside your library manager and makes it visible for any design in the Active-HDL software.
14. Repeat this procedure to create and compile another simulation library, if necessary.

Compile the Testbench and Design File into the Work Library

To compile design files and the testbench into a work library, you must create a new workspace. Perform the following steps to create a new workspace and compile your testbench and ***.vo** file into the work library:

1. In the Active-HDL software, on the File menu, point to **New** and click **Design**. The **New Design Wizard** appears.
2. Select **Create an Empty Design** and keep the **Create New Workspace** option selected.
3. Click **Next**. The **Property** page appears.
4. In the **Property** page, click **Next** to reach the design name and library name options.
5. Type `work` for the design name and select the location of your RTL design. For simplicity, keep the design name and the library name the same.
6. Click **Finish** to complete the wizard.
7. On the Design menu, click **Add files to Design**.
8. Browse to the Verilog HDL output file directory (for example, *<project directory>***/simulation/activehdl**) and add the ***.vo** file and ***.sdo** file. Browse to the testbench file directory and add the testbench file.
9. On the Design menu, click **Compile All** to compile the testbench and Verilog HDL output netlist files.

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following steps:

1. In the **Design Browser**, click the **Top-level Selection** list. Select the top-level entity, which is your testbench with corresponding architecture.
2. On the Simulation menu, click **Initialize Simulation**. This loads the simulation.
3. The **Design Browser** automatically switches to the **Structure** tab and shows you the design tree.

Running the Simulation

To run the simulation, perform the following steps:

1. On the File menu, point to **New** and click **Waveform**.
2. Drag signals of interest from the **Design Browser** (in the **Structure** tab) to the Waveform window.

3. On the Simulation menu, click **Run Until.**
4. In the pop-up window, specify how long you want your simulation to run (for example, 500 ns).

Perform Simulation Using the Active-HDL Software (Batch Mode)

Perform simulation of Verilog HDL or VHDL designs with Active-HDL software at various levels to verify designs from different aspects. There are three categories of simulation:

- RTL functional simulation
- Post-synthesis simulation
- Gate-level simulation

Simulation helps you verify your design and debug it against any possible errors in the design.

For high-speed simulation, you must select **ps** in the **Resolution** list for your simulator resolutions. If you choose slower than **ps**, the high-speed simulation may fail.

There are two modes in which the Active-HDL software can operate in the batch mode:

- Shell mode
- Command-line mode

In shell mode, all commands are executed in the Active-HDL shell that is started by **vsimsa.bat**. Commands can be grouped into macro files (***.do**) that are executed within the tool shell.

In command-line mode, standalone commands, such as `vlib`, `vcom`, and `vsim`, are executed in the system shell (for example, `cygwin`). These standalone commands can be grouped into script files (**tcl**, **perl**, **windows batch**) that are run from the system shell.

Before running Active-HDL in a standalone command, make sure that the **Active-HDL/bin** directory is located in PATH environment variables.

If you are running the command from the Active-HDL GUI console, workspace creation is the first step. The following commands create the workspace and open the workspace:

```
createdesign <workspace name> <your design path> ↵
opendesign -a <workspace name>.adf ↵
```

Simulating the VHDL Designs

The following sections show how to perform RTL functional, post-synthesis, and gate-level timing simulations in command-line mode or shell mode of the Active-HDL software.

 © March 2009 Altera Corporation

Perform RTL Functional Simulation

RTL functional simulation is typically performed to verify the syntax of the code and to check the functionality of the design. The following sections show how to perform RTL functional simulation in the Active-HDL software for VHDL designs.

If your Active-HDL software comes with precompiled simulation libraries, creating or compiling simulation libraries is not required and you can skip the following section.

Create Simulation Libraries

Simulation libraries are required to simulate a design that contains an Altera primitive, LPM function, or Altera megafunction. Depending on your design, you must create the required simulation libraries and link them to your design correctly.

Execute the following command to create a simulation library:

```
vlib <library_name> ↵
```

The library name for Altera megafunctions should be **altera_mf** and the library name for LPM should be **lpm**. For a list of all functional simulation library files, refer to "RTL Functional Simulation Library Files" on page 5–4.

For example, to create simulation libraries for **altera_mf, lpm**, and **altera**, type the following commands

```
vlib altera_mf ↵
vlib lpm ↵
vlib altera ↵
```

Compile Simulation Models into Simulation Libraries

Type the following command to compile simulation models into simulation libraries:

```
vcom -work <simulation_library> <Quartus II installation directory>
/eda/sim_lib/<simulation_library_files> ↵
```

For example, compile the **altera_mf_components.vhd** and **altera_mf.vhd** model files into the **altera_mf** library, and the **220pack.vhd** and **220model.vhd** model files into the LPM library.

Use Example 5–1 to compile the simulation model files into the simulation libraries for **altera_mf**, **lpm**, and **altera**:

Example 5–1.

```
vcom -work altera_mf \
<Quartus II installation directory>/eda/sim_lib/altera_mf_components.v <Quartus II \
installation directory>/eda/sim_lib/altera_mf.vhd
vcom -work lpm \
<Quartus II installation directory>/eda/sim_lib/220model.vhd \
<Quartus II installation directory>/eda/sim_lib/220model.vhd
vcom -work altera <Quartus II installation directory> \
/eda/sim_lib/altera_primitives_components.vhd \
<Quartus II installation directory>/eda/sim_lib/altera_primitives.vhd
```

Compile the Testbench and Design Files into the Work Library

The following commands show how to compile your testbench and design files into the work library in the Active-HDL software's batch mode.

Type the following command to create the work library:

```
vlib work ↵
```

Type the following command to compile the testbench and design files into work library:

```
vcom -work work <my_testbench>.vhd <my_design_files>.vhd ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, type the following command:

```
vsim -t ps -L <library1> -L <library2> work .<my_testbench> ↵
```

<library1> and *<library2>* are the required libraries to load the design. If you have multiple libraries, use the `-L` option multiple times with the `vsim` command.

Running the Simulation

Type the following commands to trace the signals and run the simulation:

```
trace <hierarchical signal name> ↵
run <time period> ↵
```

The following commands trace the `DUT/clk1` and `DUT/U1/DATA` signal and run the simulation for 100 ps:

```
trace DUT/clk1 ↵
trace DUT/U1/DATA ↵
....
run 100 ps ↵
```

When the simulation is finished, type the following command to end the simulation:

```
endsim ↵
```

To view the simulation result in the GUI waveform, perform the following command to open the **wave.asdb** file:

```
open -asdb {<your_design_path> \work\src\wave.asdb} ↵
```

Perform Post-Synthesis Simulation

Perform post-synthesis simulation to verify that functionality of the design is not lost after synthesis. You can create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the Active-HDL software.

Before you run post-synthesis simulation, generate post-synthesis simulation netlist files. Refer to "Generate Post-Synthesis Simulation Netlist Files" on page 5–7.

The following sections help you perform a post-synthesis simulation for a VHDL design in the Active-HDL software.

If your Active-HDL software comes with precompiled simulation libraries, creating or compiling simulation libraries is not required, and you can skip the following section.

Create Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-synthesis primitives. Depending on the device family you are using, you must create the required simulation libraries and link them to your design correctly.

Execute the following command to create a simulation library:

```
vlib <library_name> ↵
```

For example, the library name for the Stratix III family should be **stratixiii**. For a list of all gate-level timing simulation library files, refer to "Gate-Level Timing Simulation Library Files" on page 5–4.

For example, to create simulation libraries for **stratixiii**, execute:

```
vlib stratixiii ↵
```

Compile Simulation Models into Simulation Libraries

To compile simulation models into simulation libraries, type the following command:

```
vcom -work <simulation_library> \ <Quartus II installation directory> \
/eda/sim_lib/<simulation_library_files> ↵
```

For example, compile the **stratixiii_atoms_components.vhd** and **stratixiii_atoms.vhd** model files into the **stratixiii** library.

Example 5–2 compiles simulation model files into the simulation libraries for **stratixiii**.

Example 5–2.

```
vcom -work stratixiii \
<Quartus II installation directory>/eda/sim_lib/stratixiii_atoms_components.vhd \
<Quartus II installation directory>/eda/sim_lib/stratixiii_atoms.vhd
```

Compile the Testbench and VHDL Output File into the Work Library

The following instructions show how you can compile your testbench ***.vho** files into the work library using the Active-HDL GUI.

Type the following command to create the work library:

```
vlib work ↵
```

Type the following command to compile the testbench and ***.vho** files into the work library:

```
vcom -work work <my_testbench>.vhd <my_vhdl_netlists>.vho ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, perform the following command:

```
vsim -t ps -L <library1> -L <library2> work.<my_testbench>
```

<library1> and *<library2>* are the libraries you compiled previously (for example, **stratixiii and stratixiigx**), which are required to load your design. The gate-level libraries are usually required to perform post-synthesis simulation. If you have multiple libraries, use the `-L` option multiple times with the `vsim` command.

Running the Simulation

Type the following commands to trace the signals and run a simulation:

```
trace <hierarchical signal name> ↵
run <time period> ↵
```

For example, the following commands trace the `DUT/clk1` and `DUT/U1/DATA` signal, and run the simulation for 100 ps.

```
trace DUT/clk1 ↵
trace DUT/U1/DATA ↵
....
run 100 ps ↵
```

When the simulation is finished, type the following command to end the simulation:

```
endsim ↵
```

To view the simulation result in the GUI waveform, perform the following command to open the **wave.asdb** file:

```
open -asdb {<your_design_path> \work\src\wave.asdb} ↵
```

Perform Gate-Level Timing Simulation

Gate-level simulation is a very important step in ensuring that the FPGA's functionality is still correct and meets all required timing requirements after the design was placed and routed. You can create the gate-level netlist in the Quartus II software and use the netlist to perform gate-level timing simulation with the Active-HDL software.

Before you run gate-level timing simulation, generate gate-level timing simulation netlist files. Refer to the instructions in "Generate Gate-Level Timing Simulation Netlist Files" on page 5–8.

The following sections help you perform a gate-level timing simulation for a VHDL design in the Active-HDL software.

If your Active-HDL software comes with precompiled simulation libraries, creating or compiling simulation libraries is not required, and you can skip the Create Simulation Libraries and Compile Simulation Models into Simulation Libraries sections.

Create Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-fitting primitives. Depending on the device family you are using, you must create the required simulation libraries and link them to your design correctly.

Type the following commands to create simulation libraries:

```
vlib <library_name> ↵
```

For example, the library name for the Stratix III family should be **stratixiii**. For a list of all gate-level timing simulation library files, refer to "Gate-Level Timing Simulation Library Files" on page 5–4.

To create simulation libraries for **stratixiii**, type the following command:

```
vlib stratixiii ↵
```

Compile Simulation Models into Simulation Libraries

Perform the following command to compile simulation models into simulation libraries:

```
vcom -work <simulation_library> <Quartus II installation \
directory>/eda/sim_lib/<simulation_library_files> ↵
```

For example, compile the **stratixiii_atoms_components.vhd** and **stratixiii_atoms.vhd** model files into the **stratixiii** library.

Example 5–3 compiles the simulation model files to the simulation libraries for **stratixiii**.

Example 5–3.

```
vcom -work stratixiii \
<Quartus II installation directory>/eda/sim_lib/stratixiii_atoms_components.vhd \
<Quartus II installation directory>/eda/sim_lib/stratixiii_atoms.vhd
```

Compile the Testbench and VHDL Output File into the Work Library

The following instructions show how you can compile your testbench and ***.vho** files into the work library using the Active-HDL software.

Type the following command to create the work library:

```
vlib work ↵
```

Type the following command to compile the testbench and ***.vho** files into the work library:

```
vcom -work work <my_testbench>.vhd <my_vhdl_netlists>.vho ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, type the following command:

```
vsim -t ps -L <library1> -L <library2> -sdftyp <instance path to \
design>= <path to SDO file> work.<my_testbench> ↵
```

The *<library1>* and *<library2>* are the libraries that you compiled previously (for example, **stratixiii and stratixiigx**), which are required to load your design. The gate-level libraries are usually required to perform post-synthesis simulation. If you have multiple libraries, use the `-L` option multiple times with the `vsim` command.

You do not have to set the value (**Minimum**, **Average**, **Maximum**) for the ***.sdo** file because the Quartus II EDA Netlist Writer generates the ***.sdo** file using the same value for the triplet (**Minimum**, **Average**, and **Maximum** timing values).

If your design under test is instantiated in the testbench file under the `i1` label, the *<design instance>* should be “`i1/`” (for example, `/i1=`*<my design>***.sdo**).

Running the Simulation

To trace the signals and run a simulation, type the following commands:

```
trace <hierarchical signal name> ↵
run <time period> ↵
```

For example, the following commands trace the DUT/clk1 and DUT/U1/DATA signal, and run the simulation for 100 ps.

```
trace DUT/clk1 ↵
trace DUT/U1/DATA ↵
....
run 100 ps ↵
```

When the simulation is finished, type the following command to end the simulation:

```
endsim ↵
```

To view the simulation result in the GUI waveform, type the following command to open the **wave.asdb** file:

```
open -asdb {<your_design_path> \work\src\wave.asdb} ↵
```

Simulating the Verilog HDL Designs

The following sections show how to perform RTL functional post-synthesis and gate-level timing simulations in the command-line mode of the Active-HDL software.

Perform RTL Functional Simulation

RTL functional simulation is typically performed to verify the syntax of the code and to check the functionality of the design. The following sections show how to perform RTL functional simulation in the Active-HDL software for Verilog HDL designs.

If your Active-HDL software comes with precompiled simulation libraries, creating or compiling simulation libraries is not required, and you can skip the following section.

Create Simulation Libraries

Simulation libraries are required to simulate a design that contains an Altera primitive, LPM function, or Altera megafunction. Depending on your design, you must create the required simulation libraries and link them to your design correctly.

To create a simulation library, type the following command:

```
vlib <library_name> ↵
```

The library name for Altera megafunctions should be **altera_mf_ver** and the library name for LPM should be **lpm_ver**. For a list of all functional simulation library files, refer to "RTL Functional Simulation Library Files" on page 5–4.

To create simulation libraries for **altera_mf**, **lpm**, and **altera**, type the following commands:

```
vlib altera_mf_ver ↵
vlib lpm_ver ↵
vlib altera_ver ↵
```

Compile Simulation Models into Simulation Libraries

To compile simulation models into simulation libraries, type the following command:

```
vlog –work <simulation_library> <Quartus II installation directory> \
/eda/sim_lib/<simulation_library_files> ↵
```

For example, compile the **altera_mf.v** model files into the **altera_mf_ver** library, and compile the **220model.v** model files into the **lpm_ver** library.

Use Example 5–4 to compile the simulation model files to the simulation libraries for **altera_mf_ver**, **lpm_ver**, and **altera_ver**:

Example 5–4.

```
vlog -work altera_mf_ver <Quartus II installation directory>/eda/sim_lib/altera_mf.v
vlog -work lpm_ver <Quartus II installation directory>/eda/sim_lib/220model.v
vlog -work altera_ver \
   <Quartus II installation directory> /eda/sim_lib/altera_primitives.v
```

Compile the Testbench and Design Files into the Work Library

The following commands show how to compile your testbench and design files into the work library in the Active-HDL command-line.

To create the work library, type the following command:

```
vlib work ↵
```

To compile the testbench and design files into the work library, type the following command:

```
vlog -work work <my_testbench>.v <my_design_files>.v ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, type the following command:

```
vsim -t ps -L <library1> -L <library2> work .<my_testbench>.v ↵
```

<library1> and *<library2>* are the required libraries to load your design. If you have multiple libraries, use the `-L` option multiple times with the `vsim` command.

Running the Simulation

To trace the signals and run a simulation, type the following commands:

```
trace <hierarchical signal name> ↵
run <time period>
```

For example, the following commands trace the `DUT/clk1` and `DUT/U1/DATA` signal, and run the simulation for 100 ps:

```
trace DUT/clk1 ↵
trace DUT/U1/DATA ↵
....
run 100 ps ↵
```

When the simulation is finished, type the following command to end the simulation:

```
endsim ↵
```

To view the simulation result in the GUI waveform, type the following command to open the **wave.asdb** file:

```
open -asdb {<your_design_path>\work\src\wave.asdb} ↵
```

Perform Post-Synthesis Simulation

Perform post-synthesis simulation to verify that functionality of the design is not lost after synthesis. You can create the post-synthesis netlist in the Quartus II software and use the netlist to perform Post-Synthesis simulation with the Active-HDL software.

Before you run post-synthesis simulation, generate the post-synthesis simulation netlist files. Refer to "Generate Post-Synthesis Simulation Netlist Files" on page 5–7.

The following sections help you perform a post-synthesis simulation for a Verilog HDL design in the Active-HDL software.

If your Active-HDL software comes with precompiled simulation libraries, creating or compiling simulation libraries is not required, and you can skip the following section.

Create Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-synthesis primitives. Depending on the device family you are using, you must create the required simulation libraries and link them to your design correctly.

To create a simulation library, type the following command:

```
vlib <library_name> ↵
```

For example, the library name for the Stratix III family should be **stratixiii_ver**. For a list of all gate-level timing simulation library files, refer to "Gate-Level Timing Simulation Library Files" on page 5–4.

As an example, to create simulation libraries for **stratixiii_ver**, type:

```
vlib stratixiii_ver ↵
```

Compile Simulation Models into Simulation Libraries

To compile simulation models into simulation libraries, type the following command:

```
vlog -work <library_name> <Quartus II installation directory> \
   eda/sim_lib/<simulation_library_files> ↵
```

Compile the **stratixiii_atoms.v** model files into the **stratixiii_ver** library.

Example 5–5 compiles the simulation model files into the simulation libraries for **stratixiii**.

Example 5–5.

```
vlog -work stratixiii_ver \
   <Quartus II installation directory>/eda/sim_lib/stratixiii_atoms.v
```

Compile the Testbench and Verilog Output File into the Work Library

The following instructions show how you can compile your testbench and ***.vo** files into the work library using the Active-HDL command line.

To create the work library, type the following command:

```
vlib work ↵
```

To compile the testbench and ***.vo** files into the work library, type the following command:

```
vlog -work work <my_testbench>.v <design_netlists>.vo ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

To load a design, type the following command:

```
vsim -t ps -L <library1> -L <library2> work.<my_testbench>.v ↵
```

<*library1*> and <*library2*> are the libraries that you compiled previously (for example, **stratixiii_ver** and **stratixiigx_ver**), which are required to load the design. The gate-level libraries are usually required to perform post-synthesis simulation. If you have multiple libraries, use the `-L` option multiple times with the `vsim` command.

Running the Simulation

To trace the signals and run a simulation, type the following command:

```
trace <hierarchical signal name> ↵
run <time period> ↵
```

For example, the following commands trace the `DUT/clk1` and `DUT/U1/DATA` signal, and run the simulation for 100 ps:

```
trace DUT/clk1 ↵
trace DUT/U1/DATA ↵
....
run 100 ps ↵
```

When the simulation is finished, type the following command to end the simulation:

```
endsim ↵
```

To view the simulation result in the GUI waveform, type the following command to open the **wave.asdb** file:

```
open -asdb {<your_design_path> \work\src\wave.asdb} ↵
```

Perform Gate-Level Timing Simulation

Gate-Level timing simulation is a very important step in ensuring that the FPGA's functionality is correct and meets all the timing requirements after the design was placed and routed. You can create the gate-level netlist in the Quartus II software and use the netlist to perform gate-level simulation with the Active-HDL software.

Before you run gate-level timing simulation, generate gate-level timing simulation netlist files. Refer to "Generate Gate-Level Timing Simulation Netlist Files" on page 5–8.

The following sections help you perform a gate-level timing simulation for a Verilog HDL design in the Active-HDL software.

If your Active-HDL software comes with precompiled simulation libraries, creating or compiling simulation libraries is not required, and you can skip the following sections.

Create Simulation Libraries

Simulation libraries are required to simulate a design that is mapped to post-fitting primitives. Depending on the device family you are using, you must create the required simulation libraries and link them to your design correctly.

To create a simulation library, type the following command:

```
vlib <library_name> ↵
```

For example, the library name for the Stratix III family should be **stratixiii_ver**. To see all the gate-level timing simulation library files, refer to "Gate-Level Timing Simulation Library Files" on page 5–4.

As an example, to create simulation libraries for **stratixiii**, type the following command:

```
vlib stratixiii_ver ↵
```

Compile Simulation Models into Simulation Libraries

To compile simulation models into a simulation library, type the following command:

```
vlog -work <simulation_library> <Quartus II installation directory> \
/eda/sim_lib/<simulation_library_files> ↵
```

For example, compile the **stratixiii_atoms.v** model files into the **stratixiii_ver** library.

The following is an example to compile simulation model files to the simulation libraries for **stratixiii**:

```
vlog -work stratixiii_ver <Quartus II installation directory> \
/eda/sim_lib/stratixiii_atoms.v ↵
```

Compile the Testbench and Verilog Output File into the Work Library

The following instructions show how you can compile your testbench and ***.vo** files into the work library using the Active-HDL command line.

To create the work library, type the following command:

```
vlib work ↵
```

To compile the testbench and ***.vo** files into the work library, type the following command:

```
vlog -work work <my_testbench>.v <design_netlist>.vo ↵
```

Resolve compile-time errors before proceeding to the next section.

Loading the Design

When simulating in Verilog HDL, the **.sdo** file does not have to be manually specified, because the Quartus II software generates the **.vo** file with the `$sdf_annotate` task that tells Active-HDL how to load the **.sdo** file. Ensure that the **.sdo** file is located in the same simulation output directory.

To load a design, type the following command:

```
vsim -t ps -L <library1> -L <library2> work.<my_testbench>.v ↵
```

<library1> and the *<library2>* are the libraries that you compiled previously (for example, **stratixiii_ver** and **stratixiigx_ver**), which are required to load the design. The gate-level libraries are usually required for performing gate-level simulation. If you have multiple libraries, use the `-L` option multiple times with the `vsim` command.

Running the Simulation

To trace the signals and run a simulation, type the following command:

```
trace <hierarchical signal name> ↵
run <time period> ↵
```

For example, the following commands trace the `DUT/clk1` and `DUT/U1/DATA` signal, and run the simulation for 100 ps:

```
trace DUT/clk1 ↵
trace DUT/U1/DATA ↵
....
run 100 ps ↵
```

When the simulation is finished, type the following command to end the simulation:

```
endsim ↵
```

To view the simulation result in the GUI waveform, perform the following command to open the **wave.asdb** file:

```
open -asdb {<your_design_path> \work\src\wave.asdb} ↵
```

Simulating Designs that Include Transceivers

If your design includes a Stratix IV, Stratix II GX, Stratix GX, Arria II GX, or Arria GX transceiver, you must compile additional library files to perform RTL functional or gate-level timing simulations. The following example shows how to perform simulation on designs that include Stratix GX and Stratix II GX transceivers.

For high-speed simulation, you must select **ps** in the **Resolution** list for your simulator resolutions. If you choose slower than **ps**, the high-speed simulation may fail.

Performing simulation with transceivers in Arria GX or Stratix II GX is very similar. The only requirement is to replace the **stratixiigx_atoms** and **stratixiigx_hssi_atoms** model files with the **arriagx_atoms** and **arriagx_hssi_atoms** model files.

Stratix GX RTL Functional Simulation

To perform an RTL functional simulation of your design that instantiates the ALTGXB megafunction, which enables the gigabit transceiver block on Stratix GX devices, compile the **stratixgx_mf** model file into the **altgxb** library.

The **stratixgx_mf** model file references the **lpm** and **sgate** libraries, so you must create these libraries to perform a simulation.

Performing RTL Functional Simulation for Stratix GX in VHDL

If you are using Active-HDL software that comes with precompiled simulation libraries, compiling the libraries is not necessary. Simulate the design directly by typing the commands in Example 5–6.

Example 5–6.

```
vcom -work <my design>.vhd <my testbench>.vhd
vsim -L lpm -L altera_mf –L sgate -L altgxb work.<my testbench>
```

If you are using Active-HDL software without precompiled simulation libraries, you must compile the necessary libraries before you can simulation the designs. Type the commands in Example 5–7 to compile and simulate the design.

Example 5–7.

```
vcom -work altera_mf altera_mf_components.vhd altera_mf.vhd
vcom -work lpm 220pack.vhd 220model.vhd
vcom -work sgate sgate_pack.vhd sgate.vhd
vcom -work altgxb stratixgx_mf.vhd stratixgx_mf_components.vhd
vcom -work <my design>.vhd <my testbench>.vhd
vsim -L lpm -L altera_mf -L sgate -L altgxb work.<my testbench>
```

Performing Functional Simulation for Stratix GX in Verilog HDL

If you are using Active-HDL software that comes with precompiled simulation libraries, compiling the libraries is not necessary. You can simulate the design directly by typing the command in Example 5–8.

Example 5–8.

```
vlog -work <my design>.v <my testbench>.v
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L altgxb_ver work.<my testbench>
```

If you are using Active-HDL software without precompiled simulation libraries, you must compile the necessary libraries before you can simulate the designs. Type the commands in Example 5–9 to compile and simulate the design.

Example 5–9.

```
vlib work
vlib lpm_ver
vlib altera_mf_ver
vlib sgate_ver
vlib altgxb_ver
vlog -work lpm_ver 220model.v
vlog -work altera_mf_ver altera_mf.v
vlog -work sgate_ver sgate.v
vlog -work altgxb_ver stratixgx_mf.v
vlog -work <my design>.v <my testbench>.v
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L altgxb_ver work.<my testbench>
```

Stratix GX Gate-Level Timing Simulation

Perform a gate-level timing simulation of your design that includes a Stratix GX transceiver by compiling the **stratixgx_atoms** and **stratixgx_hssi_atoms** model files into the **stratixgx** and **stratixgx_gxb** libraries, respectively.

Performing Timing Simulation for Stratix GX in VHDL

If you are using Active-HDL software that comes with precompiled simulation libraries, compiling the libraries is not necessary. You can simulate the design directly by typing the commands in Example 5–10.

Example 5–10.

```
vcom -work <my design>.vho <my testbench>.vhd
vsim -L lpm -L altera_mf -L sgate -L stratixgx -L stratixgx_gxb \
work.<my testbench> -t ps -sdftyp <design instance>=<path to SDO file>.sdo \
+transport_int_delays+transport_path_delays
```

If you are using Active-HDL software without precompiled simulation libraries, you must compile the necessary libraries before you can simulate the designs. Type the commands in Example 5–11 to compile and simulate the design.

Example 5–11.

```
vcom -work lpm 220pack.vhd 220model.vhd
vcom -work altera_mf altera_mf_components.vhd altera_mf.vhd
vcom -work sgate sgate_pack.vhd sgate.vhd
vcom -work stratixgx stratixgx_atoms.vhd stratixgx_components.vhd
vcom -work stratixgx_gxb stratixgx_hssi_atoms.vhd \
stratixgx_hssi_components.vhd
vcom -work <my design>.vho <my testbench>.vhd
vsim -L lpm -L altera_mf -L sgate -L stratixgx -L stratixgx_gxb work. \
<my testbench> -t ps -sdftyp <design instance>=<path to SDO file>.sdo \
+transport_int_delays +transport_path_delays
```

Performing Gate-Level Timing Simulation for Stratix GX in Verilog HDL

If you are using Active-HDL software that comes with precompiled simulation libraries, compiling the libraries is not necessary. You can simulate the design directly by typing the commands in Example 5–12.

Example 5–12.

```
vlog -work <my design>.v <my testbench>.v
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixgx_ver -L stratixgx_gxb_ver \
work.<my testbench> -t ps +transport_int_delays +transport_path_delays
```

If you are using Active-HDL software without precompiled simulation libraries, you must compile the necessary libraries before you can simulate the designs. Type the commands in Example 5–13 to compile and simulate the design.

Example 5–13.

```
vlog -work lpm_ver 220model.v
vlog -work altera_mf_ver altera_mf.v
vlog -work sgate_ver sgate.v
vlog -work stratixgx_ver stratixgx_atoms.v
vlog -work stratixgx_gxb_ver stratixgx_hssi_atoms.v
vlog -work <my design>.vo <my testbench>.v
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixgx_ver -L stratixgx_gxb_ver \
work.<my testbench> -t ps +transport_int_delays +transport_path_delaysr
```

Stratix II GX RTL Functional Simulation

Stratix II GX RTL functional simulation is similar to Arria GX functional simulation. The following example shows only the RTL functional simulation for designs that include transceivers in Stratix II GX. To simulate transceivers in Arria GX, replace the **stratixiigx_hssi** model file with the **arriagx_hssi** model file.

To perform an RTL functional simulation of your design that instantiates the ALT2GXB megafunction, which enables the gigabit transceiver blocks on Stratix II GX devices, you must generate a functional simulation netlist and compile the **stratixiigx_hssi** model file into the **stratixiigx_hssi** library.

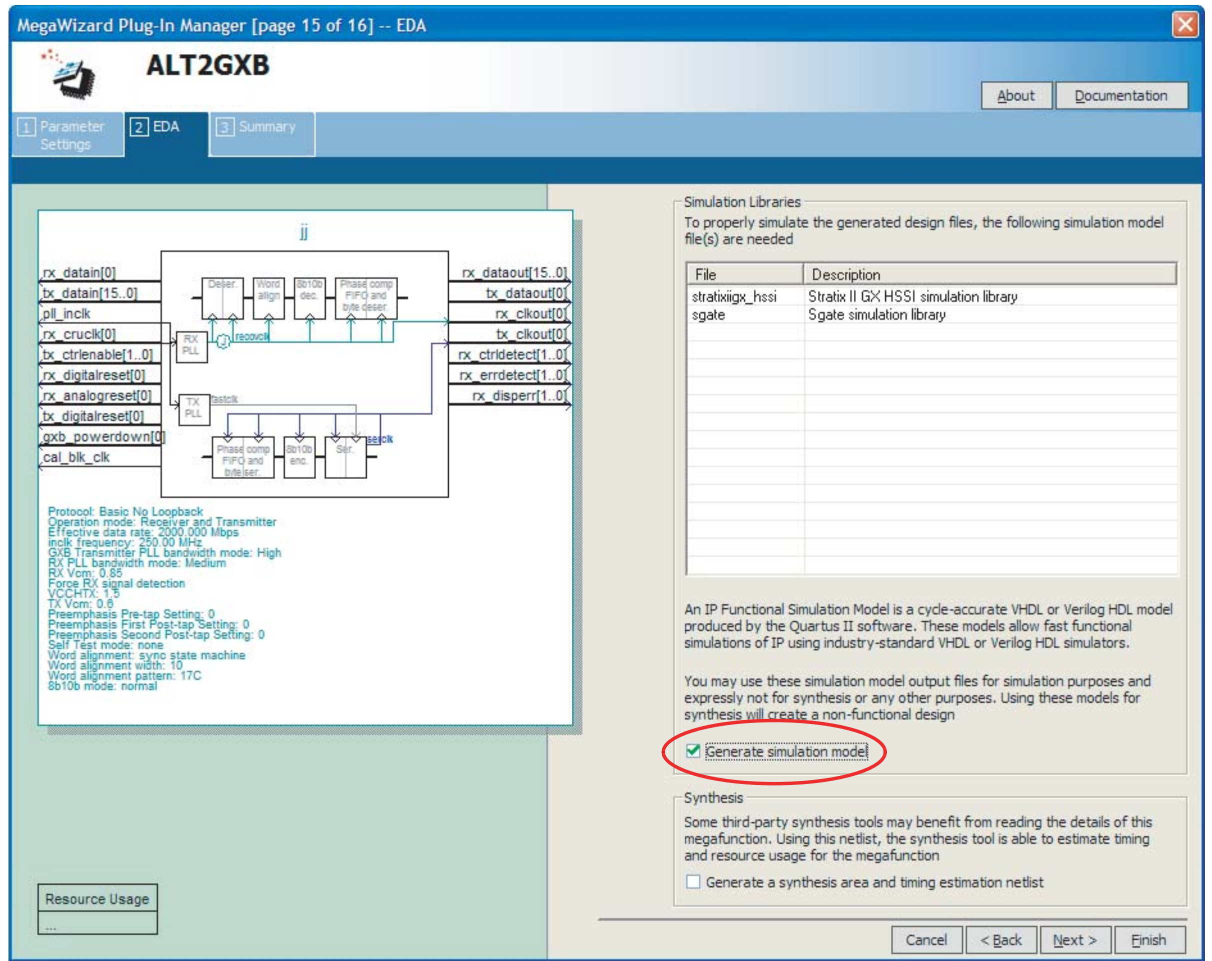
The **stratixgx_hssi_atoms** model file references the **lpm** and **sgate** libraries; you must create these libraries to perform a simulation.

To run the RTL functional simulation, you must generate a functional simulation netlist by turning on **Generate Simulation Model** in the **Simulation Libraries** tab of the ALT2GXB MegaWizard Plug-In Manager (see Figure 5–2).

The <*alt2gxb entity name*>**.vho** or <*alt2gxb module name*>**.vo** is generated in the current project directory.

The Quartus II-generated ALT2GXB functional simulation library file references **stratixiigx_hssi** wysiwyg atoms.

Figure 5–2. ALT2GXB MegaWizard Plug-In Manager



Performing RTL Functional Simulation for Stratix II GX in VHDL

If you are using Active-HDL software that comes with precompiled simulation libraries, compiling the libraries is not necessary. You can simulate the design directly by typing the commands in Example 5–14.

Example 5–14.

```
vcom -work work <alt2gxb entity name>.vho
vcom -work <my design>.vhd <my testbench>.vhd
vsim -L lpm -L altera_mf -L sgate -L stratixgx_hssi work.<my testbench>
```


© March 2009 Altera Corporation

If you are using Active-HDL software without precompiled simulation libraries, you must compile the necessary libraries before you can simulate the designs. Type the commands in Example 5–15 to compile and simulate the design.

Example 5–15.

```
vcom -work lpm 220pack.vhd 220model.vhd
vcom -work altera_mf altera_mf_components.vhd altera_mf.vhd
vcom -work sgate sgate_pack.vhd sgate.vhd
vcom -work stratixiigx_hssi stratixiigx_hssi_components.vhd \
stratixiigx_hssi_atoms.vhd
vcom -work work <alt2gxb entity name>.vho
vcom -work <my design>.vhd <my testbench>.vhd
vsim -L lpm -L altera_mf -L sgate -L stratixgx_hssi work.<my testbench>
```

Performing RTL Functional Simulation for Stratix II GX in Verilog HDL

If you are using Active-HDL software that comes with precompiled simulation libraries, compiling the libraries is not necessary. You can simulate the design directly by typing the commands in Example 5–16.

Example 5–16.

```
vlog -work work <alt2gxb module name>.vo
vlog -work <my design>.v <my testbench>.v
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixgx_hssi_ver \
   work.<my testbench>
```

If you are using Active-HDL software without precompiled simulation libraries, you must compile the necessary libraries before you can simulate the designs. Type the commands in Example 5–17 to compile and simulate the design.

Example 5–17.

```
vlog -work lpm_ver 220model.v
vlog -work altera_mf_ver altera_mf.v
vlog -work sgate_ver sgate.v
vlog -work stratixiigx_hssi_ver stratixiigx_hssi_atoms.v
vlog -work work <alt2gxb module name>.vo
vlog -work <my design>.v <my testbench>.v
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixgx_hssi \
   work.<my testbench>
```

Stratix II GX Gate-Level Timing Simulation

To perform a gate-level timing simulation of your design that includes a Stratix II GX transceiver, you must compile **stratixiigx_atoms** and s**tratixiigx_hssi_atoms** into the **stratixiigx** and **stratixiigx_hssi** libraries, respectively.

Performing Timing Simulation for Stratix II GX in VHDL

If you are using Active-HDL software that comes with precompiled simulation libraries, compiling the libraries is not necessary. You can simulate the design directly by typing the commands in Example 5–18.

Example 5–18.

```
vcom -work <my design>.vho <my testbench>.vhd
vsim -L lpm -L altera_mf -L sgate -L stratixiigx -L stratixiigx_hssi \
work.<my testbench> -t ps -sdftyp <design instance>=<path to SDO file>.sdo \
+transport_int_delays +transport_path_delays
```

If you are using Active-HDL software without precompiled simulation libraries, you must compile the necessary libraries before you can simulate the designs.

Type the commands in Example 5–19 to compile and simulate the design.

Example 5–19.

```
vcom -work lpm 220pack.vhd 220model.vhd
vcom -work altera_mf altera_mf_components.vhd altera_mf.vhd
vcom -work sgate sgate_pack.vhd sgate.vhd
vcom -work stratixiigx stratixiigx_atoms.vhd stratixiigx_components.vhd
vcom -work stratixiigx_hssi stratixiigx_hssi_components.vhd stratixiigx_hssi_atoms.vhd
vcom -work <my design>.vho <my testbench>.vhd
vsim -L lpm -L altera_mf -L sgate -L stratixiigx -L stratixiigx_hssi \
work.<my testbench> -t ps -sdftyp <design instance>=<path to SDO file>.sdo \
+transport_int_delays +transport_path_delays
```

Performing Timing Simulation for Stratix II GX in Verilog HDL

If you are using Active-HDL software that comes with precompiled simulation libraries, compiling the libraries is not necessary. You can simulate the design directly by typing the command in Example 5–20.

Example 5–20.

```
vlog -work <my design>.vo <my testbench>.v
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixiigx_ver -L \
stratixiigx_hssi_ver work.<my design> -t ps +transport_int_delays +transport_path_delays
```

If you are using Active-HDL software without precompiled simulation libraries, you must compile the necessary libraries before you can simulate the designs.

Type the commands in Example 5–21 to compile and simulate the design.

Example 5–21.

```
vlog -work lpm_ver 220model.v
vlog -work altera_mf_ver altera_mf.v
vlog -work sgate_ver sgate.v
vlog -work stratixiigx_ver stratixiigx_atoms.v
vlog -work stratixiigx_hssi_ver stratixiigx_hssi_atoms.v
vlog -work <my design>.vo <my testbench>.v
vsim -L lpm -L altera_mf_ver -L sgate_ver -L stratixiigx_ver -L stratixiigx_hssi_ver \
work.<my testbench> -t ps +transport_int_delays +transport_path_delays
```

Transport Delays

By default, the Active-HDL software filters out all pulses that are shorter than the propagation delay between primitives. Turning on the **transport delay** options in the Active-HDL software prevents the simulation tool from filtering out these pulses. Use the following options to ensure that all signal pulses are visible in the simulation results:

- **+transport_path_delays**

  Use this option when the pulses in your simulation may be shorter than the delay within a gate-level primitive.

- **+transport_int_delays**

  Use this option when the pulses in your simulation may be shorter than the interconnect delay between gate-level primitives.

The **+transport_path_delays** and **+transport_int_delays** options are also used by default in the NativeLink feature for gate-level timing simulation.

For more information about either of these options, refer to the Active-HDL online documentation installed with the Active-HDL software.

Example 5–22 shows an Active-HDL software command in command-line syntax to perform a gate-level timing simulation with the device family library.

Example 5–22.

```
vsim -t 1ps -L stratixii -sdftyp /i1=filtref_vhd.sdo \
work.filtref_vhd_vec_tst +transport_int_delays +transport_path_delays
```

Using the NativeLink Feature in Active-HDL Software

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools and allows you to run Active-HDL within the Quartus II software.

Setting Up NativeLink

To run the Active-HDL software automatically from the Quartus II software using the NativeLink feature, you must specify the path to your simulation tool by performing the following steps:

1. On the Tools menu, click **Options**. The **Options** dialog box appears.
2. In the **Category** list, select **EDA Tool Options**.
3. Double-click the entry under **Location of executable** beside the name of your EDA Tool.
4. Type or browse to the directory containing the executables of your EDA tool.

For Active-HDL, the executable files are stored in the **bin** directory (for example, **C:\Program Files\Aldec\Active-HDL X.X\BIN**).

You can also specify the path to the simulator's executables by using the `set_user_option` TCL command:

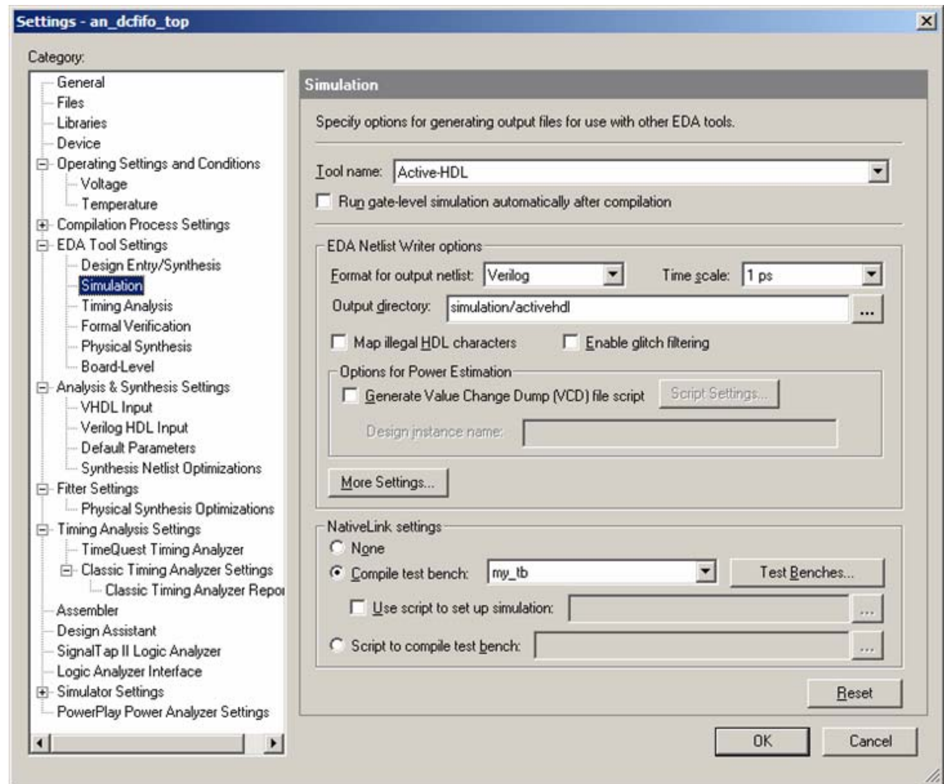
```
set_user_option -name EDA_TOOL_PATH_ACTIVEHDL <path to executables> ↵
```

Performing an RTL Simulation Using NativeLink

To run an RTL functional simulation with the Active-HDL software in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears (Figure 5–3).

Figure 5–3. Simulation Page in the Settings Dialog Box



3. In the **Tool name** list, select **Active-HDL**.
4. If your design is written entirely in Verilog HDL or VHDL, the NativeLink feature automatically chooses the correct language and Altera simulation libraries. If your design is written with mixed languages, the NativeLink feature uses the default language specified in the **Format for output netlist** list. To change the default language when there is a mixed language design, under **EDA Netlist Writer options**, in the **Format for output netlist** list, select **VHDL** or **Verilog**. Table 5–7 shows the design languages for output netlists and simulation models.

Table 5–7. NativeLink Design Languages

Device File	Format for Output Netlist	Simulation Models Used
Verilog HDL	Any	Verilog HDL
VHDL	Any	VHDL
Mixed	Verilog HDL	Verilog HDL
Mixed	VHDL	VHDL

For mixed language simulation, choose the same language that was used to generate your megafunctions to ensure correct parameter passing between the megafunctions and the Altera libraries. For example, if your ALTSYNCRAM megafunction was generated in VHDL, choose VHDL as the format for the output netlist.

When creating mixed language designs, it is important to be aware of the following:

- Seamless passing of parameters when a VHDL entity is instantiated in Verilog HDL designs is not allowed.
- The use of Verilog User Defined Primitives (UDPs) to be instantiated in VHDL designs is not allowed.

5. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.

For more information about setting up a testbench with NativeLink, refer to “Setting Up a Testbench” on page 5–49.

6. Click **OK**.
7. On the Processing menu, point to **Start** and click **Start Analysis and Elaboration** to perform an analysis and elaboration. This command collects all your file name information and builds your design hierarchy in preparation for simulation.
8. On the Tools menu, point to **Run EDA Simulation Tool** and click **EDA RTL Simulation** to automatically run the Active-HDL software, compile all necessary design files, and complete a simulation.
9. If you want to generate only the Tcl script without using the GUI after the NativeLink process, perform the following steps:
   a. On the **Simulation** page, click **More NativeLink Settings**. The **More NativeLink Settings** dialog box appears.
   b. Under **Existing option settings**, click **Generate third-party EDA tool command scripts without running the EDA tool**.
   c. Under **Setting**, click **On**.

   If you turn this option on and run NativeLink, the Tcl file for the simulation process is generated without showing the results in the GUI. You can then run the simulation by typing the following command:

```
do <design_name>.do ↵
```

Performing a Gate-Level Timing Simulation Using NativeLink

To run a gate-level timing simulation with the Active-HDL software in the Quartus II software, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears (Figure 5–3).
3. In the **Tool name** list, select **Active-HDL**.
4. Under **EDA Netlist Writer options**, in the **Format for output netlist** list, choose **VHDL** or **Verilog**. You can also modify where you want the post-synthesis netlist generated by editing or browsing to a directory in the **Output directory** box.
5. To run a gate-level timing simulation after each full compilation, turn on **Run Gate Level Simulation automatically after compilation**.
6. If you have testbench files or macro scripts, enter the information under **NativeLink settings**.
7. Click **OK**.
8. On the Processing menu, point to **Start** and click **Start EDA Netlist Writer** to generate a simulation netlist of your design.

   If you have run full compilation after you set the EDA Tool Settings, the Start EDA Netlist Writer is not required.

9. On the Tools menu, point to **Run EDA Simulation Tool** and click **EDA Gate Level Simulation** to automatically run Active-HDL, compile all necessary design files, and complete a simulation.

   If multi-corner Timing Analysis is performed, a dialog box appears. Select the timing corner and click **Run**.

   The Active-HDL Macro File (***.do**) is generated in the <*project_directory*>**\simulation\activehdl** directory while running NativeLink. You can perform a simulation with the **.do** file directly from Active-HDL when you rerun a simulation without using NativeLink. To perform the simulation directly without NativeLink, type the following command in the Active-HDL console: `do` <*generated_do_file*>`.do`.

10. If you want to run Active-HDL in command-line mode when running it automatically after full compilation, you can perform the following steps:

    a. On the Simulation page, click **More NativeLink Settings**. The **More NativeLink Settings** dialog box appears.

    b. Under **Existing option settings**, click **Launch third-party EDA tool in command-line mode**.

    c. Under Setting, click **On**.

    d. Click **OK**.

11. To generate only the Tcl script without using the GUI after the NativeLink process, perform the following steps:
    a. On the **Simulation** Page, click **More NativeLink Settings**. The **More NativeLink Settings** dialog box appears.
    b. Under **Existing option settings**, click **Generate third-party EDA tool command scripts without running the EDA tool.**
    c. Under **Setting**, click **On**.

    If you turn this option on and run NativeLink, the Tcl file for the simulation process is generated without showing the results in the GUI. You can then run the simulation by typing the following command:

    ```
    do <design_name>.do ↵
    ```

Setting Up a Testbench

You can use NativeLink to compile your design files and testbench files, and run an EDA simulation tool to automatically perform a simulation.

To set up NativeLink for simulation, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the "+" icon to expand **EDA Tool Settings** and select **Simulation**. The **Simulation** page appears.
3. Under **NativeLink settings**, select **None**, **Compile test bench**, or **Script to compile test bench** (see Table 5–8).

Table 5–8. NativeLink Settings

Settings	Description
None	Compile simulation models and design files.
Compile test bench	NativeLink compiles simulation models, design files, testbench files, and starts simulation.
Script to compile test bench	NativeLink compiles the simulation models and design files. The script you provide is sourced after design files compile. Use this option when you want to create your own script to compile your testbench and perform simulation.

4. If you select **Compile test bench**, select your testbench setup from the **Compile testbench** list. You can use different testbench setups to specify different test scenarios. If there are no testbench setups entered, create a testbench setup by performing the following steps:

   a. Click **Test Benches**. The **Test Benches** dialog box appears.

   b. Click **New**. The **New Test Bench Settings** dialog box appears.

   c. In the **Test Bench name** dialog box, type in the testbench setup name that identifies the different testbench setups.

   d. In the **Test level module** box, type in the top-level testbench entity name. For example, for a Quartus II generated VHDL testbench, type `<Vector Waveform File name>_vhd_vec_tst`.

   e. In the **Instance** box, type in the full instance path to the top level of your FPGA design. For example, for a Quartus II generated VHDL testbench, type in `i1`.

   f. Under **Simulation period**, select **Run simulation until all vector stimuli are used** or specify the end time of the simulation.

   g. Under **Test bench files**, browse and add all your testbench files in the **File name** box. Use the **Up** and **Down** button to reorder your files. The script used by NativeLink compiles the files in order from top to bottom.

You can also specify the library name and the HDL version to compile the testbench file. NativeLink compiles the testbench into a library name using the specified HDL version.

   h. Click **OK**.

   i. In the **Test Benches** dialog box, click **OK**.

Creating a Testbench

In the Quartus II software, you can create a Verilog HDL or VHDL testbench from a Vector Waveform File. The generated testbench includes the behavior of the input stimulus and applies it to your instantiated top-level FPGA design.

To create a testbench, perform the following steps:

1. On the File menu, click **Open**. The **Open** dialog box appears.
2. Click the **Files of type** arrow and select **Waveform/Vector Files**. Select your Vector Waveform File.
3. Click **Open**.
4. On the File menu, click **Export**. The **Export** dialog box appears.
5. Click the **Save as type** arrow and select **VHDL Test Bench File** (***.vht**) or **Verilog Test Bench File** (***.vt**).
6. You can turn on **Add self-checking code to file** to check your simulation results against your Vector Waveform File.
7. Click **Export**. Your VHDL or Verilog HDL testbench file is generated in your project directory.

Generate Simulation Script from EDA Netlist Writer

In the Quartus II software version 9.0 and later, you can generate the simulation script (including the **.do** file, Tcl script, and option file) when you run the EDA Netlist Writer. You can use the simulation script to run your simulation in the preferred simulator in stand-alone mode.

To generate a simulation script with the EDA Netlist Writer, perform the following steps:

1. On the **Simulation** page, click **More EDA Netlist Writer Settings**. The **More EDA Netlist Writer Settings** dialog box appears.
2. Under Existing option settings, click **Generate third-party EDA tool command script for RTL simulation** (if you want to generate an RTL simulation script) or click **Generate third-party EDA tool command script for Gate-Level simulation** (if you want to generate a Gate-Level simulation script).
3. Under Setting, click **On**.
4. Click **OK**.
5. Follow the steps in "Generate Post-Synthesis Simulation Netlist Files" on page 5–7.

In the command-line, to generate the simulation script with the EDA Netlist Writer, type the following command:

```
quartus_eda --simulation --tools=<Your decided tool> --format=<vhdl or verilog>
    --gen_script=<rtl or gate_level> <Project Name> -c <Revision name> ↵
```

Generating VCD Files for PowerPlay

To generate a VCD file for PowerPlay, you must first generate a VCD script in the Quartus II software and run the VCD script from the Active-HDL software to generate a VCD file. This VCD file can then be used by PowerPlay for power estimation. The following instructions show you how to generate a VCD file.

Perform the following steps to generate VCD Scripts in the Quartus II software:

1. In the Quartus II software, on the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulator Settings**.
3. On the **Simulator Settings** page, in the **Tool Name** list, choose the appropriate third-party simulation tool (that is, **Active-HDL**), and turn on the **Generate Value Change Dump File Script** option.
4. To generate the VCD Script file, perform a full compilation.

Perform the following steps to generate a VCD file in the Active-HDL software:

1. In the Active-HDL software, before simulating your design, source the **<*revision_name*>_dump_all_vcd_nodes.tcl** script. To source the TCL script, run the following command before running the **vsim** command:

   ```
   source <revision_name>_dump_all_vcd_nodes.tcl ↵
   ```

2. Continue to run the simulation until the simulation is completed. Exit the Active-HDL software. If you do not exit the software, the Active-HDL software may end the writing process of the VCD files improperly, resulting in a corrupted VCD file.

For more details about using the VCD file for power estimation, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Scripting Support

You can run procedures and create settings described in this chapter in a Tcl script. You can also run some procedures at the command-line prompt.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For detailed information about scripting command options, refer to the **Qhelp** command line and Tcl API help browser.

To start the Qhelp help browser, type the following command:

```
quartus_sh -qhelp ↵
```

Generating a Post-Synthesis Simulation Netlist for Active-HDL

You can use the Quartus II software to generate a post-synthesis simulation netlist with Tcl commands or with a command at the command-line prompt. The following examples assume you are selecting Active-HDL (Verilog HDL output from the Quartus II software).

Tcl Commands

Use the following Tcl commands to set the output format to Verilog HDL, the simulation tool to Active-HDL for Verilog HDL, and to generate a functional netlist:

```
set_global_assignment-name EDA_SIMULATION_TOOL "Active-HDL(Verilog)" ↵
set_global_assignment-name EDA_GENERATE_FUNCTIONAL_NETLIST ON ↵
```

Command Prompt

Use the following command to generate a simulation output file for the Active-HDL software. Specify VHDL or Verilog HDL for the format:

```
quartus_eda <project name> --simulation=on --format=<format> \
   --tool=activehdl --functional ↵
```

Generating a Gate-Level Timing Simulation Netlist for Active-HDL

You can use the Quartus II software to generate a gate-level timing simulation netlist with Tcl commands or with a command at the command prompt.

Tcl Commands

Use one of the following Tcl commands:

```
set_global_assignment -name EDA_SIMULATION_TOOL "Active-HDL (Verilog)" ↵
set_global_assignment -name EDA_SIMULATION_TOOL "Active-HDL (VHDL)" ↵
```

Command Line

Generate a simulation output file for the Active-HDL software by specifying VHDL or Verilog HDL for the format by typing the following command at the command prompt:

```
quartus_eda <project name> --simulation=on --format=<format> \
    --tool=activehdl ↵
```

Conclusion

Using the Active-HDL simulation software within the Altera FPGA design flow allows you to easily and accurately perform RTL functional simulations, post-synthesis simulations, and gate-level timing simulations on your designs. Proper verification of designs at the functional, post-synthesis, and post place-and-route stages using the Active-HDL software helps ensure design functionality success and, ultimately, a quick time-to-market.

Referenced Documents

This chapter references the following documents:

- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Installation and Licensing for UNIX and Linux Workstations manual*
- *Quartus II Installation and Licensing for Windows manual*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 5–9 shows the revision history for this chapter.

Table 5–9. Document Revision History

Date / Revision	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Removed "Compile Libraries Using the Altera Simulation Library Compiler". ■ Added "Compile Libraries Using the EDA Simulation Library Compiler" on page 5–10. ■ Added "Generate Simulation Script from EDA Netlist Writer" on page 5–51. ■ Minor editorial updates.	Updated for the Quartus II software version 9.0 release.
November 2008 v.8.1.0	Added the following sections: ■ "Compile Libraries Using the Altera Simulation Library Compiler" on page 5–10 ■ Added steps to the procedure "Performing an RTL Simulation Using NativeLink" on page 5–45 for using the Altera Simulation Library Compilation ■ Added steps to the procedure "Performing a Gate-Level Timing Simulation Using NativeLink" on page 5–47 for using the Altera Simulation Library Compilation ■ Minor editorial updates ■ Updated entire chapter using 8½" × 11" chapter template	Updated for the Quartus II software version 8.1 release.
May 2008 v.8.0.0	Initial Release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

6. Simulating Altera IP in Third-Party Simulation Tools

QII53014-9.0.0

Introduction

The capacity and complexity of Altera® FPGAs continues to increase as the need for intellectual property (IP) becomes increasingly critical. Using IP megafunctions reduces the design and verification time, allowing you to focus on design customization. Altera and the Altera Megafunction Partners Program (AMPP℠) offer a broad portfolio of IP megafunctions optimized for Altera FPGAs. Through parameterization, these reusable blocks of IP can be customized to meet your design requirements.

Even when the IP source code is encrypted or otherwise restricted, Altera's Quartus® II software allows you to easily simulate designs that contain Altera IP. With the Quartus II software, you can custom configure IP designs, then generate a VHDL or Verilog HDL functional simulation model to use with your choice of simulation tools.

This chapter provides an overview of the process for instantiating the IP megafunctions in your design and simulating its functional simulation model in an Altera-supported, third-party simulation tool. In this chapter, IP megafunctions refer to Altera megafunctions, IP MegaCore® functions, and IP AMPP megafunctions.

All IP MegaCore functions come with IP functional simulation (IPFS) models to support functional simulation. Some Altera megafunctions and some AMPP megafunctions also require IPFS models for functional simulation.

An IPFS model is a cycle-accurate VHDL or Verilog HDL model produced by the Quartus II software. The model allows for fast functional simulation of IP using industry-standard VHDL and Verilog HDL simulators.

Use these models only for simulation. Do not use them for synthesis or any other purpose. Using these models for synthesis creates a nonfunctional design.

This chapter discusses the following topics:

- "IP Functional Simulation Flow"
- "Instantiate the IP in Your Design" on page 6–3
- "Perform Simulation" on page 6–3
- "Design Language Examples" on page 6–9

IP Functional Simulation Flow

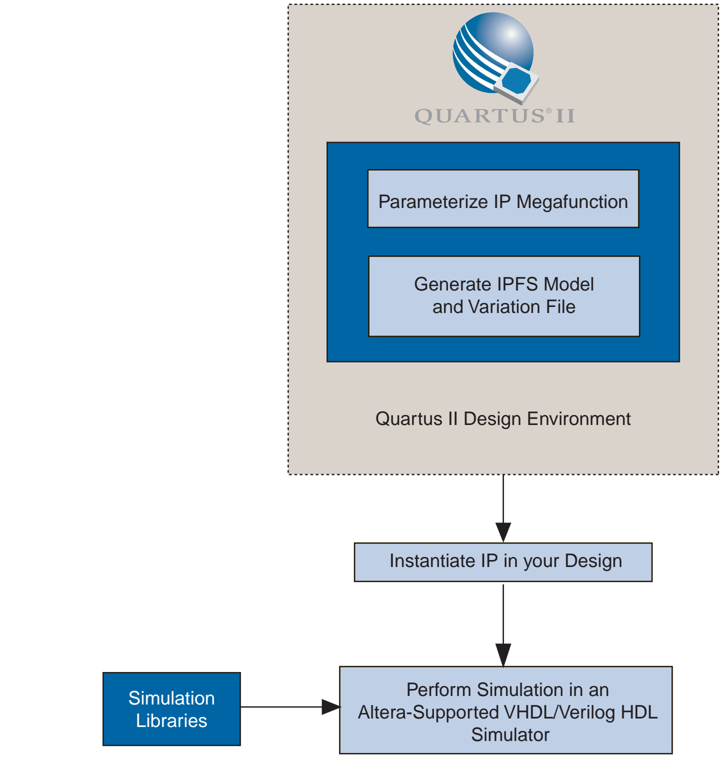
The IP megafunction's MegaWizard™ interface allows you to quickly and easily view documentation, specify parameters, generate an IPFS model, and output the files necessary to integrate a parameterized IP megafunction into your design. Within the Quartus II software, the MegaWizard Plug-In Manager can be used to select and parameterize your choice of IP megafunctions. The Quartus II software generates an

IP megafunction's variation file that is included in your Quartus II project. For IP megafunctions that require IPFS models, the Quartus II software can also generate a Verilog Output File (**.vo**) or VHDL Output File (**.vho**) that contains a Register Transfer Level (RTL) IPFS model after you have parameterized the megafunction. IPFS models are written to the Quartus II project directory.

Most Altera megafunctions and IP MegaCore functions support functional simulation in Verilog and VHDL for all Altera supported third-party simulators. Simulation libraries are required to simulate IP megafunctions. Refer to Table 6–2 on page 6–8 for a subset of simulation libraries supplied with the Quartus II software.

Figure 6–1 shows a typical simulation flow for Altera IP with third-party simulators.

Figure 6–1. IPFS Model Design Flow



Verilog HDL and VHDL IPFS Models

Some IP megafunctions require IPFS models to support functional simulation. These IPFS models are generated in RTL HDL. These RTL models in Verilog HDL or VHDL format differ from the low-level synthesized netlist in Verilog HDL or VHDL format generated by the Quartus II software for post-synthesis or post place-and-route simulations. The IPFS models generated by the Quartus II software are much faster than the low-level post-synthesis or post place-and-route netlists of your design because they are mapped to higher-level primitives such as adders, multipliers, and multiplexers. These IPFS models can be simulated together with the rest of your design in any Altera-supported simulator. Altera recommends that you generate IPFS models in the same hardware language as the IP megafunction's variation file hardware language.

You can use an IPFS model for simulation only. Do not use it for synthesis or any other purpose. Attempting to synthesize an IPFS model will result in a nonfunctional design.

Generating an IPFS model for Altera MegaCore functions does not require a license. However, generating an IPFS model for AMPP megafunctions may require a license. For more information about licensing requirements, contact the IP megafunction vendor.

For details about how to parameterize and generate an IP, refer to the applicable IP user guide.

Instantiate the IP in Your Design

For each IP megafunction in your design, you must instantiate the corresponding entity or module in your design. Each IP megafunction entity or module name is defined in its Quartus II generated megafunction variation file. After instantiating the IP megafunction in your design, you do not need to edit your design for synthesis or simulation.

To synthesize your design using the Quartus II software, add the Quartus II-generated Verilog HDL or VHDL variation file to your Quartus II project. When you create new variation files for a Quartus II project, they are added to the current open project when the megafunction is generated.

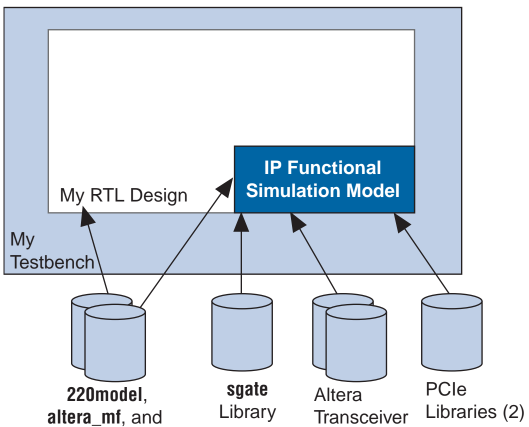
To synthesize your design using a third-party EDA tool, add the Quartus II-generated CMP file (<*megafunction variation*>**.cmp**) for your VHDL design or the Verilog HDL black box file (<*megafunction variation*>**_bb.v**) for your Verilog HDL design to your third-party synthesis project.

For more information about synthesis and compilation with the Quartus II software, refer to the applicable chapters in *Volume 1: Design Synthesis* of the *Quartus II Handbook*.

Perform Simulation

To perform simulation, in addition to adding your design files and testbench files, you also have to add the IP megafunction'is variation file or IPFS model to your simulation project. If the IP megafunction does not require an IPFS model for simulation, add the megafunction's variation file to your simulation project. If the IP megafunction you are simulating requires an IPFS model, add the IPFS model to your simulation project. Your simulation project also requireS Altera-supplied libraries for successful simulation. Figure 6–2 shows how the Altera libraries are used in IP functional simulation.

Figure 6–2. IPFS Library Usage



Notes to Figure 6–2:

(1) The IP that uses the transceiver requires the Altera transceiver libraries for simulation. Refer to Table 6–2 for all the Altera transceiver libraries.

(2) An IP that uses the PCI Express (PCIe) hard core in a Stratix IV device requires the PCIe libraries. Refer to Table 6–2 for all the PCIe transceiver libraries.

The Quartus II software contains all the libraries required for setting up and running a successful simulation of Altera IP. You can use the Quartus II NativeLink feature to set up your simulation if the IP megafunction you are using supports Quartus II NativeLink. Refer to the applicable IP megafunction user guide to determine if NativeLink is supported. Alternatively, you can simulate Altera IP with third-party simulators directly.

Simulating Altera IP Using the Quartus II NativeLink Feature

The Quartus II NativeLink feature eases the task of setting up and running a simulation. The NativeLink feature lets you launch the third-party simulator to perform simulation from within the Quartus II software. The NativeLink feature automates the compilation and simulation of testbenches.

The following list briefly describes the steps to simulate IP megafunctions with third-party simulators using the Quartus II NativeLink feature. Each of these steps is described in more detail in the sections that follow.

1. Select the Third-Party Simulation Tool.
2. Specify the Path for the Third-Party Simulator.
3. Specify the Testbench Settings.
4. Analyze and Elaborate the Quartus II Project.
5. Run RTL Functional Simulation.

Set up a Quartus II Project

To simulate IP megafunctions with the Quartus II NativeLink feature, you must open an existing project or create a new project in the Quartus II software. You can create and parameterize the IP you want to use in your design using the MegaWizard Plug-In Manager within the Quartus II software. Altera IP megafunction variation files are added to your Quartus II project when you create and parameterize the IP. You can also add any other required design files to your Quartus II project. If you are using the Quartus II NativeLink feature and your Quartus II project contains IP megafunctions that require IPFS models for simulation, you do not have to manually add the IPFS models to the Quartus II project for these IP megafunctions. When the Quartus II NativeLink feature launches the third-party simulator tool and starts the simulation, it automatically adds the IPFS model files required for simulation as long as they are present in the Quartus II project directory.

Select the Third-Party Simulation Tool

You can select the third-party simulation tool on the **Simulation** page in the **Settings** dialog box.

Table 6–1 lists the third-party simulators supported by the Quartus II NativeLink feature.

Table 6–1. Third-Party Simulator Support with the Quartus II NativeLink Feature

Third-Party Simulator	Can be Launched from Quartus II Software *(1)*	Testbench Support	Mixed Design (Verilog HDL and VHDL)
ModelSim® PE/SE	Yes	Yes	Yes
ModelSim-Altera Edition	Yes	Yes	No
Synopsys VCS	Yes	Yes	No
Synopsys VCS-MX	Yes	Yes	Yes
Cadence NC-Sim	Yes	Yes	Yes
Aldec Active-HDL	Yes	Yes	Yes

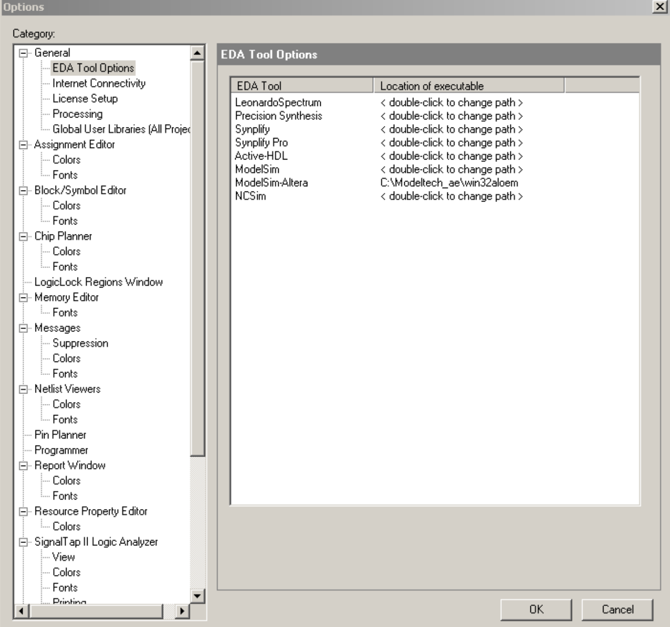
Note to Table 6–1:

(1) The Quartus II software must be running on the same platform as the third-party simulator to launch the simulator tool.

Specify the Path for the Third-Party Simulator

To launch the third-party simulation tool, the absolute path for the selected simulator must be provided on the **Options** page under the Tools menu. See Figure 6–3. Double-click the **Location of executable** field to change or specify the absolute path.

Figure 6–3. Specifying the Simulator Path



Specify the Testbench Settings

Specify the applicable testbench settings as follows:

1. Under **NativeLink settings** in the **Settings** dialog box (Figure 6–3), select the **Compile test bench** radio button and click **Test Benches** to display the **Test Benches** dialog box.
2. Click **New** to display the **New Test Bench Settings** dialog box.
3. In the **New Test Bench Settings** dialog box, set the appropriate fields with the names for the testbenches.

For specific instructions about specifying testbench settings for your MegaCore function, refer to your MegaCore function user guide.

4. After specifying the testbench files, close the **New Test Bench Settings**, **Test Benches**, and **Settings** dialog boxes.

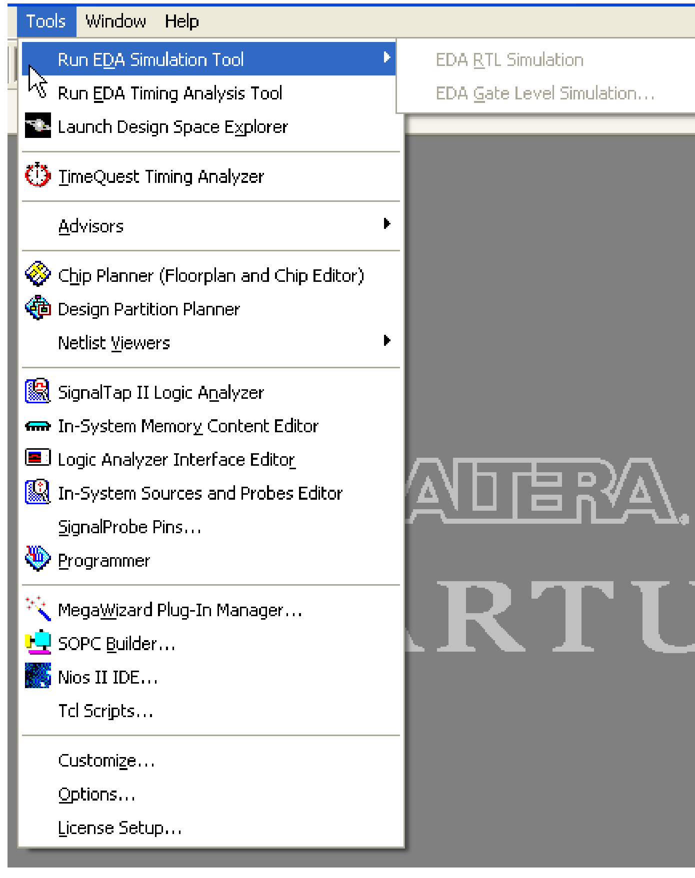
Analyze and Elaborate the Quartus II Project

Before starting the simulation using the NativeLink feature, make sure that each IP megafunction's variation files are included in your design project. On the Quartus II Processing menu, point to **Start**, then click **Start Analysis & Elaboration**.

Run RTL Functional Simulation

After the design is analyzed and elaborated, you can start the simulation by clicking **Run EDA Simulation Tool** from the Tools menu. See Figure 6–4. During RTL functional simulation, the IPFS models are compiled and used by the simulator.

Figure 6–4. Running Functional Simulation for IP using NativeLink



Simulating Altera IP Without the Quartus II NativeLink Feature

You can also simulate Altera IP directly with third-party simulators. If your design instantiates an IP megafunction, add its variation file to your simulation project. If the IP megafunction requires IPFS model files, **do not** add the megafunction's variation file to your simulation project. Rather, add its IPFS model files (either Verilog HDL or VHDL) to your simulation project. The IPFS model generated by the Quartus II software instantiates high-level primitives such as adders, multipliers, and multiplexers, as well as the library of parameterized modules (LPM) functions and Altera megafunctions.

To properly compile, load, and simulate the IP megafunctions, you must first compile the following libraries in your simulation tool:

- **sgate**—includes the definition of the high-level primitives (needed for IPFS models)
- **altera_mf**—includes the definition of Altera megafunctions
- **altera_primitives**—includes the definition of Altera primitives
- **220model**—includes the definition of LPM functions

- **Altera transceiver**—includes the definition of all Altera transceiver megafunctions. If you use IP with the transceiver block, you must compile these libraries, which are device dependent.
- **PCIe**—includes the definition of PCIe hard core megafunctions for Stratix IV. If you use IP with the PCIe hard core for Stratix IV, you must compile these libraries.

You can use these library files with any Altera-supported simulation tool. If you are using the ModelSim-Altera software, the libraries are precompiled and mapped.

To simulate a design containing a Nios® processor or Avalon® peripherals, refer to *AN 189: Simulating Nios Embedded Processor Designs*.

Table 6–2 lists the simulation library files, where <*path*> is the directory where the Quartus II software is installed.

Table 6–2. Simulation Library Files (Part 1 of 2)

Location	HDL Language	Description
<*path*>**/eda/sim_lib/sgate.v**	Verilog HDL	Libraries that contain simulation models for IP functional models
<*path*>**/eda/sim_lib/sgate.vhd**	VHDL	
<*path*>**/eda/sim_lib/sgate_pack.vhd**	VHDL	Libraries that contain VHDL component declarations for the **sgate.vhd** library
<*path*>**/eda/sim_lib/220model.v**	Verilog HDL	Libraries that contain simulation models for the Altera LPM version 2.2.0
<*path*>**/eda/sim_lib/220model.vhd**	VHDL	
<*path*>**/eda/sim_lib/220pack.vhd**	VHDL	Libraries that contain VHDL component declarations for the **220model.vhd** library
<*path*>**/eda/sim_lib/altera_mf.v**	Verilog HDL	Libraries that contain simulation models for Altera-specific megafunctions
<*path*>**/eda/sim_lib/altera_mf.vhd**	VHDL	
<*path*>**/eda/sim_lib/altera_primitives.v**	Verilog HDL	Libraries that contain simulation models for Altera primitives
<*path*>**/eda/sim_lib/altera_primitives.vhd**	VHDL	
<*path*>**/eda/sim_lib/altera_mf_components.vhd**	VHDL	Libraries that contain VHDL component declarations for the **altera_mf.vhd** library
<*path*>**/eda/sim_lib/stratixgx_mf.v**	Verilog HDL	Libraries that contain simulation models for Stratix® GX transceivers
<*path*>**/eda/sim_lib/stratixgx_mf.vhd**	VHDL	
<*path*>**/eda/sim_lib/stratixgx_mf_components.vhd**	VHDL	Libraries that contain VHDL component declarations for the **stratixgx_mf.vhd** library
<*path*>**/eda/sim_lib/stratixiigx_hssi_atoms.v**	Verilog HDL	Libraries that contain simulation models for Stratix GX transceivers
<*path*>**/eda/sim_lib/stratixiigx_hssi_atoms.vhd**	VHDL	
<*path*>**/eda/sim_lib/stratixiigx_hssi_components.vhd**	VHDL	Libraries that contain VHDL component declarations for the **stratixgx_mf.vhd** library
<*path*>**/eda/sim_lib/arriagx_hssi_atoms.v**	Verilog HDL	Libraries that contain simulation models for Arria® GX transceivers
<*path*>**/eda/sim_lib/arriagx_hssi_atoms.vhd**	VHDL	
<*path*>**/eda/sim_lib/arriagx_hssi_components.vhd**	VHDL	Libraries that contain VHDL component declarations for the **arriagx_mf.vhd** library
<*path*>**/eda/sim_lib/stratixiv_hssi_atoms.v**	Verilog HDL	Libraries that contain simulation models for Stratix IV transceivers
<*path*>**/eda/sim_lib/stratixiv_hssi_atoms.vhd**	VHDL	

Table 6–2. Simulation Library Files (Part 2 of 2)

Location	HDL Language	Description
<path>/**eda/sim_lib/stratixiv_hssi_components.vhd**	VHDL	Libraries that contain VHDL component declarations for the **stratixiv_mf.vhd** library
<path>/**eda/sim_lib/stratixiv_pcie_hip_atoms.v**	Verilog HDL	Libraries that contain simulation models for Stratix IV PCI Express
<path>/**eda/sim_lib/stratixiv_pcie_hip_atoms.vhd**	VHDL	
<path>/**eda/sim_lib/stratixiv_pcie_hip_components.vhd**	VHDL	Libraries that contain VHDL component declarations for the **stratix_iv_pcie_hip_atoms.vhd**

Design Language Examples

The following design language examples explain how to simulate IP megafunctions directly with third-party simulator tools. These design examples describe simulation with:

- ModelSim Verilog
- ModelSim VHDL
- NC-VHDL
- VCS

Verilog HDL Example: Simulating the IPFS Model in the ModelSim Software

The following example shows the process of simulating a Verilog HDL-based megafunction. The example assumes that the megafunction variation and the IPFS model are generated.

1. Create a ModelSim project by performing the following steps:
   a. In the ModelSim software, on the File menu, point to **New** and click **Project**. The **Create Project** dialog box appears.
   b. Specify the name of your simulation project.
   c. Specify the desired location for your simulation project.
   d. Specify the default library name and click **OK**.
   e. Add relevant files to your simulation project:
   - Your design files
   - The IPFS model generated by the Quartus II software (if you are using the ModelSim-Altera software, skip to step 5)
   - The **sgate.v**, **220model.v**, and **altera_mf.v** library files
   - The transceiver library files in Verilog HDL, if you use IP with transceivers. Transceiver libraries are family independent. Refer to Table 6–2 for more information.
   - The PCIe library files in Verilog HDL, if you use IP with the PCIe hard core for Stratix IV devices. Refer to Table 6–2 for more information.

2. Create the required simulation libraries by typing the following commands at the ModelSim prompt:

   ```
   vlib sgate ↵
   vlib lpm ↵
   vlib altera_mf ↵
   ```

3. Map to the required simulation libraries by typing the following commands at the ModelSim prompt:

   ```
   vmap sgate sgate ↵
   vmap lpm lpm ↵
   vmap altera_mf altera_mf ↵
   ```

4. Compile the HDL into libraries by typing the following commands at the ModelSim prompt:

   ```
   vlog -work altera_mf altera_mf.v ↵
   vlog -work sgate sgate.v ↵
   vlog -work lpm 220model.v ↵
   ```

5. Compile the IPFS model by typing the following command at the ModelSim prompt:

   ```
   vlog -work work <my_IP>.vo ↵
   ```

6. Compile your RTL by typing the following command at the ModelSim prompt:

   ```
   vlog -work work <my_design>.v ↵
   ```

7. Compile the testbench by typing the following command at the ModelSim prompt:

   ```
   vlog -work work <my_testbench>.v ↵
   ```

 © March 2009 Altera Corporation

8. Load the testbench by typing the following command at the ModelSim prompt:

```
vsim -L <altera_mf library_path> -L <lpm_library_path> \
-L <sgate_library_path> work.<my_testbench> ↵
```

VHDL Example: Simulating the IPFS Model in the ModelSim Software

The following example shows the process of performing a functional simulation of a VHDL-based, megafunction IPFS model. The example assumes that the megafunction's variation and the IPFS model are generated.

1. Create a ModelSim project by performing the following steps:
   a. In the ModelSim software, on the File menu, point to **New** and click **Project**. The **Create Project** dialog box appears.
   b. Specify the name for your simulation project.
   c. Specify the desired location for your simulation project.
   d. Specify the default library name and click **OK**.
   e. Add the relevant files to your simulation project:
   - Add your design files
   - Add the IPFS model generated by the Quartus II software (if you are using the ModelSim-Altera software, skip to step 5)
   - Add the **sgate.vhd**, **sgate_pack.vhd**, **220model.vhd**, **220pack.vhd**, **altera_mf.vhd**, and **altera_mf_components.vhd** library files
   - The transceiver library files in VHDL, if you use IP with transceivers. Transceiver libraries are family independent. Refer to Table 6–2 for more information.
   - The PCIe library files in VHDL, if you use IP with the PCIe hard core for Stratix IV. Refer to Table 6–2 for more information.
2. Create the required simulation libraries by typing the following commands at the ModelSim prompt:

```
vlib sgate ↵
vlib lpm ↵
vlib altera_mf ↵
```

3. Map to the required simulation libraries by typing the following commands at the ModelSim prompt:

```
vmap sgate sgate ↵
vmap lpm lpm ↵
vmap altera_mf altera_mf ↵
```

4. Compile the HDL into libraries by typing the following commands at the ModelSim prompt:

```
vcom -work altera_mf -93 -explicit altera_mf_components.vhd ↵
vcom -work altera_mf -93 -explicit altera_mf.vhd ↵
vcom -work lpm -93 -explicit 220pack.vhd ↵
vcom -work lpm -93 -explicit 220model.vhd ↵
vcom -work sgate -93 -explicit sgate_pack.vhd ↵
```

```
vcom -work sgate -93 -explicit sgate.vhd ↵
```

5. Compile the IPFS model by typing the following command at the ModelSim prompt:

```
vcom -work work -93 -explicit <output_netlist>.vho ↵
```

6. Compile the RTL by typing the following command at the ModelSim prompt:

```
vcom -work work -93 -explicit <RTL>.vhd ↵
```

7. Compile the testbench by typing the following command at the ModelSim prompt:

```
vcom -work work -93 -explicit <my_testbench>.vhd ↵
```

8. Load the testbench by typing the following command at the ModelSim prompt:

```
vsim work.my_testbench ↵
```

NC-VHDL Example: Simulating the IPFS Model in the NC-VHDL Software

The following example shows the process of performing a functional simulation of an NC-VHDL-based, megafunction IP functional-simulation model. The example assumes that the megafunction's variation and the IPFS model are generated.

1. Create a **cds.lib** file by typing the following entries:

```
DEFINE worklib ./worklib

DEFINE sgate ./sgate

DEFINE altera_mf ./altera_mf

DEFINE lpm ./lpm
```

2. Compile library files into appropriate libraries by typing the following commands at the command prompt:

```
ncvhdl -V93 -WORK lpm 220pack.vhd ↵
ncvhdl -V93 -WORK lpm 220model.vhd ↵
ncvhdl -V93 -WORK altera_mf altera_mf_components.vhd ↵
rncvhdl -V93 -WORK altera_mf altera_mf.vhd ↵
ncvhdl -V93 -WORK sgate sgate_pack.vhd ↵
ncvhdl -V93 -WORK sgate sgate.vhd ↵
```

3. Compile source code and testbench files by typing the following commands at the command prompt:

```
ncvhdl -V93 -WORK worklib <my_design>.vhd ↵
ncvhdl -V93 -WORK worklib <my_testbench>.vhd ↵
ncvhdl -V93 -WORK worklib <my_IPtoolbench_output_netlist>.vho ↵
```

4. Elaborate the design by typing the following command at the command prompt:

```
ncelab worklib.<my_testbench>:entity ↵
```

 © March 2009 Altera Corporation

Verilog HDL Example: Simulating Your IPFS Model in VCS

The following example illustrates the process of performing a functional simulation of a design that contains a Verilog HDL-based, megafunction IPFS model. This example assumes that the megafunction variation and the IPFS model are generated.

Single-Step Process

For the single-step process, type the following at the command prompt:

```
vcs <testbench>.v <RTL>.v <output_netlist>.v -v 220model.v \
altera_mf.v sgate.v -R ↵
```

Two-Step Process (Compilation and Simulation)

For compilation and simulation, perform the following steps:

1. Compile your design files by typing the following at the command prompt:

```
vcs <testbench>.v <RTL>.v <output_netlist>.v -v 220model.v \
altera_mf.v sgate.v -o simulation_out ↵
```

2. Load your simulation by typing the following at a command prompt:

```
source simulation_out ↵
```

For more information about simulating a design in VCS, refer to the *Synopsys VCS and VCS-MX Support* chapter in volume 3 of the *Quartus II Handbook*.

Conclusion

Altera Quartus II software provides full support for simulating IP megafunctions with third-party tools either directly or using its NativeLink feature. Using the Quartus II software, you can also generate IPFS models for supported megafunctions that enhance and simplify design verification. Using an IPFS model is transparent, requiring only the addition of different files in which to synthesize and simulate projects.

Referenced Documents

This chapter references the following documents:

- *AN 189: Simulating Nios Embedded Processor Designs*
- *Synopsys VCS and VCS-MX Support* chapter in volume 3 of the *Quartus II Handbook*
- *Volume 1: Design Synthesis* of the *Quartus II Handbook*

Document Revision History

Table 6–3 shows the revision history for this chapter.

Table 6–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Removed figures.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	Updated for the Quartus II 8.1 software release
May 2008 v8.0.0	■ Updated “Intoduction.” ■ Updated “Simulating Altera IP without the Quartus II NativeLink Feature.” ■ Updated “Verilog HDL Example: Simulating the IPFS Model in the ModelSim Software.” ■ Updated “VHDL Example: Simulating the IPFS Model in the ModelSim Software.” ■ Updated Figure 6–2. ■ Updated Table 6–1. ■ Updated Table 6–2.	Updated for the Quartus II software, version 8.0.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

 © March 2009 Altera Corporation

Section II. Timing Analysis

As designs become more complex, advanced timing analysis capability requirements grow. Static timing analysis is a method of analyzing, debugging, and validating the timing performance of a design. The Quartus® II software provides the features necessary to perform advanced timing analysis for today's system-on-a-programmable-chip (SOPC) designs.

Synopsys PrimeTime is an industry standard sign-off tool, used to perform static timing analysis on most ASIC designs. The Quartus II software provides a path to enable you to run PrimeTime on your Quartus II software designs, and export a netlist, timing constraints, and libraries to the PrimeTime environment.

This section explains the basic principles of static timing analysis, the advanced features supported by the Quartus II Timing Analyzer, and how you can run PrimeTime on your Quartus designs.

This section includes the following chapters:

- Chapter 7, The Quartus II TimeQuest Timing Analyzer
- Chapter 8, Best Practices for the Quartus II TimeQuest Timing Analyzer
- Chapter 9, Switching to the Quartus II TimeQuest Timing Analyzer
- Chapter 10, Quartus II Classic Timing Analyzer
- Chapter 11, Synopsys PrimeTime Support

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

7. The Quartus II TimeQuest Timing Analyzer

QII53018-9.0.0

Introduction

The Quartus® II TimeQuest Timing Analyzer is a powerful ASIC-style timing analysis tool that validates the timing performance of all logic in your design using an industry-standard constraint, analysis, and reporting methodology. Use the Quartus II TimeQuest Timing Analyzer's GUI or command-line interface to constrain, analyze, and report results for all timing paths in your design.

Before running the Quartus II TimeQuest Timing Analyzer, you must specify initial timing constraints that describe the clock characteristics, timing exceptions, and signal transition arrival and required times. You can specify timing constraints in the Synopsys Design Constraints (**.sdc**) file format using the GUI or command-line interface. The Quartus II Fitter optimizes the placement of logic to meet your constraints.

During timing analysis, the Quartus II TimeQuest Timing Analyzer analyzes the timing paths in the design, calculates the propagation delay along each path, checks for timing constraint violations, and reports timing results as slack in the **Report** pane and in the **Console** pane. If the Quartus II TimeQuest Timing Analyzer reports any timing violations, you can customize the reporting to view precise timing information about specific paths, and then constrain those paths to correct the violations. When your design is free of timing violations, you can be confident that the logic will operate as intended in the target device.

The Quartus II TimeQuest Timing Analyzer is a complete static timing analysis tool that you can use as a sign-off tool for Altera® FPGAs and HardCopy® ASICs.

This chapter contains the following sections:

- "Getting Started with the Quartus II TimeQuest Timing Analyzer" on page 7–2
- "Compilation Flow with the Quartus II TimeQuest Timing Analyzer Guidelines" on page 7–2
- "Timing Analysis Overview" on page 7–6
- "The Quartus II TimeQuest Timing Analyzer Flow Guidelines" on page 7–19
- "Collections" on page 7–21
- "SDC Constraint Files" on page 7–22
- "Clock Specification" on page 7–24
- "I/O Specifications" on page 7–39
- "Timing Exceptions" on page 7–44
- "Constraint and Exception Removal" on page 7–51
- "Timing Reports" on page 7–51
- "Timing Analysis Features" on page 7–74
- "The TimeQuest Timing Analyzer GUI" on page 7–79

■ "Conclusion" on page 7–89

For more information about the TimeQuest Timing Analyzer and the SOPC Builder, refer to *Volume 4: SOPC Builder* in the *Quartus II Handbook*.

Getting Started with the Quartus II TimeQuest Timing Analyzer

The Quartus II TimeQuest Timing Analyzer caters to the needs of the most basic to the most advanced designs for FPGAs.

This section provides a brief overview of the Quartus II TimeQuest Timing Analyzer, including the necessary steps to properly constrain a design, perform a full place-and-route, and perform reporting on the design.

Setting Up the Quartus II TimeQuest Timing Analyzer

The Quartus II software version 7.2 and later supports two native timing analysis tools: Quartus II TimeQuest Timing Analyzer and Quartus II Classic Timing Analyzer. When you specify the Quartus II TimeQuest Timing Analyzer as the default timing analysis tool, the Quartus II TimeQuest Timing Analyzer guides the Fitter and analyzes timing results after compilation.

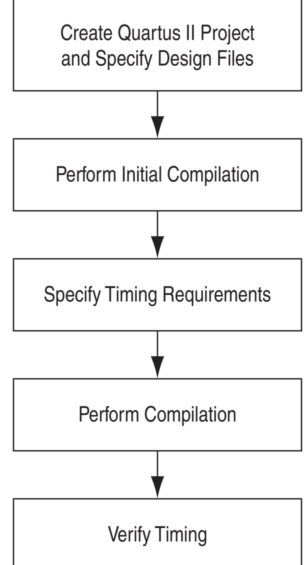
To specify the Quartus II TimeQuest Timing Analyzer as the default timing analyzer, on the Assignments menu, click **Settings**. In the **Settings** dialog box, in the **Category** list, select **Timing Analysis Settings** and turn on **Use TimeQuest Timing Analyzer during compilation**.

To add the TimeQuest icon to the Quartus II toolbar, on the Tools menu, click **Customize**. In the Customize dialog box, click the **Toolbars** tab, turn on **Processing**, and click **Close**.

Compilation Flow with the Quartus II TimeQuest Timing Analyzer Guidelines

When you enable the Quartus II TimeQuest Timing Analyzer as the default timing analyzer, everything from constraint validation to timing verification is performed by the Quartus II TimeQuest Timing Analyzer. Figure 7–1 shows the recommended design flow steps to maximize and leverage the benefits the Quartus II TimeQuest Timing Analyzer. Details about each step are provided after the figure.

Figure 7–1. Design Flow with the Quartus II TimeQuest Timing Analyzer



- **Create Quartus II Project and Specify Design Files**—Creates a project before you can compile design files. In this step you specify the target FPGA, any EDA tools used in the design cycle, and all design files.

  You can also modify existing design files for design optimization and add additional design files. For example, you can add HDL files or schematics to the project.

- **Perform Initial Compilation**—Creates an initial design database before you specify timing constraints for your design. Perform Analysis and Synthesis to create a post-map database, or perform a full compilation to create a post-fit database.

  Creating a post-map database for the initial compilation is faster than creating a post-fit database. A post-map database is sufficient for the initial database.

  Creating a post-fit database is recommended only if you previously created and specified an **.sdc** file for the project. A post-map database is sufficient for the initial compilation.

- **Specify Timing Requirements**—Timing requirements guide the Fitter as it places and routes your design.

  You must enter all timing constraints and exceptions in an **.sdc** file. This file must be included as part of the project. To add this file to your project, on the Project menu, click **Add/Remove Files in Project** and add the **.sdc** file in the **Files** dialog box.

- **Perform Compilation**—Synthesizes, places, and routes your design into the target FPGA.

  When compilation is complete, the TimeQuest Timing Analyzer generates summary clock setup and clock hold, recovery, and removal reports for all defined clocks in the design.

- **Verify Timing—**Verifies timing in your design with the Quartus II TimeQuest Timing Analyzer. Refer to "The Quartus II TimeQuest Timing Analyzer Flow Guidelines" on page 7–19.

Running the Quartus II TimeQuest Timing Analyzer

You can run the Quartus II TimeQuest Timing Analyzer in one of the following modes:

- Directly from the Quartus II software
- Stand-alone mode
- Command-line mode

This section describes each of the modes, and the behavior of the Quartus II TimeQuest Timing Analyzer.

Directly from the Quartus II Software

To run the Quartus II TimeQuest Timing Analyzer from the Quartus II software, on the Tools menu, click **TimeQuest Timing Analyzer**. The Quartus II TimeQuest Timing Analyzer is available after you have created a database for the current project. The database can be either a post-map or post-fit database; perform Analysis and Synthesis to create a post-map database, or a full compilation to create a post-fit database.

After a database is created, you can create a timing netlist based on that database. If you create a post-map database, you cannot create a post-fit timing netlist in the Quartus II TimeQuest Timing Analyzer.

When you launch the TimeQuest Timing Analyzer directly from the Quartus II software, the current project opens by default.

Stand-Alone Mode

To run the Quartus II TimeQuest Timing Analyzer in stand-alone mode, type the following command at the command prompt:

```
quartus_staw ↵
```

In stand-alone mode, you can perform static analysis on any project that contains either a post-map or post-fit database. To open a project, double-click **Open Project** in the **Tasks** pane.

Command-Line Mode

Use command-line mode for easy integration with scripted design flows. Using the command-line mode avoids interaction with the user interface provided by the Quartus II TimeQuest Timing Analyzer, but allows the automation of each step of the static timing analysis flow. Table 7–1 provides a summary of the options available in the command-line mode.

Table 7–1. Summary of Command Line Options

Command Line Option	Description
`-h | --help`	Provides help information on `quartus_sta`.
`-t <script file> |` `--script=<script file>`	Sources the <*script file*>.
`-s | --shell`	Enters shell mode.
`--tcl_eval <tcl command>`	Evaluates the Tcl command <*tcl command*>.

`--do_report_timing`	For all clocks in the design, run the following commands:
	`report_timing -npaths 1 -to_clock $clock`
	`report_timing -setup -npaths 1 -to_clock $clock`
	`report_timing -hold -npaths 1 -to_clock $clock`
	`report_timing -recovery -npaths 1 -to_clock $clock`
	`report_timing -removal -npaths 1 -to_clock $clock`

`--force_dat`	Forces the Delay Annotator to annotate the new delays from the recently compiled design to the compiler database.
`--lower_priority`	Lowers the computing priority of the `quartus_sta` process.
`--post_map`	Uses the post-map database results.
`--qsf2sdc`	Converts assignments from the Quartus II Settings File (**.qsf**) format to the Synopsys Design Constraints File format.
`--sdc=<SDC file>`	Specifies the **.sdc** file to read.
`--report_script=<script>`	Specifies a custom report script to be called.
`--speed=<value>`	Specifies the device speed grade to be used for timing analysis.
`--tq2hc`	Generate temporary files to convert the Quartus II TimeQuest Timing Analyzer **.sdc** file(s) to a PrimeTime **.sdc** file that can be used by the HardCopy Design Center (HCDC).
`--tq2pt`	Generates temporary files to convert the Quartus II TimeQuest Timing Analyzer **.sdc** file(s) to a PrimeTime **.sdc** file.
`-f <argument file>`	Specifies a file containing additional command-line arguments.
`-c <revision name> |` `--rev=<revision_name>`	Specifies which revision and its associated Quartus II Settings File (**.qsf**) to use.
`--multicorner`	Specifies that all slack summary reports be generated for both slow and fast corners.
`--multicorner[=on|off]`	Turns off the multicorner analysis by using the off value.
`--voltage=<value_in_mV>`	Specifies the device voltage (mV) to be used in timing analysis.
`--temperature=` `<value_in_C>`	Specifies the device temperature (C) to be used in timing analysis.
`--parallel` `[=<num_processors>]`	Specifies the number of computer processors to use on a multi-processor system.
`--64bit`	Enables 64-bit version of the executable.

To run the Quartus II TimeQuest Timing Analyzer in command-line mode, type the following command at the command prompt:

```
quartus_sta <options> ↵
```

Timing Analysis Overview

This section provides an overview of the Quartus II TimeQuest Timing Analyzer concepts. Understanding these concepts allows you to take advantage of the powerful timing analysis features available in the Quartus II TimeQuest Timing Analyzer.

The Quartus II TimeQuest Timing Analyzer follows the flow shown in Figure 7–2 when it analyzes your design. Table 7–2 lists the most commonly used commands for each step.

Figure 7–2. The Quartus II TimeQuest Timing Analyzer Flow

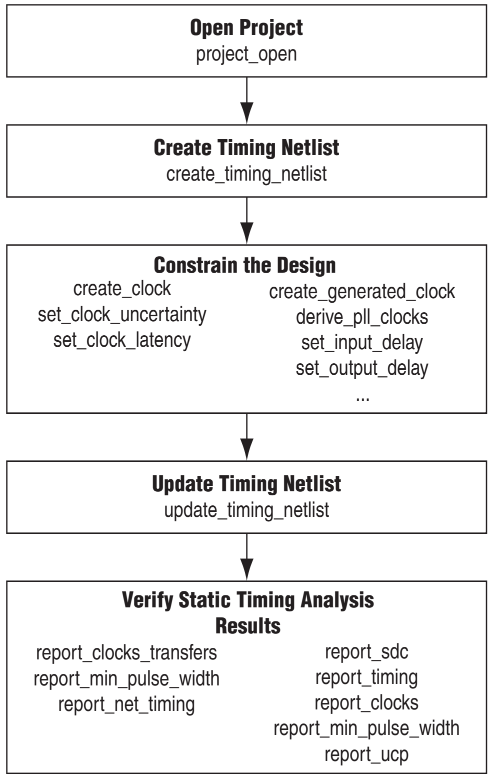


Table 7–2 describes Quartus II TimeQuest Timing Analyzer terminology.

Table 7–2. Quartus II TimeQuest Timing Analyzer Terms (Part 1 of 2)

Terminology	Definition
Nodes	Most basic timing netlist unit. Use to represent ports, pins, and registers.
Keepers	Ports or registers. *(1)*
Cells	Look-up table (LUT), registers, digital signal processing (DSP) blocks, TriMatrix memory, IOE, and so on. *(2)*
Pins	Inputs or outputs of cells.
Nets	Connections between pins.
Ports	Top-level module inputs or outputs; for example, device pins.

Table 7–2. Quartus II TimeQuest Timing Analyzer Terms (Part 2 of 2)

Terminology	Definition
Clocks	Abstract objects outside of the design.

Notes to Table 7–2:

(1) Pins can indirectly refer to keepers. For example, when the value in the `-from` field of a constraint is a clock pin to a dedicated memory. In this case, the clock pin refers to a collection of registers.

(2) For Stratix® devices and other early device families, the LUT and registers are contained in logic elements (LE) and act as cells for these device families.

The Quartus II TimeQuest Timing Analyzer requires a timing netlist before it can perform a timing analysis on any design. For example, for the design shown in Figure 7–3, the Quartus II TimeQuest Timing Analyzer generates a netlist equivalent to the one shown in Figure 7–4.

Figure 7–3. Sample Design

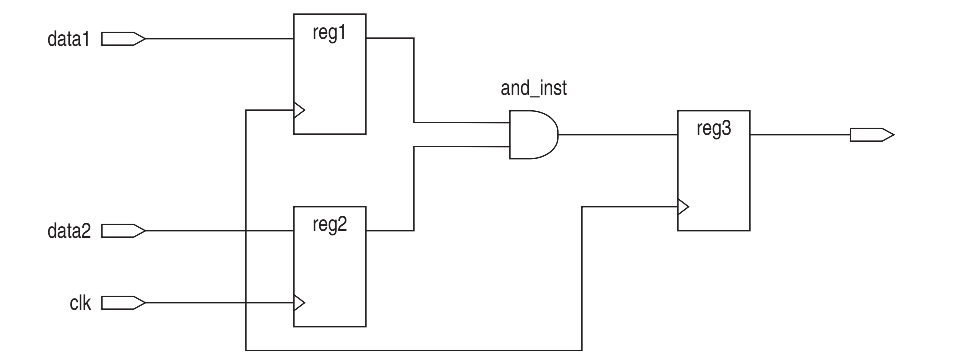


Figure 7–4. The Quartus II TimeQuest Timing Analyzer Timing Netlist

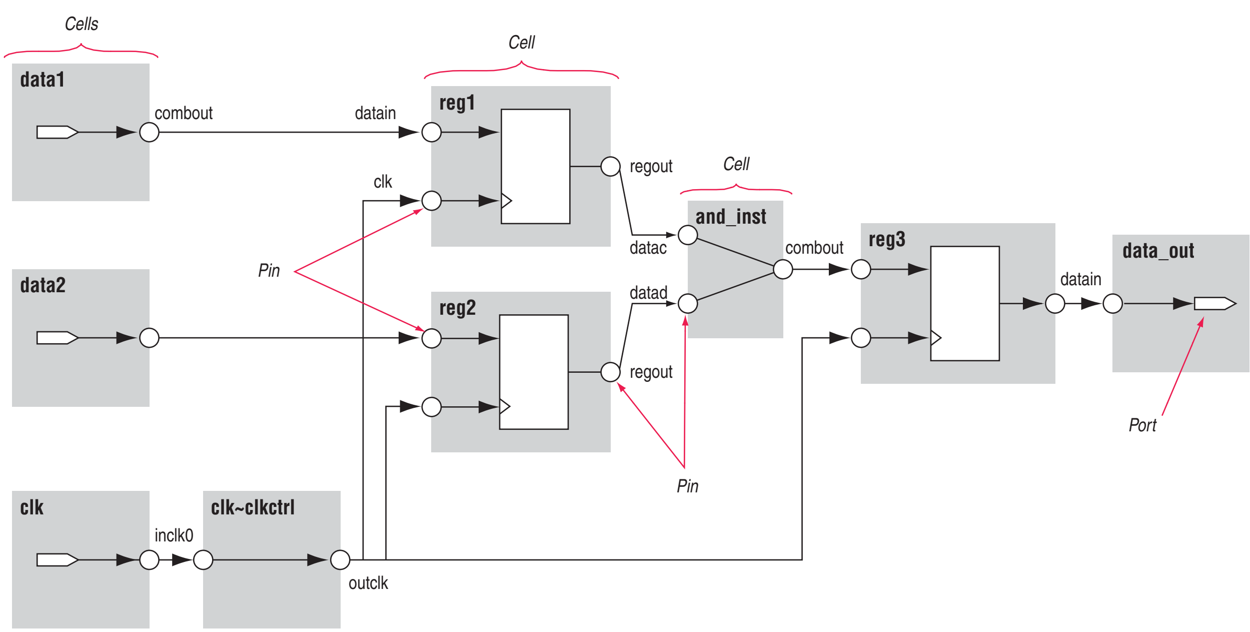


Figure 7–4 shows various cells, pins, nets, and ports. The following sample cell names are included:

- reg1
- reg2
- and_inst

The following sample pins names are included:

- data1|combout
- reg1|regout
- and_inst|combout

The following net names are included:

- data1~combout
- reg1
- and_inst

The following port names are included:

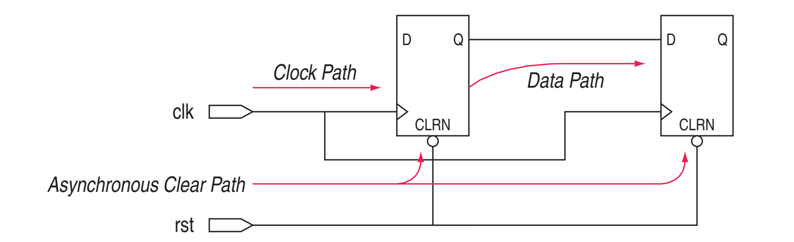
- data1, clk
- data_out

Paths connect two design nodes, such as the output of a register to the input of another register. Timing paths play a significant role in timing analysis. Understanding the types of timing paths is important to timing closure and optimization. The following list shows some of the commonly analyzed paths that are described in this section:

- **Edge paths**—the connections from ports-to-pins, from pins-to-pins, and from pins-to-ports.
- **Clock paths**—the edges from device ports or internally generated clock pins to the clock pin of a register.
- **Data paths**—the edges from a port or the data output pin of a sequential element to a port or the data input pin of another sequential element.
- **Asynchronous paths**—the edges from a port or sequential element to the asynchronous set or clear pin of a sequential element.

Figure 7–5 shows some of these commonly analyzed path types.

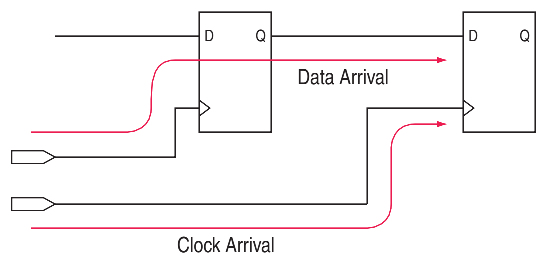
Figure 7–5. Path Types



After the Quartus II TimeQuest Timing Analyzer identifies the path type, it can report data and clock arrival times for valid register-to-register paths. The Quartus II TimeQuest Timing Analyzer calculates data arrival time by adding the delay from the clock source to the clock pin of the source register, the micro clock-to-out (μt_{CO}) of the source register, and the delay from the source register's Q pin to the destination register's D pin, where the μt_{CO} is the intrinsic clock-to-out for the internal registers in the FPGA.

The Quartus II TimeQuest Timing Analyzer calculates clock arrival time by adding the delay from the clock source to the destination register's clock pin. Figure 7–6 shows a data arrival path and a clock arrival path. The Quartus II TimeQuest Timing Analyzer calculates data required time by accounting for the clock arrival time and micro setup time (μt_{SU}) of the destination register, where the μt_{SU} is the intrinsic setup for the internal registers in the FPGA.

Figure 7–6. Data Arrival and Clock Arrival

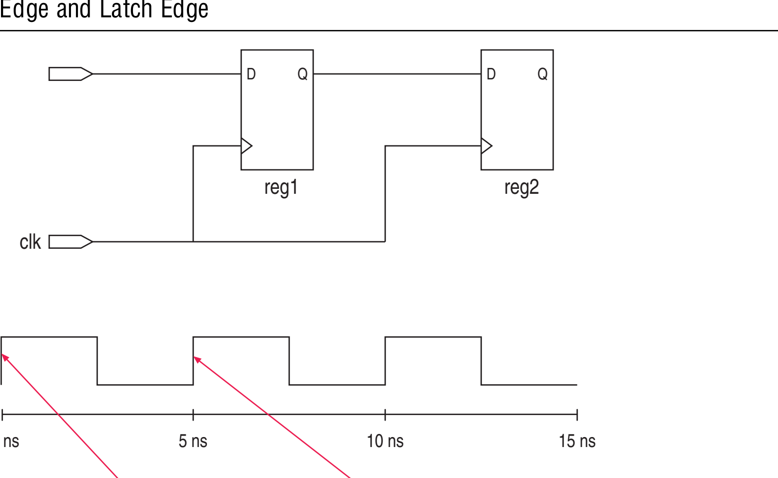


In addition to identifying various paths in a design, the Quartus II TimeQuest Timing Analyzer analyzes clock characteristics to compute the worst-case requirement between any two registers in a single register-to-register path. You should constrain all clocks in your design before performing this analysis.

The launch edge is an active clock edge that sends data out of a sequential element, acting as a source for the data transfer. A latch edge is the active clock edge that captures data at the data port of a sequential element, acting as a destination for the data transfer.

Figure 7–7 shows a single-cycle system that uses consecutive clock edges to transfer and capture data, a register-to-register path, and the corresponding launch and latch edges timing diagram. In this example, the launch edge sends the data out of register `reg1` at 0 ns, and register `reg2` latch edge captures the data at 5 ns.

Figure 7–7. Launch Edge and Latch Edge



The Quartus II TimeQuest Timing Analyzer validates clock setup and hold requirements relative to the launch and latch edges.

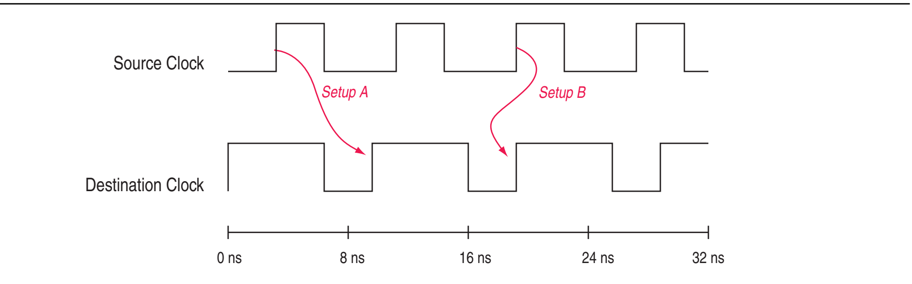
Clock Analysis

A comprehensive static timing analysis includes analysis of register-to-register, I/O, and asynchronous reset paths. The Quartus II TimeQuest Timing Analyzer uses data required times, data arrival times, and clock arrival times to verify circuit performance and detect possible timing violations. The Quartus II TimeQuest Timing Analyzer determines the timing relationships that must be met for the design to correctly function and checks arrival times against required times to verify timing.

Clock Setup Check

To perform a clock setup check, the Quartus II TimeQuest Timing Analyzer determines a setup relationship by analyzing each launch and latch edge for each register-to-register path. For each latch edge at the destination register, the Quartus II TimeQuest Timing Analyzer uses the closest previous clock edge at the source register as the launch edge. In Figure 7–8, two setup relationships are defined and are labeled setup A and setup B. For the latch edge at 10 ns, the closest clock that acts as a launch edge is at 3 ns and is labeled setup A. For the latch edge at 20 ns, the closest clock that acts as a launch edge is 19 ns and is labeled setup B.

Figure 7–8. Setup Check



The Quartus II TimeQuest Timing Analyzer reports the result of clock setup checks as slack values. Slack is the margin by which a timing requirement is met or not met. Positive slack indicates the margin by which a requirement is met; negative slack indicates the margin by which a requirement is not met. The Quartus II TimeQuest Timing Analyzer determines clock setup slack, as shown in Equation 7–1, for internal register-to-register paths.

Equation 7–1.

$$\text{Clock Setup Slack} = \text{Data Required Time} - \text{Data Arrival Time}$$
$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay to Source Register} + \mu t_{CO} + \text{Register-to-Register Delay}$$
$$\text{Data Required} = \text{Clock Arrival Time} - \mu t_{SU} - \text{Setup Uncertainty}$$
$$\text{Clock Arrival Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register}$$

If the data path is from an input port to a internal register, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 7–2 to calculate the setup slack time.

Equation 7–2.

$$\text{Clock Setup Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}$$
$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay} + \text{Input Maximum Delay of Pin} + \text{Pin-to-Register Delay}$$
$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} - \mu t_{SU}$$

If the data path is an internal register to an output port, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 7–3 to calculate the setup slack time.

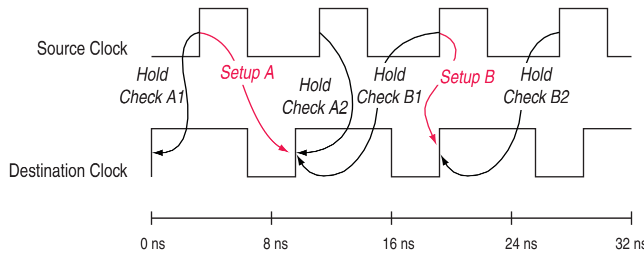
Equation 7–3.

$$\text{Clock Setup Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}$$
$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay to Source Register} + \mu t_{CO} + \text{Register-to-Pin Delay}$$
$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay} - \text{Output Maximum Delay of Pin}$$

Clock Hold Check

To perform a clock hold check, the Quartus II TimeQuest Timing Analyzer determines a hold relationship for each possible setup relationship that exists for all source and destination register pairs. The Quartus II TimeQuest Timing Analyzer checks all adjacent clock edges from all setup relationships to determine the hold relationships. The Quartus II TimeQuest Timing Analyzer performs two hold checks for each setup relationship. The first hold check determines that the data launched by the current launch edge is not captured by the previous latch edge. The second hold check determines that the data launched by the next launch edge is not captured by the current latch edge. Figure 7–9 shows two setup relationships labeled setup A and setup B. The first hold check is labeled hold check A1 and hold check B1 for setup A and setup B, respectively. The second hold check is labeled hold check A2 and hold check B2 for setup A and setup B, respectively.

Figure 7–9. Hold Checks



From the possible hold relationships, the Quartus II TimeQuest Timing Analyzer selects the hold relationship that is the most restrictive. The hold relationship with the largest difference between the latch and launch edges (that is, latch – launch and not the absolute value of latch and launch) is selected because this determines the minimum allowable delay for the register-to-register path. For Figure 7–9, the hold relationship selected is hold check A2.

The Quartus II TimeQuest Timing Analyzer determines clock hold slack as shown in Equation 7–4.

Equation 7–4.

$$\text{Clock Hold Slack} = \text{Data Arrival Time} - \text{Data Required Time}$$

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay to Source Register} + \mu t_{CO} + \text{Register-to-Register Delay}$$

$$\text{Data Required Time} = \text{Clock Arrival Time} + \mu t_{H} + \text{Hold Uncertainty}$$

$$\text{Clock Arrival Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register}$$

If the data path is from an input port to an internal register, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 7–5 to calculate the hold slack time.

Equation 7–5.

$$\text{Clock Hold Slack Time} = \text{Data Arrival Time} - \text{Data Required Time}$$

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay} + \text{Input Minimum Delay of Pin} + \text{Pin-to-Register Delay}$$

$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} + \mu t_{H}$$

If the data path is an internal register to an output port, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 7–6 to calculate the setup hold time.

Equation 7–6.

$$\text{Clock Hold Slack Time} = \text{Data Arrival Time} - \text{Data Required Time}$$

$$\text{Data Arrival Time} = \text{Latch Edge} + \text{Clock Network Delay to Source Register} + \mu t_{CO} + \text{Register-to-Pin Delay}$$

$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay} - \text{Output Minimum Delay of Pin}$$

Recovery and Removal

Recovery time is the minimum length of time the de-assertion of an asynchronous control signal; for example, `clear` and `preset`, must be stable before the next active clock edge. The recovery slack time calculation is similar to the clock setup slack time calculation, but it applies to asynchronous control signals. If the asynchronous control signal is registered, the Quartus II TimeQuest Timing Analyzer uses Equation 7–7 to calculate the recovery slack time.

Equation 7–7.

$$\text{Recovery Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}$$
$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay to Source Register} + \mu t_{CO} + \text{Register-to-Register Delay}$$
$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} - \mu t_{SU}$$

If the asynchronous control is not registered, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 7–8 to calculate the recovery slack time.

Equation 7–8.

$$\text{Recovery Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}$$
$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay} + \text{Maximum Input Delay} + \text{Port-to-Register Delay}$$
$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register Delay} - \mu t_{SU}$$

If the asynchronous reset signal is from a port (device I/O), you must make an Input Maximum Delay assignment to the asynchronous reset port for the Quartus II TimeQuest Timing Analyzer to perform recovery analysis on that path.

Removal time is the minimum length of time the de-assertion of an asynchronous control signal must be stable after the active clock edge. The Quartus II TimeQuest Timing Analyzer removal time slack calculation is similar to the clock hold slack calculation, but it applies asynchronous control signals. If the asynchronous control is registered, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 7–9 to calculate the removal slack time.

Equation 7–9.

$$\text{Removal Slack Time} = \text{Data Arrival Time} - \text{Data Required Time}$$
$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay to Source Register} + \mu t_{CO} \text{ of Source Register} + \text{Register-to-Register Delay}$$
$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} + \mu t_{H}$$

If the asynchronous control is not registered, the Quartus II TimeQuest Timing Analyzer uses the equations shown in Equation 7–10 to calculate the removal slack time.

Equation 7–10.

$$\text{Removal Slack Time} = \text{Data Arrival Time} - \text{Data Required Time}$$

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Clock Network Delay} + \text{Input Minimum Delay of Pin} + \text{Minimum Pin-to-Register Delay}$$

$$\text{Data Required Time} = \text{Latch Edge} + \text{Clock Network Delay to Destination Register} + \mu t_H$$

If the asynchronous reset signal is from a device pin, you must specify the Input Minimum Delay constraint to the asynchronous reset pin for the Quartus II TimeQuest Timing Analyzer to perform a removal analysis on this path.

Multicycle Paths

Multicycle paths are data paths that require more than one clock cycle to latch data at the destination register. For example, a register may be required to capture data on every second or third rising clock edge. Figure 7–10 shows an example of a multicycle path between a multiplier's input registers and output register where the destination latches data on every other clock edge.

Figure 7–10. Example Diagram of a Multicycle Path

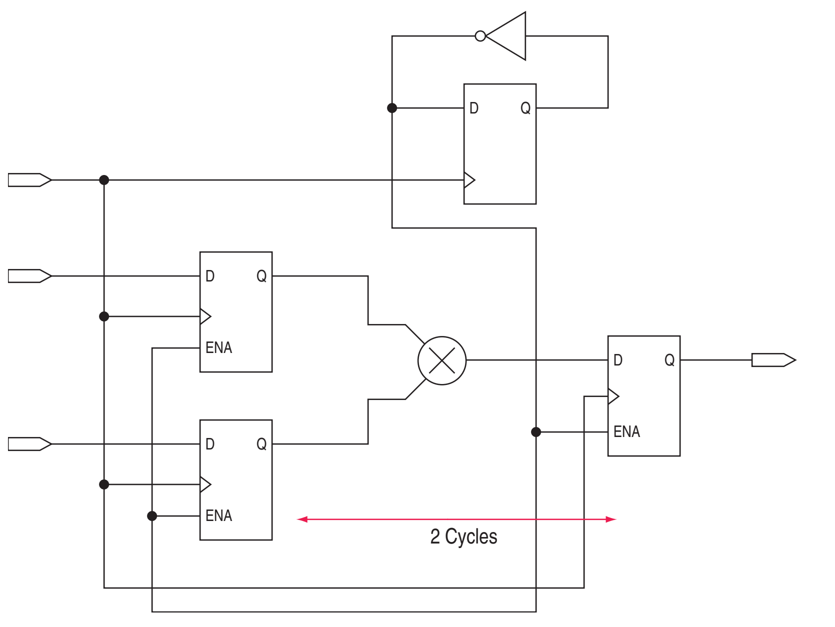


Figure 7–11 shows a register-to-register path where the source clock, `src_clk`, has a period of 10 ns and the destination clock, `dst_clk`, has a period of 5 ns.

Figure 7–11. Register-to-Register Path

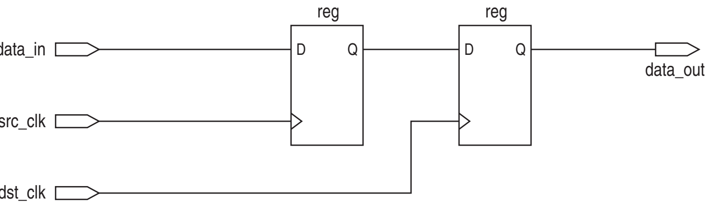
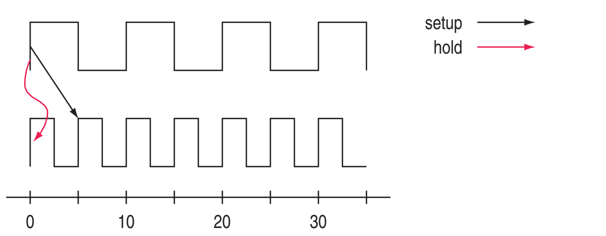


Figure 7–12 shows the respective timing diagrams for the source and destination clocks and the default setup and hold relationships. The default setup relationship is 5 ns; the default hold relationship is 0 ns.

Figure 7–12. Default Setup and Hold Timing Diagram



The default setup and hold relationships can be modified with the `set_multicycle_path` command to accommodate the system requirements.

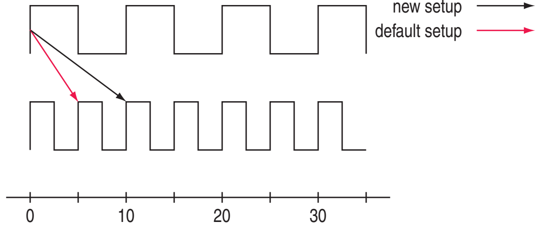
Table 7–3 shows the commands used to modify either the launch or latch edge times that the Quartus II TimeQuest Timing Analyzer Timing Analyzer uses to determine a setup relationship or hold relationship.

Table 7–3. Commands to Modify Edge Times

Command	Description of Modification
`set_multicycle_path -setup -end`	Latch edge time of the setup relationship
`set_multicycle_path -setup -start`	Launch edge time of the setup relationship
`set_multicycle_path -hold -end`	Latch edge time of the hold relationship
`set_multicycle_path -hold -start`	Launch edge time of the hold relationship

Figure 7–13 shows the timing diagram after a multicycle setup of two has been applied. The command moves the latch edge time to 10 ns from the default 5 ns.

Figure 7–13. Modified Setup Diagram



Metastability

Metastability problems can occur when a signal is transferred between circuitry in unrelated or asynchronous clock domains because the designer cannot guarantee that the signal will meet setup and hold time requirements. To minimize the failures due to metastability, circuit designers typically use a sequence of registers (synchronization register chain or synchronizer) in the destination clock domain to resynchronize the data signals to the new clock domain.

The Mean Time Before Failure (MTBF) is an estimate of the average time between instances of failure due to metastability.

The TimeQuest Timing Analyzer analyzes the robustness of the design for metastability and can calculate the MTBF for synchronization register chains in the design. The MTBF of the entire design is then estimated based on the synchronization chains it contains.

In addition to reporting synchronization register chains found in the design, the Quartus II software also protects these registers from optimizations that might negatively impact MTBF, such as register duplication and logic retiming. The Quartus II software can also optimize the MTBF of your design if the MTBF is too low.

Refer to "report_metastability" on page 7–57 for information about how to enable metastability analysis and report metastability in the TimeQuest Timing Analyzer.

For more information about metastability, its effects in FPGAs, and how MTBF is calculated, refer to the *Understanding Metastability in FPGAs* White Paper. For more information about metastability analysis, reporting, and optimization features in the Quartus II software, refer to the *Managing Metastability with the Quartus II Software* chapter in volume 1 of the *Quartus II Handbook*.

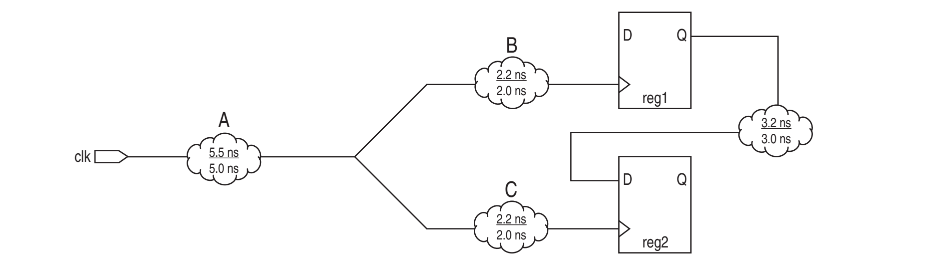
Common Clock Path Pessimism

Common clock path pessimism (CCPP) removal accounts for the minimum and maximum delay variation associated with common clock paths during a static timing analysis. CCPP removal accounts for this variation by adding the difference between the maximum and minimum delay value of the common clock path to the appropriate slack equation.

The minimum and maximum delay variation might arise when two different delay values are used for the same clock path. For example, in a simple setup analysis, the maximum clock path delay to the source register is used to determine the data arrival time. The minimum clock path delay to the destination register is used to determine the data required time. However, if the clock path to the source register and to the destination register share a common clock path, the analysis uses both the maximum delay and the minimum delay to model the common clock path. This results in an overly pessimistic analysis since two different delay values, the maximum and minimum delays, cannot be used to model the same clock path.

Figure 7–14 shows a typical register-to-register path with the maximum and minimum delay values shown.

Figure 7–14. Common Clock Path



Segment A is the common clock path between reg1 and reg2. The minimum delay is 5.0 ns; the maximum delay is 5.5 ns. The difference between the maximum and minimum delay value equals the CCPP removal value; in this case, CCPP equals 0.5 ns. The CCPP removal value is then added to the appropriate slack equation. Therefore, if the setup slack for the register-to-register in Figure 7–14 equals 0.7 ns *without* CCPP removal, the slack would be 1.2 ns *with* CCPP removal.

CCPP is also used when determining the minimum pulse width of a register. A clock signal must meet a register's minimum pulse width requirement to be recognized by the register. A minimum high time defines the minimum pulse width for a positive-edge triggered register. A minimum low time defines the minimum pulse width for a negative-edge triggered register.

Clock pulses that violate the minimum pulse width of a register prevent data from being latched at the data pin of the register. To calculate the slack of the minimum pulse width, the required minimum pulse width time is subtracted from the actual minimum pulse width time. The actual minimum pulse width time is determined by the clock requirement specified for the clock that feeds the clock port of the register. The required minimum pulse width time is determined by the maximum rise, minimum rise, maximum fall, and minimum fall times. Figure 7–15 shows a diagram of the required minimum pulse width time for both the high pulse and low pulse.

Figure 7–15. Required Minimum Pulse Width

With CCPP, the minimum pulse width slack can be increased by the smallest value of either the maximum rise time minus the minimum rise time, or the maximum fall time minus the minimum fall time. For Figure 7–15, the slack value can be increased by 0.2 ns, which is the smallest value between 0.3 ns (0.8 ns – 0.5 ns) and 0.2 ns (0.9 ns – 0.7 ns).

Refer to "report_min_pulse_width" on page 7–59 for more information about reporting CCPP in the TimeQuest Timing Analyzer.

You must use the **Enable common clock path pessimism removal** option to account for CCPP in the Fitter and timing analysis. This option defaults to ON for Stratix III, Cyclone III, and newer device families.

To access this option, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, next to **Timing Analysis Settings**, click the "+" icon to expand the menu. Click **TimeQuest Timing Analyzer**.

3. Turn on **Enable common clock path pessimism removal**.
4. Click **OK**.

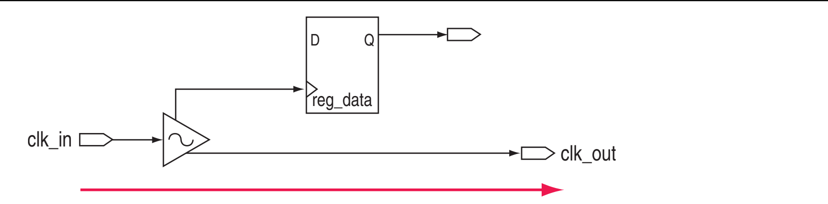
CCPP is supported for Stratix III, Cyclone III, and newer devices.

Clock-As-Data

The majority of FPGA designs contain simple connections between any two nodes known as either a data path or a clock path. A data path is a connection between the output of a synchronous element to the input of another synchronous element. A clock is a connection to the clock pin of a synchronous element. However, as FPGA designs become more complex, such as using source-synchronous interfaces, this simplified view is no longer sufficient.

The connection between port `clk_in` and port `clk_out` can be treated either as a clock path or a data path. The clock path is from the port `clk_in` to the register `reg_data` clock pin. The data path is from port `clk_in` to the port `clk_out`. In the design shown in Figure 7–16, the path from port `clk_in` to port `clk_out` is both a clock and a data path.

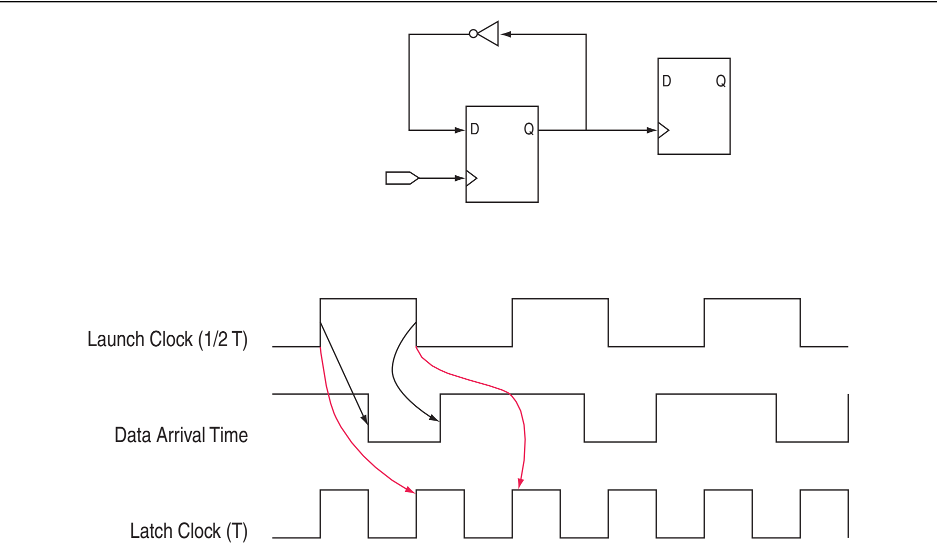
Figure 7–16. Simplified Source Synchronous Output



With clock-as-data analysis, the TimeQuest Timing Analyzer provides a more accurate analysis of the path based on the user constraints. For the clock path analysis, any phase shift associated with the PLL is taken into consideration. For the data path, any phase shift associated with the PLL is taken into consideration instead of being ignored.

The clock-as-data analysis also applies to internally generated clock dividers similar to Figure 7–17.

Figure 7–17. Clock Divider *(Note 1)*



Note to Figure 7–17:

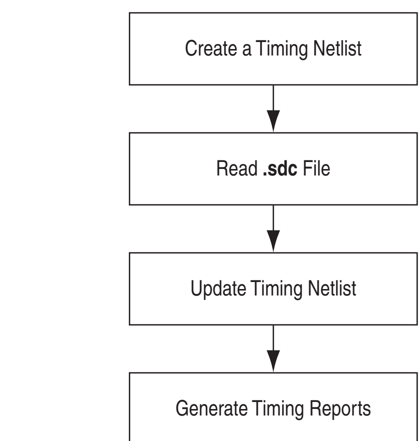
(1) In this figure, the inverter feedback path is analyzed during timing analysis. The output of the divider register is used to determine the launch time and the clock port of the register is used to determine the latch time.

A source-synchronous interface contains a clock signal that travels in parallel with data signals. The clock and data pair originates or terminates at the same device.

The Quartus II TimeQuest Timing Analyzer Flow Guidelines

Use the steps shown in Figure 7–18 to verify timing in the TimeQuest Timing Analyzer.

Figure 7–18. Timing Verification in the TimeQuest Timing Analyzer



The following sections describe each of the steps shown in Figure 7–18.

Create a Timing Netlist

After you perform a full compilation, you must create a timing netlist based on the fully annotated database from the post-fit results.

To create the timing netlist, double-click **Create Timing Netlist** in the **Tasks** pane, or type the following command in the **Console** pane:

```
create_timing_netlist ↵
```

Read the Synopsys Design Constraints File

After you create a timing netlist, you must read an **.sdc** file. This step reads all constraints and exceptions defined in the **.sdc** file.

You can read the **.sdc** file from either the **Task** pane or the **Console** pane.

To read the **.sdc** file from the **Tasks** pane, double-click the **Read SDC File** command.

The **Read SDC File** task reads the <*current revision*>**.sdc** file.

To read the **.sdc** file from the Console, type the following command in the Console:

```
read_sdc ↵
```

For more information about reading **.sdc** files in the TimeQuest Timing Analyzer, refer to "Synopsys Design Constraints File Precedence" on page 7–24.

Update Timing Netlist

You must update the timing netlist after you read an **.sdc** file. The TimeQuest Timing Analyzer applies all constraints to the netlist for verification and removes any invalid or false paths in the design from verification.

To update the timing netlist, double-click Update Timing Netlist in the **Tasks** pane, or type the following command in the Console pane:

```
update_timing_netlist ↵
```

Generate Timing Reports

You can generate timing reports for all critical paths in your design. The Tasks pane contains the commonly used reporting commands. Individual or custom reports can be generated for your design.

For more information about reporting, refer to the section "Timing Reports" on page 7–51.

For a full list of available report application program interfaces (APIs), refer to the *SDC & TimeQuest API Reference Manual*.

As you verify timing, you may encounter failures along critical paths. If this occurs, you can refine the existing constraints or create new ones to change the effects of existing constraints. If you modify, remove, or add constraints, you should perform a full compilation. This allows the Fitter to re-optimize the design based on the new constraints and brings you back to the Perform Compilation step in the process. This iterative process allows you to resolve your timing violations in the design.

For a sample Tcl script to automate the timing analysis flow, refer to the TimeQuest Quick Start Tutorial.

Collections

The Quartus II TimeQuest Timing Analyzer supports collection APIs that provide easy access to ports, pins, cells, or nodes in the design. Use collection APIs with any valid constraints or Tcl commands specified in the Quartus II TimeQuest Timing Analyzer.

Table 7–4 describes the collection commands supported by the Quartus II TimeQuest Timing Analyzer.

Table 7–4. Collection Commands

Command	Description
`all_clocks`	Returns a collection of all clocks in the design.
`all_inputs`	Returns a collection of all input ports in the design.
`all_outputs`	Returns a collection of all output ports in the design.
`all_registers`	Returns a collection of all registers in the design.
`get_cells`	Returns a collection of cells in the design. All cell names in the collection match the specified pattern. Wildcards can be used to select multiple cells at the same time.
`get_clocks`	Returns a collection of clocks in the design. When used as an argument to another command, such as the `-from` or `-to` of `set_multicycle_path`, each node in the clock represents all nodes clocked by the clocks in the collection. The default uses the specific node (even if it is a clock) as the target of a command.
`get_nets`	Returns a collection of nets in the design. All net names in the collection match the specified pattern. You can use wildcards to select multiple nets at the same time.
`get_pins`	Returns a collection of pins in the design. All pin names in the collection match the specified pattern. You can use wildcards to select multiple pins at the same time.
`get_ports`	Returns a collection of ports (design inputs and outputs) in the design.

Table 7–5 describes the SDC extension collection commands supported by the Quartus II TimeQuest Timing Analyzer.

Table 7–5. SDC Extension Collection Commands (Part 1 of 2)

Command	Description
`get_fanouts <filter>`	Returns a collection of fan-out nodes starting from <*filter*>.
`get_keepers <filter>`	Returns a collection of keeper nodes (non-combinational nodes) in the design.
`get_nodes <filter>`	Returns a collection of nodes in the design. The `get_nodes` collection cannot be used when specifying constraints or exceptions.
`get_partitions <filter>`	Returns a collection of partitions matching the <*filter*>.

Table 7–5. SDC Extension Collection Commands (Part 2 of 2)

Command	Description
`get_registers <filter>`	Returns a collection of registers in the design.
`get_fanins <filter>`	Returns a collection of fan-in nodes starting from <*filter*>.
`derive_pll_clocks`	Automatically creates generated clocks on the outputs of the PLL. The generated clock properties reflect the PLL properties that have been specified by the MegaWizard™ Plug-In Manager.
`get_assignment_groups <filter>`	Returns either a collection of keepers, ports, or registers that have been saved to the Quartus Settings File (**.qsf**) with the **Assignment (Time) Groups** option.
`remove_clock <clock list>`	Removes the list of clocks specified by <*clock list*>.
`set_scc_mode <size>`	Allows you to set the maximum Strongly Connected Components (SCC) loop size or force the Quartus II TimeQuest Timing Analyzer to always estimate delays through SCCs.
`set_time_format`	Sets time format, including time unit and decimal places.

For more information about collections, refer to the **.sdc** file and the *SDC and TimeQuest API Reference Manual*.

Application Examples

Example 7–1 shows various uses of the `create_clock` and `create_generated_clock` commands and specific design structures.

Example 7–1. create_clock and set_multicycle_path Commands and Specific Design Structures

```
# Create a simple 10 ns with clock with a 60 % duty cycle
create_clock -period 10 -waveform {0 6} -name clk [get_ports clk]
# The following multicycle applies to all paths ending at registers
# clocked by clk
set_multicycle_path -to [get_clocks clk] 2
```

SDC Constraint Files

The Quartus II TimeQuest Timing Analyzer stores all timing constraints in an **.sdc** file. You can create an **.sdc** file with different constraints for place-and-route and for timing analysis.

The **.sdc** file should contain only SDC and Tcl commands. Commands to manipulate the timing netlist or control the compilation flow should not be included in the **.sdc** file.

The Quartus II software does not automatically update **.sdc** files. You must explicitly write new or updated constraints in the TimeQuest Timing Analyzer GUI. Use the `write_sdc` command, or, in the Quartus II TimeQuest Timing Analyzer, on the Constraints menu, click **Write SDC File** to write your constraints to an **.sdc** file.

The constraints in the **.sdc** file are order-sensitive. A constraint must first be declared before any references are made to that constraint. For example, if a generated clock references a base clock with a name `clk`, the base clock constraint must be declared before the generated clock constraint.

Fitter and Timing Analysis with SDC Files

You can specify the same or different **.sdc** files for the Quartus II Fitter for place-and-route, and the Quartus II TimeQuest Timing Analyzer for static timing analysis. Using different **.sdc** files allows you to have one set of constraints for place-and-route and another set of constraints for final timing sign-off in the Quartus II TimeQuest Timing Analyzer.

Specifying SDC Files for Place-and-Route

To specify an **.sdc** file for the Fitter, you must add the **.sdc** file to your Quartus II project. To add the file to your project, use the following command in the Tcl console:

```
set_global_assignment -name SDC_FILE <SDC file name> ↵
```

Or, in the Quartus II software GUI, on the Project menu, click **Add/Remove Files in Project**.

The Fitter optimizes your design based on the requirements in the **.sdc** files in your project.

The results shown in the timing analysis report located in the Compilation Report are based on the **.sdc** files added to the project.

You must specify the Quartus II TimeQuest Timing Analyzer as the default timing analyzer to make the Fitter read the **.sdc** file.

Specifying SDC Files for Static Timing Analysis

After you create a timing netlist in the Quartus II TimeQuest Timing Analyzer, you must specify timing constraints and exceptions before you can perform a timing analysis. The timing requirements do not have to be identical to those provided to the Fitter. You can specify your timing requirements manually or you can read a previously created **.sdc** file.

To manually enter your timing requirements, you can use constraint entry dialog boxes or SDC commands. If you have an **.sdc** file that contains your timing requirements, use this file to apply your timing requirements. To specify the **.sdc** file for timing analysis in the Quartus II TimeQuest Timing Analyzer, use the following command:

```
read_sdc [<SDC file name>] ↵
```

If you use the TimeQuest GUI to apply the **.sdc** file for timing analysis, in the Quartus II TimeQuest Timing Analyzer, on the Constraints menu, click **Read SDC File**.

The `read_sdc` command has the `-hdl` option, allowing `read_sdc` to read SDC commands embedded in HDL that uses that `ALTERA_ATTRIBUTE` attribute.

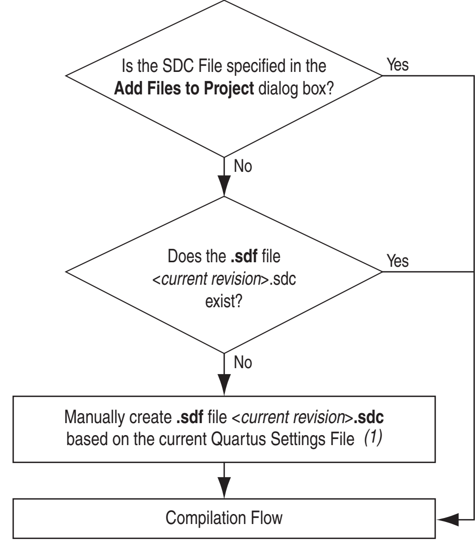
The `read_sdc` command without any options reads both your **.sdc** files and any HDL-embedded commands. By default, the **Read SDC File** command in the Tasks pane reads the **.sdc** files specified in the Quartus II Settings File (**.qsf**), which are the same **.sdc** files used by the Fitter.

Synopsys Design Constraints File Precedence

The Quartus II Fitter and the Quartus II TimeQuest Timing Analyzer reads the **.sdc** files from the files list in the **.qsf** file in the order they are listed, from top to bottom.

The Quartus II software searches for an **.sdc** file, as shown in Figure 7–19.

Figure 7–19. Synopsys Design Constraints File Order of Precedence



Note to Figure 7–19:

(1) This occurs only in the Quartus II TimeQuest Timing Analyzer and not during compilation in the Quartus II software. The Quartus II TimeQuest Timing Analyzer has the ability to automate the conversion of the QSF timing assignments to SDC if no **.sdc** file exists when the Quartus II TimeQuest Timing Analyzer is opened.

If you type the `read_sdc` command at the command line without any arguments, the precedence order shown in Figure 7–19 is followed.

Clock Specification

The specification of all clocks and any associated clock characteristics in your design is essential for accurate static timing analysis results. The Quartus II TimeQuest Timing Analyzer supports many SDC commands that accommodate various clocking schemes and any clock characteristics.

This section describes the **.sdc** file API available to create and specify clock characteristics.

Clocks

Use the `create_clock` command to create a clock at any register, port, or pin. You can create each clock with unique characteristics. Example 7–2 shows the `create_clock` command and options.

Example 7–2. create_clock Command

```
create_clock
-period <period value>
[-name <clock name>]
[-waveform <edge list>]
[-add]
<targets>
```

Table 7–6 describes the options for the `create_clock` command.

Table 7–6. create_clock Command Options

Option	Description
`-period <period value>`	Specifies the clock period. You can also specify the clock period in units of frequency, such as `-period <num>MHz`. (1)
`-name <clock name>`	Name of the specific clock; for example, `sysclock`. If you do not specify the clock name, the clock name is the same as the node to which it is assigned.
`-waveform <edge list>`	Specifies the clock's rising and falling edges. The edge list alternates between rising edge and falling edge. For example, a 10 ns period where the first rising edge occurs at 0 ns and the first falling edge occurs at 5 ns would be written as `-waveform {0 5}`. The difference must be within one period unit, and the rise edge must come before the fall edge. The default edge list is `{0 <period>/2}`, or a 50% duty cycle.
`-add`	Allows you to specify more than one clock to the same port or pin.
`<targets>`	Specifies the port(s) or pin(s) to which the assignment applies. If source objects are not specified, the clock is a virtual clock. Refer to "Virtual Clocks" on page 7–28 for more information.

Note to Table 7–6:

(1) The default time unit in the Quartus II TimeQuest Timing Analyzer is nanoseconds (ns).

Example 7–3 shows how to create a 10 ns clock with a 50% duty cycle, where the first rising edge occurs at 0 ns applied to port `clk`.

Example 7–3. 100MHz Clock Creation

```
create_clock -period 10 -waveform { 0 5 } clk
```

Example 7–4 shows how to create a 10 ns clock with a 50% duty cycle that is phase shifted by 90 degrees applied to port `clk_sys`.

Example 7–4. 100MHz Shifted by 90 Degrees Clock Creation

```
create_clock -period 10 -waveform { 2.5 7.5 } clk_sys
```

Clocks defined with the `create_clock` command have a default source latency value of zero. The Quartus II TimeQuest Timing Analyzer automatically computes the clock's network latency for non-virtual clocks.

Generated Clocks

The Quartus II TimeQuest Timing Analyzer considers clock dividers, ripple clocks, or circuits that modify or change the characteristics of the incoming or master clock as generated clocks. You should define the output of these circuits as generated clocks. This definition allows the Quartus II TimeQuest Timing Analyzer to analyze these clocks and account for any network latency associated with them.

Use the `create_generated_clock` command to create generated clocks. Example 7–5 shows the `create_generated_clock` command and the available options.

Example 7–5. create_generated_clock Command

```
create_generated_clock
[-name <clock name>]
-source <master pin>
[-edges <edge list>]
[-edge_shift <shift list>]
[-divide_by <factor>]
[-multiply_by <factor>]
[-duty_cycle <percent>]
[-add]
[-invert]
[-master_clock <clock>]
[-phase <phase>]
[-offset <offset>]
<targets>
```

Table 7–7 describes the options for the `create_generated_clock` command.

Table 7–7. create_generated_clock Command Options (Part 1 of 2)

Option	Description
`-name <clock name>`	Name of the generated clock; for example, `clk_x2`. If you do not specify the clock name, the clock name is the same as the first node to which it is assigned.
`-source <master pin>`	The <*master pin*> specifies the node in the design from which the clock settings derive.
`-edges <edge list>` | `-edge_shift <shift list>`	The `-edges` option specifies the new rising and falling edges with respect to the master clock's rising and falling edges. The master clock's rising and falling edges are numbered `1..`<*n*> starting with the first rising edge; for example, edge 1. The first falling edge after that is edge number 2, the next rising edge number 3, and so on. The <*edge list*> must be in ascending order. The same edge may be used for two entries to indicate a clock pulse independent of the original waveform's duty cycle. `-edge_shift` specifies the amount of shift for each edge in the <*edge list*>. The `-invert` option can be used to invert the clock after the `-edges` and `-edge_shifts` are applied. *(1)*
`-divide_by <factor>` | `-multiply_by <factor>`	The `-divide_by` and `-multiply_by` factors are based on the first rising edge of the clock, and extend or contract the waveform by the specified factors. For example, a `-divide_by 2` is equivalent to `-edges {1 3 5}`. For multiplied clocks, the duty cycle can also be specified. The Quartus II TimeQuest Timing Analyzer supports specifying multiply and divide factors at the same time.
`-duty_cycle <percent>`	Specifies the duty cycle of the generated clock. The duty cycle is applied last.
`-add`	Allows you to specify more than one clock to the same pin.

Table 7–7. create_generated_clock Command Options (Part 2 of 2)

Option	Description
-invert	Inversion is applied at the output of the clock after all other modifications are applied, except duty cycle.
-master_clock <*clock*>	-master_clock is used to specify the clock if multiple clocks exist at the master pin.
-phase <*phase*>	Specifies the phase of the generated clock.
-offset <*offset*>	Specifies the offset of the generated clock.
<*targets*>	Specifies the port(s) or pin(s) to which the assignment applies.

Note to Table 7–7:

(1) The Quartus II TimeQuest Timing Analyzer supports a maximum of three edges in the edge list.

Source latencies are based on clock network delays from the master clock (not necessarily the master pin). You can use the `set_clock_latency -source` command to override source latency.

Figure 7–20 shows how to create an inverted generated clock based on a 10 ns clock.

Figure 7–20. Generating an Inverted Clock

```
create_clock -period 10 [get_ports clk]
create_generated_clock -divide_by 1 -invert -source [get_registers clk] \
   [get_registers gen|clkreg]
```

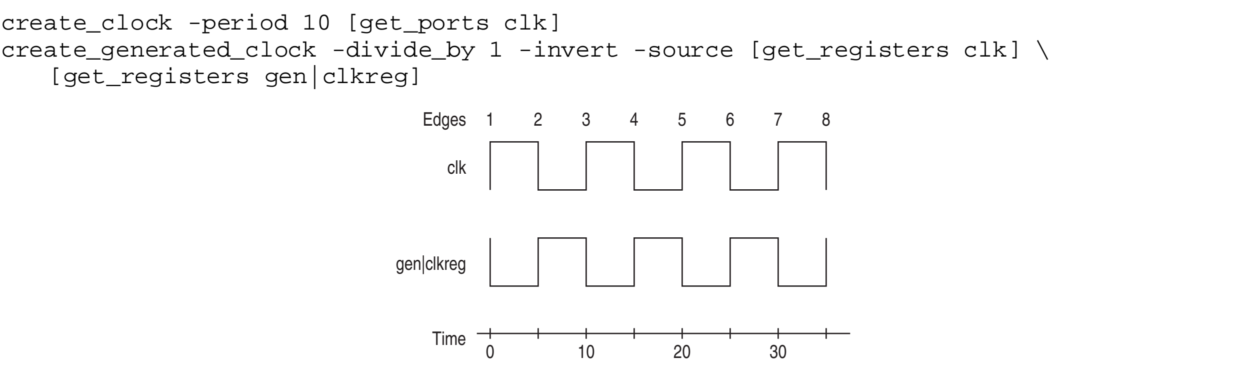


Figure 7–21 shows how to modify the generated clock using the -edges and -edge_shift options.

Figure 7–21. Edges and Edge Shifting a Generated Clock

```
create_clock -period 10 -waveform { 0 5} [get_ports clk]
# Creates a divide-by-t clock
create_generated_clock -source [get_ports clk] -edges {1 3 5 } [get_registers \
clkdivA|clkreg]
# Creates a divide-by-2 clock independent of the master clocks' duty cycle (now 50%)
create_generated_clock -source [get_ports clk] -edges {1 1 5} -edge_shift { 0 2.5 0 } \
[get_registers clkdivB|clkreg]
```


Figure 7–22 shows the effect of the -multiply_by option on the generated clock.

Figure 7–22. Multiplying a Generated Clock

```
create_clock -period 10 -waveform { 0 5 } [get_ports clk]
# Creates a multiply-by-2 clock
create_generated_clock -source [get_ports clk] -multiply_by 2 [get_registers \
clkmult|clkreg]
```


Virtual Clocks

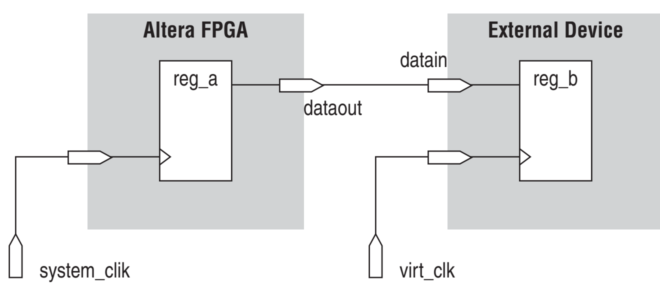
A virtual clock is a clock that does not have a real source in the design or that does not interact directly with the design. For example, if a clock feeds only an external device's clock port and not a clock port in the design, and the external device then feeds (or is fed by) a port in the design, it is considered a virtual clock.

Use the `create_clock` command to create virtual clocks, with no value specified for the `source` option.

You can use virtual clocks for `set_input_delay` and `set_output_delay` constraints.

Figure 7–23 shows an example where a virtual clock is required for the Quartus II TimeQuest Timing Analyzer to properly analyze the relationship between the external register and those in the design. Because the oscillator labeled `virt_clk` does not interact with the Altera device, but acts as the clock source for the external register, the clock `virt_clk` must be declared. Example 7–6 shows the command to create a 10 ns virtual clock named `virt_clk` with a 50% duty cycle where the first rising edge occurs at 0 ns. The virtual clock is then used as the clock source for an output delay constraint.

Figure 7–23. Virtual Clock Board Topology



After you create the virtual clock, you can perform register-to-register analysis between the register in the Altera device and the register in the external device.

Example 7–6. Virtual Clock Example 1

```
#create base clock for the design
create_clock -period 5 [get_ports system_clk]
#create the virtual clock for the external register
create_clock -period 10 -name virt_clk -waveform { 0 5 }
#set the output delay referencing the virtual clock
set_output_delay -clock virt_clk -max 1.5 [get_ports dataout]
```

Example 7–7 shows the command to create a 10 ns virtual clock with a 50% duty cycle that is phase shifted by 90°.

Example 7–7. Virtual Clock Example 2

```
create_clock -name virt_clk -period 10 -waveform { 2.5 7.5 }
```

Multi-Frequency Clocks

Certain designs have more than one clock source feeding a single clock port in the design. The additional clock may act as a low-power clock, with a lower frequency than the primary clock. To analyze this type of design, the `create_clock` command supports the `-add` option that allows you to add more than one clock to a clock node.

Example 7–8 shows the command to create a 10 ns clock applied to clock port `clk`, and then add an additional 15 ns clock to the same clock port. The Quartus II TimeQuest Timing Analyzer uses both clocks when it performs timing analysis.

Example 7–8. Multi-Frequency Example

```
create_clock -period 10 -name clock_primary -waveform { 0 5 } [get_ports
clk]
create_clock -period 15 -name clock_secondary -waveform { 0 7.5 }
[get_ports clk] -add
```

Automatic Clock Detection

To create clocks for all clock nodes in your design automatically, use the `derive_clocks` command. This command creates clocks on ports or registers to ensure every register in the design has a clock.

Example 7–9 shows the `derive_clocks` command and options.

Example 7–9. derive_clocks Command

```
derive_clocks
[-period <period value>]
[-waveform <edge list>]
```

Table 7–8 describes the options for the `derive_clocks` command.

Table 7–8. derive_clocks Command Options

Option	Description
`-period <period value>`	Creates the clock period. You can also specify the frequency as `-period <num>MHz`. (1)
`-waveform <edge list>`	Creates the clock's rising and falling edges. The edge list alternates between the rising edge and falling edge. For example, for a 10 ns period where the first rising edge occurs at 0 ns and the first falling edge occurs at 5 ns, the edge list is `waveform {0 5}`. The difference must be within one period unit, and the rising edge must come before the falling edge. The default edge list is `{0 period/2}`, or a 50% duty cycle.

Note to Table 7–8:

(1) This option uses the default time unit nanoseconds (ns).

The `derive_clocks` command does not create clocks for the output of the PLLs.

The `derive_clocks` command is equivalent to using `create_clock` for each register or port feeding the clock pin of a register.

Using the `derive_clocks` command for final timing sign-off is not recommended. You should create clocks for all clock sources using the `create_clock` and `create_generated_clock` commands.

Derive PLL Clocks

PLLs are used for clock management and synthesis in Altera devices. You can customize the clocks generated from the outputs of the PLL based on design requirements. Because a clock should be created for all clock nodes, all outputs of the PLL should have an associated clock.

 © March 2009 Altera Corporation

You can manually create a clock for each output of the PLL with the `create_generated_clock` command, or you can use the `derive_pll_clocks` command, which automatically searches the timing netlist and creates generated clocks for all PLL outputs according to the settings specified for each PLL output.

Use the `derive_pll_clocks` command to automatically create a clock for each output of the PLL. Example 7–10 shows the `derive_pll_clocks` command and options.

Example 7–10. derive_pll_clocks Command

```
derive_pll_clocks
[-create_base_clocks]
[-use_tan_name]
```

Table 7–9 describes the options for the `derive_pll_clocks` command.

Table 7–9. derive_pll_clocks Command Options

Option	Description
`-use_tan_name`	By default, the clock name is the output clock name. This option uses the net name similar to the names used by the Quartus II Classic Timing Analyzer.
`-create_base_clocks`	Creates the base clocks on input clock ports of the design that are feeding the PLL.

The `derive_pll_clocks` command calls the `create_generated_clock` command to create generated clocks on the outputs of the PLL. The source for the `create_generated_clock` command is the input clock pin of the PLL. Before or after the `derive_pll_clocks` command has been issued, you must manually create a base clock for the input clock port of the PLL. If a clock is not defined for the input clock node of the PLL, no clocks are reported for the PLL outputs. Instead, the Quartus II TimeQuest Timing Analyzer issues a warning message similar to Example 7–11 when the timing netlist is updated.

Example 7–11. Warning Message

```
Warning: The master clock for this clock assignment could not be
derived.
Clock: <name of PLL output clock pin name> was not created.
```

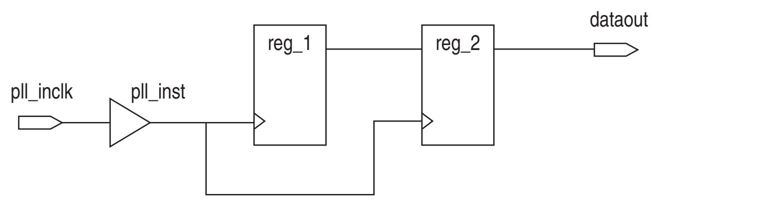
You can use the `-create_base_clocks` option to create the input clocks for the PLL inputs automatically.

You can include the `derive_pll_clocks` command in your **.sdc** file, which allows the `derive_pll_clocks` command to automatically detect any changes to the PLL. With the `derive_pll_clocks` command in your **.sdc** file, each time the file is read, the appropriate `create_generated_clocks` command for the PLL output clock pin is generated. If you use the `write_sdc-expand` command after the `derive_pll_clock` command, the new **.sdc** file contains the individual `create_generated_clock` commands for the PLL output clock pins and not the `derive_pll_clocks` command. Any changes to the properties of the PLL are not automatically reflected in the new **.sdc** file. You must manually update the `create_generated_clock` commands in the new **.sdc** file written by the `derive_pll_clocks` command to reflect the changes to the PLL.

The derive_pll_clocks constraint will also constrain any LVDS transmitters or LVDS receivers in the design by adding the appropriate multicycle constraints to account for any deserialization factors.

For example, Figure 7–24 shows a simple PLL design with a register-to-register path.

Figure 7–24. Simple PLL Design



Use the derive_pll_clocks command to automatically constrain the PLL. When this command is issued for the design shown in Figure 7–24, the messages shown in Example 7–12 are generated.

Example 7–12. derive_pll_clocks Generated Messages

```
Info:
Info: Deriving PLL Clocks:
Info: create_generated_clock -source
pll_inst|altpll_component|pll|inclk[0] -divide_by 2 -name
pll_inst|altpll_component|pll|clk[0]
pll_inst|altpll_component|pll|clk[0]
Info:
```

The node name pll_inst|altpll_component|pll|inclk[0] used for the source option refers to the input clock pin of the PLL. In addition, the name of the output clock of the PLL is the name of the PLL output clock node, pll_inst|altpll_component|pll|clk[0].

If the PLL is in clock switchover mode, multiple clocks are created for the output clock of the PLL; one for the primary input clock (for example, inclk[0]), and one for the secondary input clock (for example, inclk[1]). In this case, you should cut the primary and secondary output clocks using the set_clock_groups command with the -exclusive option.

Before you can generate any reports for this design, you must create a base clock for the PLL input clock port. Use a the following command or one similar:

```
create_clock -period 5 [get_ports pll_inclk]
```

You do not have to generate the base clock on the input clock pin of the PLL: pll_inst|altpll_component|pll|inclk[0]. The clock created on the PLL input clock port propagates to all fan-outs of the clock port, including the PLL input clock pin.

Default Clock Constraints

To provide a complete clock analysis, the Quartus II TimeQuest Timing Analyzer, by default, automatically creates clocks for all detected clock nodes in your design that have not be constrained, if there are no base clock constraints in the design. The Quartus II TimeQuest Timing Analyzer creates a base clock with a 1 GHz requirement to unconstrained clock nodes, using the following command:

```
derive_clocks -period 1 ↵
```

Individual clock constraints (for example, `create_clock` and `create_generated_clock`) should be made for all clocks in the design. This results in a thorough and realistic analysis of the design's timing requirements. Avoid using `derive_clocks` for final timing sign-off.

The default clock constraint is only applied when the Quartus II TimeQuest Timing Analyzer detects that all synchronous elements have no clocks associated with them. For example, if a design contains two clocks and only one clock has constraints, the default clock constraint is not applied. However, if both clocks have not been constrained, the default clock constraint is applied.

Clock Groups

Many clocks can exist in a design; however, not all of the clocks interact with one another and certain clock interactions are not possible.

Use the `set_clock_groups` command to specify clocks that are exclusive or asynchronous. Example 7–13 shows the `set_clock_groups` command and options.

Example 7–13. set_clock_groups Command

```
set_clock_groups
[-asynchronous | -exclusive]
-group <clock name>
[-group <clock name>]
[-group <clock name>] ...
```

Table 7–10 describes the options for the `set_clock_groups` command.

Table 7–10. set_clock_groups Command Options

Option	Description
`-asynchronous`	Asynchronous clocks—when the two clocks have no phase relationship and are active at the same time.
`-exclusive`	Exclusive clocks—when only one of the two clocks is active at any given time. An example of an exclusive clock group is when two clocks feed a 2-to-1 MUX.
`-group <clock name>`	Specifies valid destination clock names that are mutually exclusive. <*clock name*> is used to specify the clock names.

The exclusive option is used to declare when two clocks are mutually exclusive to each other and cannot coexist in the design at the same time. This can happen when multiple clocks are created on the same node or for multiplexed clocks. For example, a port can be clocked by either a 25-MHz or a 50-MHz clock. To constrain this port, two clocks should be created on the port with the `create_clock` command, then use `set_clock_groups -exclusive` to declare that they cannot coexist in the design at the same time.

This eliminates any clock transfers that may be derived between the 25-MHz clock and the 50-MHz clock. Example 7–14 shows the constraints for this.

Example 7–14. Exclusive Option

```
create_clock -period 40 -name clk_A [get_ports {port_A}]
create_clock -add -period 20 -name clk_B [get_ports {port_A}]
set_clock_groups -exclusive -group {clk_A} -group {clk_B}
```

A group is defined with the `-group` option. The TimeQuest Timing Analyzer cuts the timing paths between clocks each of the separate `-groups` groups.

The asynchronous option is used to group related and unrelated clocks. With the asynchronous option, clocks that are contained in groups are considered asynchronous to each other. Any clocks within each group are considered synchronous to each other.

For example, suppose you have three clocks: `clk_A`, `clk_B`, and `clk_C`. The clocks `clk_A` and `clk_B` are related to each other, but clock `clk_C` operates completely asynchronous with `clk_A` or `clk_B`. Example 7–15 makes `clk_A` and `clk_B` related in the same group and unrelated with the second group which contains `clk_C`.

Example 7–15. Asynchronous Option Example 1

```
set_clock_groups -asynchronous -group {clk_A clk_B} -group {clk_C}
```

Example 7–16 shows an alternative method of specifying the same constraint as Example 7–15.

Example 7–16. Asynchronous Option Example 2

```
set_clock_groups -asynchronous -group {clk_C}
```

This makes `clk_C` unrelated with every other clock in the design because `clk_C` is the only group in the constraint.

The TimeQuest Timing Analyzer assumes all clocks are related by default, unless constrained otherwise.

Example 7–17 shows a `set_clock_groups` command and the equivalent `set_false_path` commands.

Example 7–17. set_clock_groups

```
# Clocks A and C are never active when clocks B and D are active
set_clock_groups -exclusive -group {A C} -group {B D}

# Equivalent specification using false paths
set_false_path -from [get_clocks A] -to [get_clocks B]
set_false_path -from [get_clocks A] -to [get_clocks D]
set_false_path -from [get_clocks C] -to [get_clocks B]
set_false_path -from [get_clocks C] -to [get_clocks D]
set_false_path -from [get_clocks B] -to [get_clocks A]
set_false_path -from [get_clocks B] -to [get_clocks C]
set_false_path -from [get_clocks D] -to [get_clocks A]
set_false_path -from [get_clocks D] -to [get_clocks C]
```

Clock Effect Characteristics

The `create_clock` and `create_generated_clock` commands create ideal clocks that do not account for any board effects. This section describes how to account for clock effect characteristics with clock latency and clock uncertainty.

Clock Latency

There are two forms of clock latency: source and network. Source latency is the propagation delay from the origin of the clock to the clock definition point (for example, a clock port). Network latency is the propagation delay from a clock definition point to a register's clock pin. The total latency (or clock propagation delay) at a register's clock pin is the sum of the source and network latencies in the clock path.

The `set_clock_latency` command supports only source latency. The `-source` option must be specified when using the command.

Use the `set_clock_latency` command to specify source latency to any clock ports in the design. Example 7–18 shows the `set_clock_latency` command and options.

Example 7–18. set_clock_latency Command

```
set_clock_latency
-source
[-clock <clock_list>]
[-rise | -fall]
[-late | -early]
<delay>
<targets>
```

Table 7–11 describes the options for the `set_clock_latency` command.

Table 7–11. set_clock_latency Command Options (Part 1 of 2)

Option	Description
`-source`	Specifies a source latency.
`-clock <clock list>`	Specifies the clock to use if the target has more than one clock assigned to it.
`-rise | -fall`	Specifies the rising or falling delays.
`-late | -early`	Specifies the earliest or the latest arrival times to the clock.

Table 7–11. set_clock_latency Command Options (Part 2 of 2)

Option	Description
<delay>	Specifies the delay value.
<targets>	Specifies the clocks or clock sources if a clock is clocked by more than one clock.

The Quartus II TimeQuest Timing Analyzer automatically computes network latencies; therefore, the `set_clock_latency` command specifies only source latencies.

Clock Uncertainty

The `set_clock_uncertainty` command specifies clock uncertainty or skew for clocks or clock-to-clock transfers. Specify the uncertainty separately for setup and hold. Specify separate rising and falling clock transitions. The Quartus II TimeQuest Timing Analyzer subtracts setup uncertainty from the data required time for each applicable path and adds the hold uncertainty to the data required time for each applicable path.

Use the `set_clock_uncertainty` command to specify any clock uncertainty to the clock port. Example 7–19 shows the `set_clock_uncertainty` command and options.

Example 7–19. set_clock_uncertainty Command and Options

```
set_clock_uncertainty
[-rise_from <rise from clock> | -fall_from <fall from clock> |
   -from <from clock>]
[-rise_to <rise to clock> | -fall_to <fall to clock> | -to <to clock>]
[-setup | -hold]
<value>
-add
```

Table 7–12 describes the options for the `set_clock_uncertainty` command.

Table 7–12. set_clock_uncertainty Command Options

Option	Description
-from *<from clock>*	Specifies the `from` clock.
-rise_from *<rise from clock>*	Specifies the `rise-from` clock.
-fall_from *<fall from clock>*	Specifies the `fall-from` clock.
-to *<to clock>*	Specifies the `to` clock.
-rise_to *<rise to clock>*	Specifies the `rise-to` clock.
-fall_to *<fall to clock>*	Specifies the `fall-to` clock.
-setup | -hold	Specifies setup or hold.
<value>	Uncertainty value.
-add	Specifies that the uncertainty *<value>* should be added to the uncertainty value derived by the `derive_clock_uncertainty` command.

Derive Clock Uncertainty

Use the `derive_clock_uncertainty` command to automatically apply inter-clock, intra-clock, and I/O interface uncertainties. Both setup and hold uncertainties are calculated for each clock-to-clock transfer. Example 7–20 shows the `derive_clock_uncertainty` command and options.

Example 7–20. derive_clock_uncertainty Command

```
derive_clock_uncertainty
[-overwrite]
[-add]
```

Table 7–13 describes the options for the `derive_clock_uncertainty` command.

Table 7–13. derive_clock_uncertainty Command Options

Option	Description
`-overwrite`	Overwrites previously performed clock uncertainty assignments.
`-add`	Adds derived uncertainty results to any user-defined clock uncertainty assignments.

The Quartus II TimeQuest Timing Analyzer automatically applies clock uncertainties to clock-to-clock transfers in the design.

Any clock uncertainty constraints that have been applied to source and destination clock pairs with the `set_clock_uncertainty` command have a higher precedence than the clock uncertainties derived from the `derive_clock_uncertainty` command for the same source and destination clock pairs. For example, if `set_clock_uncertainty` is applied between `clka` and `clkb`, the `derive_clock_uncertainty` values for the clock transfer is ignored by default. The `set_clock_uncertainty` constraint has priority over the `derive_clock_uncertainty` constraint.

The clock uncertainty value that would have been used; however, is still reported for informational purposes. You can use the `-overwrite` command to overwrite previous clock uncertainty assignments, or remove them manually with the `remove_clock_uncertainty` command. You can also use the `-add` option to add clock uncertainty determined by the `derive_clock_uncertainty` command to any previously defined clock uncertainty value.

The following list shows the types of clock-to-clock transfers in which clock certainties can arise. They are modeled by the `derive_clock_uncertainty` command automatically.

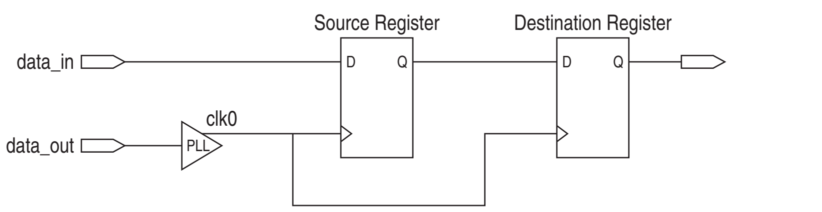
- Inter-clock
- Intra-clock
- I/O Interface

Altera recommends using the `derive_clock_uncertainty` command.

Intra-Clock Transfers

Intra-clock transfers occur when the register-to-register transfer happens in the core of the FPGA and source and destination clocks come from the same PLL output pin or clock port. An example of an intra-clock transfer is shown in Figure 7–25.

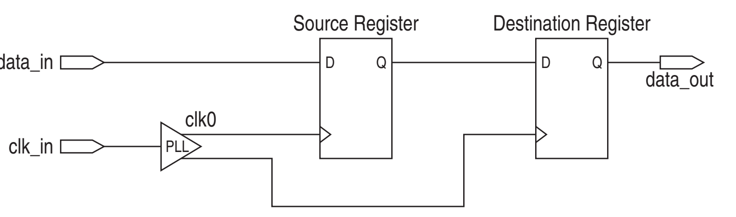
Figure 7–25. Intra-Clock Transfer



Inter-Clock Transfers

Inter-clock transfers occur when a register-to-register transfer happens in the core of the FPGA and source and destination clocks come from a different PLL output pin or clock port. An example of an inter-clock transfer is shown in Figure 7–26.

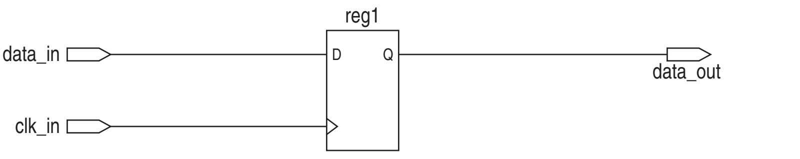
Figure 7–26. Inter-Clock Transfer



I/O Interface Clock Transfers

I/O interface clock transfers occur when data transfers from an I/O port to the core of the FPGA (input) or from the core of the FPGA to the I/O port (output). An example of an I/O interface clock transfer is shown in Figure 7–27.

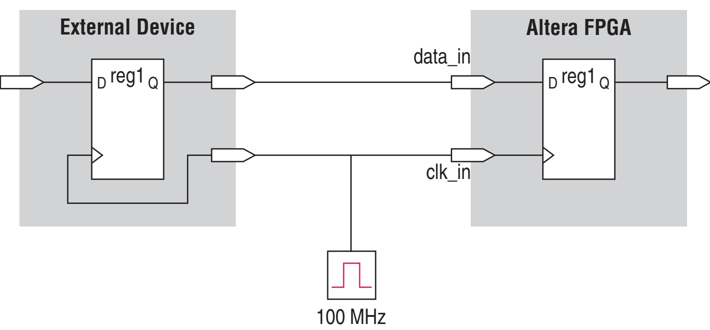
Figure 7–27. I/O Interface-Clock Transfer



For I/O interface uncertainty, you must create a virtual clock and constrain the input and output ports with the `set_input_delay` and `set_output_delay` commands that reference the virtual clock. The virtual clock is required to prevent the `derive_clock_uncertainty` command from applying clock uncertainties for either intra- or inter-clock transfers on an I/O interface clock transfer when the `set_input_delay` or `set_output_delay` commands reference a clock port or PLL output. If a virtual clock is not referenced in the `set_input_delay` or `set_output_delay` commands, the `derive_clock_uncertainty` command calculates intra- or inter-clock uncertainty value for the I/O interface.

Create the virtual clock with the same properties as the original clock that is driving the I/O port. For example, Figure 7–28 shows a typical input I/O interface with the clock specifications.

Figure 7–28. I/O Interface Specifications



Example 7–21 shows the SDC commands to constrain the I/O interface shown in Figure 7–28.

Example 7–21. SDC Commands to Constrain the I/O Interface

```
# Create the base clock for the clock port
create_clock -period 10 -name clk_in [get_ports clk_in]
# Create a virtual clock with the same properties of the base clock
# driving the source register
create_clock -period 10 -name virt_clk_in
# Create the input delay referencing the virtual clock and not the base
# clock
# DO NOT use set_input_delay -clock clk_in <delay_value>
# [get_ports data_in]
set_input_delay -clock virt_clk_in <delay value> [get_ports data_in]
```

I/O Specifications

The Quartus II TimeQuest Timing Analyzer supports Synopsys Design Constraints that constrain the ports in your design. These constraints allow the Quartus II TimeQuest Timing Analyzer to perform a system static timing analysis that includes not only the FPGA internal timing, but also any external device timing and board timing parameters.

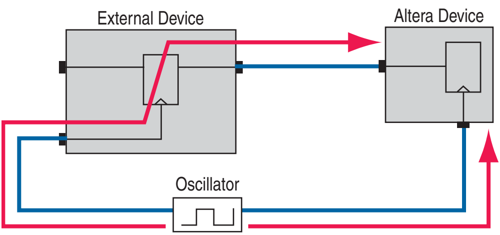
Input and Output Delay

Use input and output delay constraints to specify any external device or board timing parameters. When you apply these constraints, the Quartus II TimeQuest Timing Analyzer performs static timing analysis on the entire system.

Set Input Delay

The `set_input_delay` constraint specifies the data arrival time at a port (a device I/O) with respect to a given clock. Figure 7–29 shows an input delay path.

Figure 7–29. Set Input Delay



Use the set_input_delay command to specify input delay constraints to ports in the design. Example 7–22 shows the set_input_delay command and options.

Example 7–22. set_input_delay Command

```
set_input_delay
-clock <clock name>
[-clock_fall]
[-rise | -fall]
[-max | -min]
[-add_delay]
[-reference_pin <target>]
[-source_latency_included]
<delay value>
<targets>
```

Table 7–14 describes the options for the set_input_delay command.

Table 7–14. set_input_delay Command Options

Option	Description
-clock <*clock name*>	Specifies the source clock.
-clock_fall	Specifies the arrival time with respect to the falling edge of the clock.
-rise | -fall	Specifies either the rise or fall delay at the port.
-max | -min	Specifies the minimum or maximum data arrival time.
-add_delay	Adds another delay, but does not replace the existing delays assigned to the port.
-reference_pin <*target*>	Specifies a pin or port in the design from which to determine source and network latencies. This is useful to specify input delays relative to an output port fed by a clock.
-source_latency_ included	Specifies that the input delay value includes the source latency delay value; therefore, any source clock latency assigned to the clock is ignored.
<*delay value*>	Specifies the delay value.
<*targets*>	Specifies the destination ports or pins.

A warning message appears if you specify only a -max or -min value for the input delay value. The input minimum delay default value is equal to the input maximum delay; the input maximum delay default value is equal to the input minimum delay, if only one is specified. Similarly, a warning message appears if you specify only a -rise or -fall value for the delay value, and the delay defaults in the same manner as the input minimum and input maximum delays.

The maximum value is used for setup checks; the minimum value is used for hold checks.

By default, a set of input delays (min/max, rise/fall) is allowed for only one `-clock`, `-clock_fall`, `-reference_pin` combination. Specifying an input delay on the same port for a different clock, `-clock_fall` or `-reference_pin` removes any previously set input delays, unless you specify the `-add_delay` option. When you specify the `-add_delay` option, the worst-case values are used.

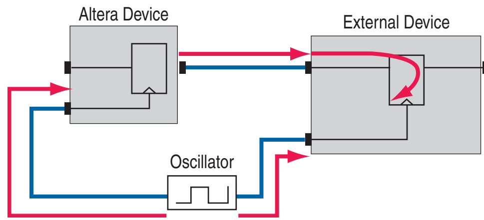
The `-rise` and `-fall` options are mutually exclusive, as are the `-min` and `-max` options.

Set Output Delay

The `set_output_delay` command specifies the data required time at a port (the device pin) with respect to a given clock.

Use the `set_output_delay` command to specify output delay constraints to ports in the design. Figure 7–30 shows an output delay path.

Figure 7–30. Output Delay



Example 7–23 shows the `set_output_delay` command and options.

Example 7–23. set_output_delay Command

```
set_output_delay
-clock <clock name>
[-clock_fall]
[-rise | -fall]
[-max | -min]
[-add_delay]
[-reference_pin <target>]
<delay value>
<targets>
```

Table 7–15 describes the options for the `set_output_delay` command.

Table 7–15. set_output_delay Command Options (Part 1 of 2)

Option	Description
`-clock <clock name>`	Specifies the source clock.
`-clock_fall`	Specifies the required time with respect to the falling edge of the clock.
`-rise | -fall`	Specifies either the rise or fall delay at the port.
`-max | -min`	Specifies the minimum or maximum data arrival time.
`-add_delay`	Adds another delay, but does not replace the existing delays assigned to the port.

Table 7–15. set_output_delay Command Options (Part 2 of 2)

Option	Description
`-reference_pin <target>`	Specifies a pin or port in the design from which to determine source and network latencies. Use this option to specify input delays relative to an output port fed by a clock.
`-source_latency_included`	Specifies that the input delay value includes the source latency delay value; therefore, any source clock latency assigned to the clock will subsequently be ignored.
`<delay value>`	Specifies the delay value.
`<targets>`	Specifies the destination ports or pins.

A warning message appears if you specify only a `-max` or `-min` value for the output delay value. The output minimum delay default value is the output maximum delay; the output maximum delay default value is the output minimum delay, if only one is specified.

The maximum value is used for setup checks; the minimum value is used for hold checks.

By default, a set of output delays (min/max, rise/fall) is allowed for only one clock, `-clock_fall`, port combination. Specifying an output delay on the same port for a different clock or `-clock_fall` removes any previously set output delays, unless you specify the `-add_delay` option. When you specify the `-add_delay` option, the worst-case values are used.

The `-rise` and `-fall` options are mutually exclusive, as are the `-min` and `-max` options.

Delay and Skew Specifications

The TimeQuest Timing Analyzer supports the ability to specify and report maximum, minimum, and skew delays between a source and destination points.

set_net_delay

Use the `set_net_delay` command in conjunction with the `report_net_delay` command to report the net delays and perform minimum or maximum analysis across nets. Example 7–24 shows the `set_net_delay` command and options.

The `set_net_delay` and `report_net_delay` commands can be used when verifying timing-critical delays for high-speed interfaces. For example, the command can be used to report the delay across a high-speed data bus for each bit.

Example 7–24. set_net_delay Command

```
set_net_delay
    -from <names>
    [-max]
    [-to <names>]
    [-min]
    <delay>
```

Table 7–16 describes the options for the `set_net_delay` command.

Table 7–16. set_net_delay Command Options

Option	Description
`-from <names>`	Valid source pins or ports (string patterns are matched using Tcl string matching).
`-max`	Specifies maximum delay.
`-min`	Specifies minimum delay.
`-to <names>`	Valid destination pins or ports (string patterns are matched using Tcl string matching). If `-to` is left unspecified, the missing value or values are substituted by an "*" character.
`<delay>`	Required delay.

When the `-min` option is specified, the slack is calculated with the minimum edge delay. When the `-max` option is specified, the slack is calculated with the maximum edge delay. When the `-skew` option is specified, the slack is calculated across all the valid edges that satisfy the `-from` and `-to` filters.

set_max_skew

Use the `set_max_skew` command to specify the maximum path-based skew requirements for registers and ports in the design. Example 7–25 shows the `set_max_delay` command and options.

Example 7–25. set_max_skew

```
set_max_skew
[-exclude <Tcl list>]
[-from <names>]
[-include <Tcl list>]
[-to <names>]
<skew>
```

Table 7–17 describes the options for the `set_max_skew` command.

Table 7–17. set_max_skew Command Options

Option	Description
`-exclude <list>`	A list of parameters to exclude during skew analysis. This list can include 1 or more of the following: `utsu`, `uth`, `utco`, `from_clock`, `to_clock`, `clock_uncertainty`, `input_delay`, `output_delay`.
`-from <names>`	Valid sources (string patterns are matched using Tcl string matching)
`-include <list>`	Tcl list of parameters to include during skew analysis. This list can include 0 or more of the following: `utsu`, `uth`, `utco`, `from_clock`, `to_clock`, `clock_uncertainty`, `input_delay`, `output_delay`.
`-to <names>`	Valid destinations (string patterns are matched using Tcl string matching)
`<skew>`	Required maximum skew

By default, the `set_max_skew` command excludes `set_input_delay`, `set_output_delay`, `utsu` and `uth`.

When this constraint is used, the results of max skew analysis are reported with the command `report_max_skew`.

For more information about the `report_max_skew` command, refer to "report_max_skew" on page 7–69.

Timing Exceptions

Timing exceptions modify the default analysis performed by the Quartus II TimeQuest Timing Analyzer. This section describes the following available timing exceptions:

- "False Path" on page 7–44
- "Minimum Delay" on page 7–45
- "Maximum Delay" on page 7–46
- "Multicycle Path" on page 7–47

Precedence

If a conflict of node names occurs between timing exceptions, the following order of precedence applies:

1. False path
2. Minimum delays and maximum delays
3. Multicycle path

The false path timing exception has the highest precedence. Within each category, assignments to individual nodes have precedence over assignments to clocks. Finally, the remaining precedence for additional conflicts is order-dependent, such that the last assignments overwrite (or partially overwrite) earlier assignments.

False Path

False paths are paths that can be ignored during timing analysis.

Use the `set_false_path` command to specify false paths in the design. Example 7–26 shows the `set_false_path` command and options.

Example 7–26. set_false_path Command

```
set_false_path
[-fall_from <clocks> | -rise_from <clocks> | -from <names>]
[-fall_to <clocks> | -rise_to <clocks> | -to <names>]
[-hold]
[-setup]
[-through <names>]
<delay>
```

Table 7–18 describes the options for the `set_false_path` command.

Table 7–18. set_false_path Command Options

Option	Description
`-fall_from <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies false path begins at the fall from <*clocks*>.
`-fall_to <clocks>`	The <names> is a collection or list of objects in the design. Specifies false path ends at the fall to <*clocks*>.
`-from <names>`	The <*names*> is a collection or list of objects in the design. Specifies false path begins at the <*names*>.
`-hold`	Specifies the false path is valid during the hold analysis only.
`-rise_from <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies false path begins at the rise from <*clocks*>.
`-rise_to <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies false path ends at the rise to <*clocks*>.
`-setup`	Specifies the false path is valid during the setup analysis only.
`-through <names>`	The <*names*> is a collection or list of objects in the design. Specifies false path passes through <*names*>.
`-to <names>`	The <*names*> is a collection or list of objects in the design. Specifies false path ends at <*names*>.
`<delay>`	Specifies the delay value.

When the objects are timing nodes, the false path only applies to the path between the two nodes. When an object is a clock, the false path applies to all paths where the source node (`-from`) or destination node (`-to`) is clocked by the clock.

Minimum Delay

Use the `set_min_delay` command to specify an absolute minimum delay for a given path. Example 7–27 shows the `set_min_delay` command and options.

Example 7–27. set_min_delay Command

```
set_min_delay
[-fall_from <clocks> | -rise_from <clocks> | -from <names>]
[-fall_to <clocks> | -rise_to <clocks> | -to <names>]
[-through <names>]
<delay>
```

Table 7–19 describes the options for the `set_min_delay` command.

Table 7–19. set_min_delay Command Options (Part 1 of 2)

Option	Description
`-fall_from <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies the minimum delay begins at the falling edge of <*clocks*>.
`-fall_to <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies the minimum delay ends at the falling of <*clocks*>.
`-from <names>`	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the start point of the path.

Table 7–19. set_min_delay Command Options (Part 2 of 2)

Option	Description
`-rise_from <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies the minimum delay at the rising edge of <*clocks*>.
`rise_to <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies the minimum delay at the rising edge of <*clocks*>.
`-through <names>`	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the through point of the path.
`-to <names>`	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the end point of the path.
`<delay>`	Specifies the delay value.

If the source or destination node is clocked, the clock paths are taken into account, allowing more or less delay on the data path. If the source or destination node has an input or output delay, that delay is also included in the minimum delay check.

When the objects are timing nodes, the minimum delay applies only to the path between the two nodes. When an object is a clock, the minimum delay applies to all paths where the source node (`-from`) or destination node (`-to`) is clocked by the clock.

You can apply the `set_min_delay` command exception to an output port that does not use a `set_output_delay` constraint. In this case, the setup summary and hold summary report the slack for these paths. Because there is no clock associated with the output port, no clock is reported for these paths and the Clock column is empty. In this case, you cannot report timing for these paths.

To report timing using clock filters for output paths with the `set_min_delay` command, you can use the `set_output_delay` command for the output port with a value of 0. You can use an existing clock from the design or a virtual clock as the clock reference in the `set_output_delay` command.

Maximum Delay

Use the `set_max_delay` command to specify an absolute maximum delay for a given path. Example 7–28 shows the `set_max_delay` command and options.

Example 7–28. set_max_delay Command

```
set_max_delay
[-fall_from <clocks> | -rise_from <clocks> | -from <names>]
[-fall_to <clocks> | -rise_to <clocks> | -to <names>]
[-through <names>]
<delay>
```

Table 7–20 describes the options for the `set_max_delay` command.

Table 7–20. set_max_delay Command Options

Option	Description
`-fall_from <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies the maximum delay begins at the falling edge of <*clocks*>.
`-fall_to <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies the maximum delay ends at the falling of <*clocks*>.
`-from <names>`	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the start point of the path.
`-rise_from <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies the maximum delay at the rising edge of <*clocks*>.
`rise_to <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies the maximum delay at the rising edge of <*clocks*>.
`-through <names>`	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the through point of the path.
`-to <names>`	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the end point of the path.
`<delay>`	Specifies the delay value.

If the source or destination node is clocked, the clock paths are taken into account, allowing more or less delay on the data path. If the source or destination node has an input or output delay, that delay is also included in the maximum delay check.

When the objects are timing nodes, the maximum delay only applies to the path between the two nodes. When an object is a clock, the maximum delay applies to all paths where the source node (`-from`) or destination node (`-to`) is clocked by the clock.

You can apply the `set_max_delay` command exception to an output port that does not use a `set_output_delay` constraint. In this case, the setup summary and hold summary report the slack for these paths. Because there is no clock associated with the output port, no clock is reported for these paths and the Clock column is empty. In this case, you cannot report timing for these paths.

To report timing using clock filters for output paths with the `set_max_delay` command, you can use the `set_output_delay` command for the output port with a value of 0. You can use an existing clock from the design or a virtual clock as the clock reference in the `set_output_delay` command.

Multicycle Path

By default, the Quartus II TimeQuest Timing Analyzer uses a single-cycle analysis. When analyzing a path, the setup launch and latch edge times are determined by finding the closest two active edges in the respective waveforms. For a hold analysis, the timing analyzer analyzes the path against two timing conditions for every possible setup relationship, not just the worst-case setup relationship. Therefore, the hold launch and latch times may be completely unrelated to the setup launch and latch edges.

A multicycle constraint relaxes setup or hold relationships by the specified number of clock cycles based on the source (`-start`) or destination (`-end`) clock. An end multicycle constraint of 2 extends the worst-case setup latch edge by one destination clock period.

Hold multicycle constraints are based on the default hold position (the default value is 0). An end hold multicycle constraint of 1 effectively subtracts one destination clock period from the default hold latch edge.

Use the `set_multicycle_path` command to specify the multicycle constraints in the design. Example 7–29 shows the `set_multicycle_path` command and options.

Example 7–29. set_multicycle_path Command

```
set_multicycle_path
[-end]
[-fall_from <clocks> | -rise_from <clocks> | -from <names>]
[-fall_to <clocks> | -rise_to <clocks> | -to <names>]
[-hold]
[-setup]
[-start]
[-through <names>]
<path multiplier>
```

Table 7–21 describes the options for the `set_multicycle_path` command.

Table 7–21. set_multicycle_path Command Options

Option	Description
`fall_from <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies the multicycle begins at the falling edge of <*clocks*>.
`fall_to <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies the multicycle ends at the falling of <*clocks*>.
`-from <names>`	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the start point of the path.
`-hold	-setup`	Specifies the type of multicycle to be applied.
`-rise_from <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies the multicycle at the rising edge of <*clocks*>.
`-rise_to <clocks>`	The <*names*> is a collection or list of objects in the design. Specifies the multicycle ends at the rising edge of <*clocks*>.
`-start	-end`	Specifies whether the start or end clock acts as the source or destination for the multicycle.
`-through <names>`	The <*names*> is a collection or list of objects in the design. Specifies multicycle passes through <*names*>.
`-to <names>`	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the end point of the path.
`<path multiplier>`	Specifies the multicycle multiplier value.

When the objects are timing nodes, the multicycle constraint only applies to the path between the two nodes. When an object is a clock, the multicycle constraint applies to all paths where the source node (`-from`) or destination node (`-to`) is clocked by the clock.

Annotated Delay

Use the `set_annotated_delay` command to annotate the cell delay between two or more pins/nodes on a cell, or the interconnect delay between two or more pins on the same net, in the current design. The annotated delay can be specified for specific transition edges: rise-rise, fall-rise, rise-fall, and fall-fall, and can also set different minimum and maximum values.

If no transition is specified, the given delay is assigned to all four values. Options `-max` and `-min` allow users to specify maximum or minimum delay.

Example 7–30 shows the `set_annotated` command and options.

Example 7–30. set_annotated_delay Command

```
set_annotated_delay
    [-cell|-net]
    [-from <names>]
    [-max|-min]
    [-operating_conditions <operating_conditions>]
    [-rr|-fr|-rf|-ff]
    [-to <names>]
    <delay>
```

Table 7–22 describes the options for the `set_annotated_delay` command.

Table 7–22. set_annotated_delay Command Options

Options	Description
`-cell`	Specifies that cell delay must be set.
`-ff`	Specifies that FF delay must be set.
`-fr`	Specifies that FR delay must be set.
`-from <names>`	Valid source pins or ports (string patterns are matched using Tcl string matching). If `-from` value is left unspecified, the "`*`" character is used.
`-max`	Specifies that only the maximum delay should be set.
`-min`	Specifies that only the minimum delay should be set.
`-net`	Specifies that net delay must be set.
`-operating_conditions <operating_conditions>`	Specifies the operating conditions Tcl object. Refer to Table 7–51 on page 7–75 for the operating conditions Tcl object.
`-rf`	Specifies that RF delay must be set.
`-rr`	Specifies that RR delay must be set.
`-to <names>`	Valid destination pins or ports (string patterns are matched using Tcl string matching). If `-to` value is left unspecified, the "`*`" character is used.
`<delay>`	The delay value in default time units.

With the `-operation_conditions` option, different operating conditions can be specified in a single **.sdc** file, removing the requirement of having multiple **.sdc** files that specify different operating conditions.

The delay annotation is deferred until the next time `update_timing_netlist` is called. To remove annotated delays, use the `remove_annotated_delay` command.

Application Examples

This section describes specific examples for the `set_multicycle_path` command.

Figure 7–31 shows a register-to-register path where the source clock, `src_clk`, has a period of 10 ns and the destination clock, `dst_clk`, has a period of 5 ns.

Figure 7–31. Register-to-Register Path

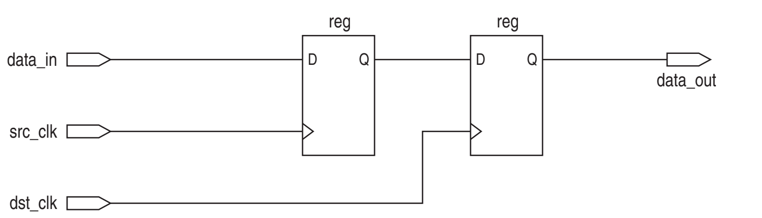
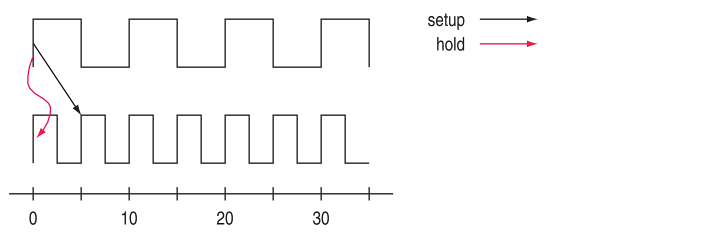


Figure 7–32 shows the respective timing diagrams for the source and destination clocks and the default setup and hold relationships. The default setup relationship is 5 ns; the default hold relationship is 0 ns.

Figure 7–32. Default Setup and Hold Timing Diagram



The default setup and hold relationships can be modified with the `set_multicycle_path` command to accommodate system requirements.

Table 7–23 shows the commands used to modify either the launch or latch edge times that the TimeQuest Timing Analyzer uses to determine a setup relationship or hold relationship.

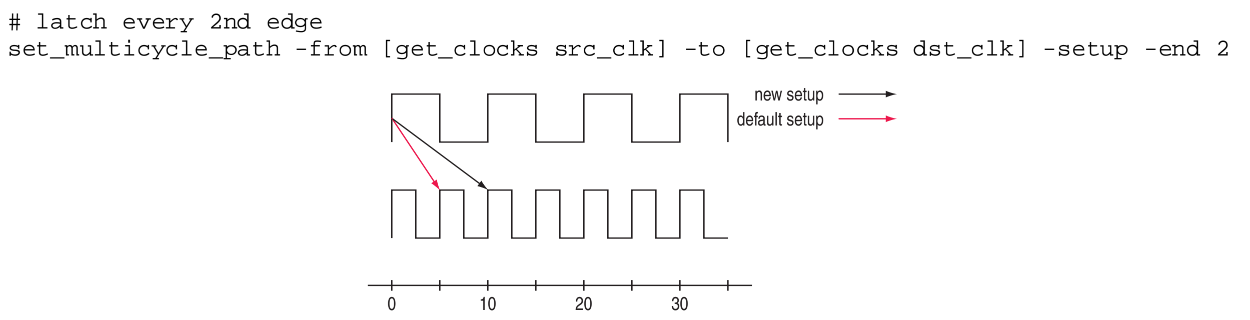
Table 7–23. Commands to Modify Edge Times

Command	Description of Modification
`set_multicycle_path -setup -end`	Latch edge time of the setup relationship
`set_multicycle_path -setup -start`	Launch edge time of the setup relationship
`set_multicycle_path -hold -end`	Latch edge time of the hold relationship
`set_multicycle_path -hold -start`	Launch edge time of the hold relationship

Figure 7–33 shows the command used to modify the setup latch edge and the resulting timing diagram. The command moves the latch edge time to 10 ns from the default 5 ns.

Figure 7–33. Modified Setup Diagram

```
# latch every 2nd edge
set_multicycle_path -from [get_clocks src_clk] -to [get_clocks dst_clk] -setup -end 2
```



Constraint and Exception Removal

When using the Quartus II TimeQuest Timing Analyzer interactively, it is usually necessary to remove a constraint or exception. In cases where constraints and exceptions either become outdated or have been erroneously entered, the Quartus II TimeQuest Timing Analyzer provides a convenient way to remove them.

Table 7–24 lists commands that allow you to remove constraints and exceptions from a design.

Table 7–24. Constraint and Exception Removal

Command	Description
`remove_clock [-all] [<clock list>]`	Removes any clocks specified by <*clock list*> that have been previously created. The `-all` option removes all declared clocks.
`remove_clock_groups -all`	Removes all clock groups previously created. Specific clock groups cannot be removed.
`remove_clock_latency -source <targets>`	Removes the clock latency constraint from the clock specified by <*targets*>.
`remove_clock_uncertainty -from <from clock> -to <to clock>`	Removes the clock uncertainty constraint from <*from clock*> to <*to clock*>.
`remove_input_delay <targets>`	Removes the input delay constraint from <*targets*>.
`remove_output_delay <targets>`	Removes the output delay constraint from <*targets*>.
`remove_annotated_delay -all`	Removes any annotated delay specified with the `set_annotated_delay` command.
`reset_design`	Removes all constraints and exceptions in the design.

Timing Reports

The Quartus II TimeQuest Timing Analyzer provides real-time static timing analysis result reports. Reports are generated only when requested. You can customize which report to display specific timing information, excluding those fields not required.

This section describes various report generation commands supported by the Quartus II TimeQuest Timing Analyzer.

report_timing

Use the `report_timing` command to generate a setup, hold, recovery, or removal report. Example 7–31 shows the `report_timing` command and options.

Example 7–31. report_timing Command

```
report_timing
[-append]
[-detail <summary|path_only|path_and_clock|full_path>]
[-fall_to_clock <names>|-rise_to_clock <names>]
[-to <names>|-to_clock <names>]
[-false_path]
[-file <name>]
[-from <names>]
[-from_clock <names>|-rise_from_clock <names>|-fall_from_clock <names>]
[-less_than_slack <slack limit>]
[-npaths <number>]
[-nworst <number>]
[-pairs_only]
[-panel_name <name>]
[-setup|-hold|-recovery|-removal]
[-show_routing]
[-stdout]
[-through <names>]
```

Table 7–25 describes the options for the `report_timing` command.

Table 7–25. report_timing Command Options (Part 1 of 2)

Option	Description
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-detail <summary|path_only|path_and_clock|full_path>`	Specifies whether or not the clock path detail is reported: `Path Only`: Clock network delay is lumped together `Summary`: Lists each individual path `Path and Clock`: Clock network delay is shown in detail `Full Path`: More clock network details, in particular for generated clocks
`-fall_from_clock <names>`	Specifies the falling edge of the <*names*> from the source register to be analyzed. The options `from_clock`, `fall_from_clock`, and `rise_from_clock` are mutually exclusive.
`-fall_to_clock <names>`	Specifies the falling edge of the <*names*> to the destination register to be analyzed; the options `to_clock`, `fall_to_clock`, and `rise_to_clock` are mutually exclusive.
`-false_path`	Reports only paths that are cut by a false path assignment.
`-file <names>`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type either ***.rpt**, ***.txt**, or ***.html**.
`-hold`	Specifies a clock hold analysis.
`-less_than_slack <slack limit>`	Limits the paths to be reported to the <*slack limit*> value.
`-npaths <number>`	Specifies the number of paths to report.
`-nworst <number>`	Restricts the number of paths per endpoint.
`-panel_name <names>`	Specifies the name of the panel in the **Reports** pane.

Table 7–25. report_timing Command Options (Part 2 of 2)

Option	Description
`-panel_name <names>`	Sends the results to the panel and specifies the name of the new panel.
`-pairs_only`	When set, paths with the same start and end points are considered equivalent; only the worst-case path for each unique combination is displayed.
`-recovery`	Specifies a recovery analysis.
`-removal`	Specifies a removal analysis.
`-rise_from_clock <names>`	Specifies the rising edge of the <*names*> from the source register to be analyzed; the options `from_clock`, `fall_from_clock`, and `rise_from_clock` are mutually exclusive.
`-rise_to_clock <names>`	Specifies the rising edge of the <*names*> to the destination register to be analyzed; the options `to_clock`, `fall_to_clock`, and `rise_to_clock` are mutually exclusive.
`-setup`	Specifies a clock setup analysis.
`-show_routing`	Displays detailed routing in the path.
`-stdout`	Indicates the report will be sent to `stdout`.
`-through <names>`	Specifies the through node for analysis.
`-to <names>`	Specifies the to node for analysis.
`-to_clock <names>`	Specifies the destination clock for analysis.

Example 7–32 shows a sample report that results from typing the following command:

```
report_timing -from_clock clk_async -to_clock clk_async -setup -npaths 1 ↵
```

Example 7–32. Sample report_timing Report (Part 1 of 2)

```
 Info:
==================================================================
Info: To Node      : dst_reg
Info: From Node    : src_reg
Info: Latch Clock  : clk_async
Info: Launch Clock : clk_async
Info:
Info: Data Arrival Path:
Info:
```

Example 7–32. Sample report_timing Report (Part 2 of 2)

```
Info: Total (ns)  Incr (ns)     Type  Node
Info: ==========  ========= ==  ====  ==========================
Info:      0.000      0.000             launch edge time
Info:      2.237      2.237  R          clock network delay
Info:      2.410      0.173     uTco  src_reg
Info:      2.410      0.000 RR  CELL  src_reg|regout
Info:      3.407      0.997 RR    IC  dataout|datain
Info:      3.561      0.154 RR  CELL  dst_reg
Info:
Info: Data Required Path:
Info:
Info: Total (ns)  Incr (ns)     Type  Node
Info: ==========  ========= ==  ====  ==========================
Info:     10.000     10.000             latch edge time
Info:     11.958      1.958  R          clock network delay
Info:     11.610     -0.348     uTsu  dst_reg
Info:
Info: Data Arrival Time  :     3.561
Info: Data Required Time :    11.610
Info: Slack              :     8.049
Info: ==========================================================
```

The report_timing command generates a report of the specified analysis type—either setup, hold, recovery, or removal. Each of the column descriptions are described in the Table 7–26.

All columns appear only when a report panel is created. If the report_timing output is directed to a file or the console, only the Total, Incr, RF, Type, and Node columns appear.

Table 7–26. Timing Report Data

Column Name	Description
Total	Shows the accumulated time delay.
Incr	Shows the increment in delay.
RF	Shows the input and output transition of the element; this can be one of the following: R, F, RR, RF, FR, FF.
Type	Shows the node type; refer to Table 7–27 of a description of the various node types.
Fanout	Shows the number of fan-outs of the element.
Location	Shows the location of the element in the FPGA.
Element	Shows the name of the element.

Table 7–27 provides a description of the possible node type in the report_timing reports.

Table 7–27. Type Description (Part 1 of 2)

Type Name	Description
CELL	Indicates the element is either a register or a combinational element in the FPGA; t.he CELL can be a register in the ALM, memory blocks, DSP blocks, or I/O block
COMP	Indicates the PLL clock network compensation delay.

Table 7–27. Type Description (Part 2 of 2)

Type Name	Description
IC	Indicates the element is an interconnect delay.
utco	Indicates the element's micro clock-to-out time.
utsu	Indicates the element's micro setup time.
uth	Indicates the element's micro hold time.
iext	Indicates the element's external input delay time.
oext	Indicates the element's external output delay time.
LOOP	Indicates a lumped delay bypassing combinational loops.
RE	Indicates a specified routing delay.

report_exceptions

Use the `report_exceptions` command to generate a report that details the slack of all paths that have the timing exceptions `set_false_path`, `set_multicycle`, `set_min_delay`, or `set_max_delay` applied to them.

The `report_exceptions` command can be used to determine if all exceptions have been applied to the applicable paths in the design.

Example 7–33 shows the `report_exceptions` command and options.

Example 7–33. report_exceptions Command

```
report_exceptions
[-append]
[-detail <summary|path_summary|path_only|path_and_clock|full_path>]
[-fall_from_clock <names>]
[-fall_to_clock <names>]
[-file <name>]
[-from <names>]
[-from_clock <names>]
[-hold]
[-less_than_slack <slack limit>]
[-npaths <number>]
[-nworst <number>]
[-pairs_only]
[-panel_name <name>]
[-recovery]
[-removal]
[-rise_from_clock <names>]
[-rise_to_clock <names>]
[-setup]
[-stdout]
[-through <names>]
[-to <names>]
[-to_clock <names>]
```

Table 7–28 describes the options for the `report_exceptions` command.

Table 7–28. report_exceptions Command Options

Option	Description
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-detail <summary|path_only|path_and_clock|full_path>`	Specifies whether or not the clock path detail is reported: `Path Only`: Clock network delay is lumped together `Summary`: Lists each individual path `Path and Clock`: Clock network delay is shown in detail `Full Path`: More clock network details, in particular for generated clocks
`-fall_from_clock <names>`	Specifies the falling edge of the <*names*> from the source register to be analyzed. The options `from_clock`, `fall_from_clock`, and `rise_from_clock` are mutually exclusive.
`-fall_to_clock <names>`	Specifies the falling edge of the <*names*> to the destination register to be analyzed; the options `to_clock`, `fall_to_clock`, and `rise_to_clock` are mutually exclusive.
`-false_path`	Reports only paths that are cut by a false path assignment.
`-file <names>`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type either ***.rpt**, ***.txt**, or ***.html**.
`-hold`	Specifies a clock hold analysis.
`-less_than_slack <slack limit>`	Limits the paths to be reported to the <*slack limit*> value.
`-npaths <number>`	Specifies the number of paths to report.
`-nworst <number>`	Restricts the number of paths per endpoint.
`-panel_name <names>`	Specifies the name of the panel in the **Reports** pane.
`-panel_name <names>`	Sends the results to the panel and specifies the name of the new panel.
`-pairs_only`	When set, paths with the same start and end points are considered equivalent; only the worst-case path for each unique combination is displayed.
`-recovery`	Specifies a recovery analysis.
`-removal`	Specifies a removal analysis.
`-rise_from_clock <names>`	Specifies the rising edge of the <*names*> from the source register to be analyzed; the options `from_clock`, `fall_from_clock`, and `rise_from_clock` are mutually exclusive.
`-rise_to_clock <names>`	Specifies the rising edge of the <*names*> to the destination register to be analyzed; the options `to_clock`, `fall_to_clock`, and `rise_to_clock` are mutually exclusive.
`-setup`	Specifies a clock setup analysis.
`-show_routing`	Displays detailed routing in the path.
`-stdout`	Indicates the report will be sent to `stdout`.
`-through <names>`	Specifies the through node for analysis.
`-to <names>`	Specifies the to node for analysis.
`-to_clock <names>`	Specifies the destination clock for analysis.

report_metastability

Use the `report_metastability` command to generate a report that lists the synchronization register chains for asynchronous transfers in your design, and details the MTBF for synchronization register chains.

To enable metastability analysis, you must set the **Synchronizer Identification** option to identify the synchronization register chains in the design. You can use automatic identification to generate a list of possible synchronizers in the metastability report, but MTBF is not reported for automatically-identified synchronizers.

The TimeQuest Timing Analyzer can analyze metastability MTBF only if a synchronization chain meets its timing requirements. Therefore, it is important for your design to be correctly constrained to get an accurate MTBF report. In addition, automatic synchronizer identification uses timing constraints to automatically detect the signal transfers between circuitry in unrelated or asynchronous clock domains, so clock domains must be related correctly with the timing constraints.

For details about metastability analysis and reporting, refer to the *Managing Metastability with the Quartus II Software* chapter in volume 1 of the *Quartus II Handbook*. This chapter describes how to use the **Synchronizer Identification** option, explains how TimeQuest timing constraints affect synchronizer chain identification and the reported MTBF, and provides details about the information reported with the `report_metastability` command.

Example 7–34. report_metastability command

```
report_metastabiity
[-append]
[-file <name>]
[-panel_name <name>]
[-stdout]
```

Table 7–29 describes the options for the `report_metastability` command.

Table 7–29. report_metastability Command Options

Option	Description
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-file <name>`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type—either ***.txt** or ***.html**.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.
`-stdout`	Indicates the report will be sent to the standard output, via messages. This option is required only if you have selected another output format, such as a file, and would also like to receive messages.

report_clock_transfers

Use the `report_clock_transfers` command to generate a report that details all clock-to-clock transfers in the design. A clock-to-clock transfer is reported if a path exists between two registers that are clocked by two different clocks. Information such as the number of destinations and sources is also reported.

Use the `report_clock_transfers` command to generate a setup, hold, recovery, or removal report.

Example 7–35 shows the `report_clock_transfers` command and options.

Example 7–35. report_clock_transfers Command

```
report_clock_transfers
[-append]
[-file <name>]
[-hold]
[-setup]
[-stdout]
[-recovery]
[-removal]
[-panel_name <name>]
```

Table 7–30 describes the options for the `report_clock_transfers` command.

Table 7–30. report_clock_transfers Command Options

Option	Description
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-file <name>`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type either ***.txt** or ***.html**.
`-hold`	Creates a clock transfer summary for hold analysis.
`-setup`	Creates a clock transfer summary for setup analysis.
`-stdout`	Indicates the report will be sent to `stdout`.
`-recovery`	Creates a clock transfer summary for recovery analysis.
`-removal`	Creates a clock transfer summary for removal analysis.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.

report_clocks

Use the `report_clocks` command to generate a report that details all clocks in the design. The report contains information such as type, period, waveform (rise and fall), and targets for all clocks in the design.

Example 7–36 shows the `report_clocks` command and options.

Example 7–36. report_clocks Command

```
report_clocks
[-append]
[-desc]
[-file <name>]
[-stdout]
[-panel_name <name>]
```

Table 7–31 describes the options for the `report_clocks` command.

Table 7–31. report_clocks Command Options

Option	Description
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-file <name>`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type—either ***.txt** or ***.html**.
`-desc`	Specifies the clock names to sort in descending order. The default is ascending order.
`-stdout`	Indicates the report will be sent to `stdout`.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.

report_min_pulse_width

The `report_min_pulse_width` command checks that a clock high or low pulse is sustained long enough to be recognized as an actual change in the clock signal. A failed minimum pulse width check indicates that the register may not recognize the clock transition. Use the `report_min_pulse_width` command to generate a report that details the minimum pulse width for all clocks in the design. The report contains information for high and low pulses for all clocks in the design.

The `report_min_pulse_width` command also reports minimum period checks for RAM and DSP, as well as I/O edge rate limits for input and output clock ports. For output ports, the port must either have a clock (or generated clock) assigned to it or used as the `-reference_pin` for input/output delays.

The `report_min_pulse_width` command checks the I/O edge rate limits, but does not always perform the check for output clock ports. For the `report_min_pulse_width` command to check the I/O edge rate limits for output clock ports, the output clock port must fall into one of the following categories:

- Have a clock or generated clock constraint assigned to it

or

- Use a `-reference_pin` for an input or output delay constraint

Each register in the design is reported twice per clock that clocks the register: once for the high pulse and once for the low pulse. Example 7–37 shows the `report_min_pulse_width` command and options.

Example 7–37. report_min_pulse_width Command

```
report_min_pulse_width
[-append]
[-file <name>]
[-nworst <number>]
[-stdout]
[<targets>]
[-panel_name <name>]
```

Table 7–32 describes the options for the `report_min_pulse_width` command.

Table 7–32. report_min_pulse_width Command Options

Option	Description
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-file <name>`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type—either ***.txt** or ***.html**.
`-nworst <number>`	Specifies the number of pulse width checks to report. The default is 1.
`-stdout`	Redirects the output to `stdout` via messages; only required if another output format, such as a file, has been selected and is also to receive messages.
`<targets>`	Specifies registers.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.

report_net_timing

Use the `report_net_timing` command to generate a report that details the delay and fan-out information about a net in the design. A net corresponds to a cell's output pin.

Example 7–38 shows the `report_net_timing` command and options.

Example 7–38. report_net_timing Command

```
report_net_timing
[-append]
[-file <name>]
[-nworst_delay <number>]
[-nworst_fanout <number>]
[-stdout]
[-panel_name <name>]
```

Table 7–33 describes the options for the `report_net_timing` command.

Table 7–33. report_net_timing Command Options

Option	Description
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-file <name>`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type—either ***.txt** or ***.html**.
`-nworst_delay <number>`	Specifies that <*number*> worst net delays be reported.
`-nworst_fanout <number>`	Specifies that <*number*> worst net fan-outs be reported.
`-stdout`	Indicates the report will be sent to `stdout`.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.

 © March 2009 Altera Corporation

report_sdc

Use the `report_sdc` command to generate a report of all the Synopsys design constraints in the project.

Example 7–39 shows the `report_sdc` command and options.

Example 7–39. report_sdc Command

```
report_sdc
[-ignored]
[-append]
[-file]
[-stdout]
[-panel_name <name>]
```

Table 7–34 describes the options for the `report_sdc` command.

Table 7–34. report_sdc Command Options

Option	Description
`-ignored`	Reports assignments that were ignored.
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-file`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type—either ***.txt** or ***.html**.
`-stdout`	Indicates that the report will be sent to `stdout`.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.

report_ucp

Use the `report_ucp` command to generate a report of all unconstrained paths in the design.

Example 7–40 shows the `report_ucp` command and options.

Example 7–40. report_ucp Command

```
report_ucp
[-append]
[-file <name>]
[-hold]
[-setup]
[-stdout]
[-summary]
[-panel_name <name>]
```

Table 7–35 describes the options for the `report_ucp` command.

Table 7–35. Option Descriptions for report_ucp (Part 1 of 2)

Option	Description
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-file <name>`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type—either ***.txt** or ***.html**.

Table 7–35. Option Descriptions for report_ucp (Part 2 of 2)

Option	Description
`-hold`	Reports all unconstrained hold paths.
`-setup`	Reports all unconstrained setup paths.
`-stdout`	Indicates the report be sent to `stdout`.
`-summary`	Generates only the summary panel.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.

Table 7–36 summarizes all reporting commands available in the Quartus II TimeQuest Timing Analyzer.

Table 7–36. Reports from the Tasks Pane and Tcl Commands

Task Pane Report	Tcl Command	Description
Report Setup Summary	`create_timing_summary -setup`	Generates a clock setup summary for all defined clocks.
Report Hold Summary	`create_timing_summary -hold`	Generates a clock hold summary for all defined clocks.
Report Recovery Summary	`create_timing_summary -recovery`	Generates a clock recovery summary for all defined clocks.
Report Removal Summary	`create_timing_summary -removal`	Generates a clock removal summary for all defined clocks.
Report Clocks	`report_clocks`	Generates a clock summary for all defined clocks.
Report Clock Transfers	`report_clock_transfers`	Generates a clock transfer summary for all clock-to-clock transfers in the design.
Report SDC	`report_sdc`	Generates a summary of all **.sdc** file commands read.
Report Unconstrained Paths	`report_ucp`	Generates a summary of all unconstrained paths in the design.
Report Timing	`report_timing`	Generates a detailed summary for specific paths in the design.
Report Net Timing	`report_net_timing`	Generates a detailed summary for specific nets in the design.
Report Minimum Pulse Width	`report_min_pulse_width`	Generates a detailed summary for specific registers in the design.
Create Slack Histogram	`create_slack_histogram`	Generates a detailed histogram for a specific clock in the design.

report_bottleneck

Use the `report_bottleneck` command to report a rating per node based on the number of failing paths through each node for the worst 1,000 setup paths. Example 7–41 shows the `report_bottleneck` command and options.

Example 7–41. report_bottleneck Command

```
report_bottleneck
[-cmetric <cmetric_name>]
[-details]
[-metric <default|tns|num_paths|num_fpaths|num_fanins|num_fanouts>]
[-panel <panel_name>]
[-stdout]
[<paths>]
```

By default, the report_bottleneck command reports a rating for the worst 1,000 setup paths.

In addition to the default metric, there are a few additional "standard" metrics to choose from, such as:

- -metric num_fanouts
- -metric tns

You can also create a custom metric to evaluate the nodes based on the combination of the number of fanouts, fanins, failing paths, total paths, and other keepers. The paths to be analyzed can be specified by passing the result of any get_timing_paths call as the last argument to report_bottleneck.

Table 7–37 describes the options for the report_bottleneck command.

Table 7–37. report_bottleneck Command

Option	Description
-cmetric <*cmetric_name*>	Custom metric function to evaluate individual nodes.
-details	Show the detailed information (number of failing edges, number of fan-ins, and so forth).
-metric <*default*|*tns*|*num_paths*|*num_fpaths*|*num_fanins*|*num_fanouts*>	Indicate the metric to use to rate individual nodes.
-panel <*panel_name*>	Sends the results to the panel and specifies the name of the new panel.
-stdout	Indicates the report will be sent to stdout.
<*paths*>	Paths to be analyzed.

Example 7–42 shows how to create a custom metric with the report_bottleneck command.

Example 7–42. report_bottleneck Custom Metric

```
#set the number of paths to be reported
set paths [ get_timing_paths -npaths 1000 -setup ]

#create the custom metric
proc report_bottleneck_custom_metric {arg} {
   # Description: use the number of fanins as the custom metric.
   upvar $arg metric
   set rating $metric(num_fanins)
   return $rating
}
#reporting the results of the custom metric
report_bottleneck -cmetric report_bottleneck_custom_metric -panel
"Timing Analysis Bottleneck Report - Custom" $paths
```

report_datasheet

Use the `report_datasheet` command to generate a datasheet report which summarizes the timing characteristics of the entire design. It reports setup (`tsu`), hold (`th`), clock-to-output (`tco`), minimum clock-to-output (`mintco`), propagation delay (`tpd`), and minimum propagation delay (`mintpd`) times. Example 7–43 shows the `report_datasheet` command and options.

Example 7–43. report_datasheet Command

```
report_datasheet
[-append]
[-file <name>]
[-stdout]
[panel_name <name>]
```

Table 7–38 describes the options for the `report_datasheet` command.

Table 7–38. report_datasheet Command Options

Option	Description
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-file <name>`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type—either ***.txt** or ***.html**.
`-stdout`	Indicates the report will be sent to `stdout`.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.

The delays are reported with respect to a base clock or port for which they are relevant. If there is a case where there are multiple paths for a clock, the maximum delay of the longest path is taken for the t_{SU}, t_H, t_{CO}, and t_{PD}, and the minimum delay of the shortest path is taken for $mint_{CO}$ and $mint_{PD}$.

report_rskm

Use the `report_rskm` command to generate a report that details the receiver skew margin for LVDS receivers.

Example 7–44 shows the `report_rskm` command and options.

Example 7–44. report_rskm Command

```
report_rskm
[-append]
[-file <name>]
[-panel_name <name>]
[-stdout]
```

Table 7–39 describes the options for the `report_rskm` command.

Table 7–39. report_rskm Command Options

Type Name	Description
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-file <name>`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type—either ***.txt** or ***.html**.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.
`-stdout`	Indicates the report will be sent to `stdout`.

The receiver input skew margin (RSKM) is the time margin available before the LVDS receiver megafunction fails to operate. RSKM is defined as the total time margin that remains after subtracting the sampling window (SW) size and the receiver channel-to-channel skew (RCCS) from the time unit interval (TUI), as expressed in the formula shown in Equation 7–11:

Equation 7–11.

$$RSKM = \frac{(TUI - SW - RCCS)}{2}$$

The time unit interval is the LVDS clock period ($1/f_{MAX}$). The sampling window is the period of time that the input data must be stable to ensure that the data is successfully sampled by the LVDS receiver megafunction. The sampling window size varies by device speed grade; RCCS reflects channel-to-channel skew seen by the LVDS receiver. This RCCS includes transmitter channel-to-channel skew (TCCS) of the upstream transmitter and maximum channel-to-channel skew between the transmitter and receiver. RCCS is equal to the difference between the maximum input delay and minimum input delay. If no input delay is set, RCCS defaults to zero.

report_tccs

Use the `report_tccs` command to generate a report that details the channel-to-channel skew margin for LVDS transmitters.

Example 7–45 shows the `report_tccs` command and options.

Example 7–45. report_tccs Command

```
report_tccs
[-append]
[-file <name>]
[-panel_name <name>]
[-quiet]
[-stdout]
```

Table 7–40 describes the options for the `report_tccs` command.

Table 7–40. report_tccs Command Options

Type Name	Description
`-append`	Specifies that the current report be appended to the file specified by the `-file` option.
`-file <name>`	Indicates that the current report is written to the file <*name*>.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.
`-quiet`	Specifies that nothing is printed if there are no LVDS receivers in the design.
`-stdout`	Indicates the report will be sent to `stdout`.

The TCCS is the timing difference between the fastest and slowest output transitions, including t_{CO} variations and clock skew.

report_partitions

Use the `report_partitions` command to generate a timing report listing the worst-case setup checks for each partition in the design.

Example 7–46 shows the `report_partitions` command and options.

Example 7–46. report_partitions Command

```
report_partitions
[-nworst <number>]
[-panel_name <name>]
[-stdout]
```

Table 7–41 describes the options for the `report_partitions` command.

Table 7–41. report_partitions Command Options

Type Name	Description
`-nworst`	Specifies the maximum number of paths to report for each endpoint.
`-panel_name`	Sends the results to the panel and specifies the name of the new panel.
`-stdout`	Indicates the report will be sent to `stdout`.

report_path

Use the `report_path` command to generate a report that details the longest delay paths between any two arbitrary keeper nodes.

Example 7–47 shows the `report_path` command and options.

Example 7–47. report_path Command

```
report_path
[-append]
[-file <name>]
[-from <names>]
[-min_path]
[-npaths <number>]
[-nworst <number>]
[-panel_name <name>]
[-stdout]
[-summary]
[-through <names>]
[-to <names>]
```

Table 7–42 describes the options for the `report_path` command.

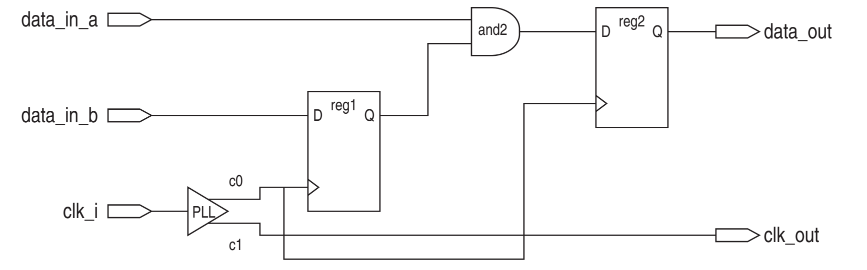
Table 7–42. report_path Command Options

Type Name	Description
`-append`	Specifies that the current report be appended to the file specified by the `-file` option.
`-file <name>`	Indicates that the current report is written to the file <*name*>.
`-from <names>`	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the start point of the path.
`-min_path`	Displays the minimum delay paths.
`-npaths <number>`	Specifies the number of paths to report.
`-nworst <number>`	Specifies the maximum number of paths to report for each endpoint.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.
`-stdout`	Indicates the report will be sent to `stdout`.
`-summary`	Creates a single table with a summary of each path found.
`-through <names>`	The <*names*> is a collection or list of objects in the design. Specifies false path passes through <*names*>.
`-to <names>`	The <*names*> is a collection or list of objects in the design. The <*names*> acts as the end point of the path.

The delay path reported cannot pass through a keeper node; for example, a register or port. Instead, the delay path must be from the output pin of a keeper node to the input pin of a keeper node.

Figure 7–34 shows a simple design with a register-to-register path.

Figure 7–34. Simple Register-to-Register Path



Example 7–48 shows the report generated from the following command:

```
report_path -from [get_pins {reg1|regout}] -to [get_pins \
{reg2|datain}] -npaths 1 -panel_name "Report Path" -stdout
```

Example 7–48. report_path from Keeper Output Pin to Keeper Input Pin

```
 Info: ===================================================================
Info: From Node : reg1|regout
Info: To Node : reg2|datain
Info:
Info: Path:
Info:
Info: Total (ns) Incr (ns) Type Element
Info: ========== ========= == ==== ====================
Info: 0.000 0.000 reg1|regout
Info: 0.206 0.206 RR IC and2|datae
Info: 0.360 0.154 RR CELL and2|combout
Info: 0.360 0.000 RR IC reg2|datain
Info:
Info: Total Path Delay : 0.360
Info: ===================================================================
```

Example 7–49 shows the report generated from the following command:

```
report_path -from [get_ports data_in_a] -to [get_pins {reg2|regout}] \
-npaths 1
```

Example 7–49. report_path from Keeper-to-Keeper Output Pin

```
Info: Report Path: No paths were found
0 0.000
```

No paths were reported in Example 7–49 because the destination passes through an input pin of a keeper node.

report_net_delay

Use the `report_net_delay` to generate a slack report for paths constrained with the `set_net_delay` command. The `report_net_delay` command reports the results of all `set_net_delay` commands in a single report. The report contains each `set_net_delay` command with the worst case slack result followed by the results of each edge matching the criteria set by that `set_net_delay` command. These results are ordered based on the slack value. Example 7–50 shows the `report_net_delay` command and options.

Example 7–50. report_net_delay Command

```
report_net_delay
    [-append]
    [-file <name>]
    [-nworst <number>]
    [-panel_name <name>]
    [-stdout]
```

Table 7–43 describes the options for the `report_net_delay` command.

Table 7–43. report_net_delay Command Options

Options	Description
`-append`	If output is sent to a file, this option appends the result to that file. Otherwise, the file is overwritten.
`-file <name>`	Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type—either ***.txt** or ***.html**.
`-nworst <number>`	Specifies the maximum number of paths to report for each analysis. If unspecified, there is no limit.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.
`-stdout`	Send output to `stdout`, via messages. You only have to use this option if you have selected another output format, such as a file, and would also like to receive messages.

report_max_skew

Use the `report_max_skew` to generate a slack report for paths constrained with the `set_max_skew` command. The `report_max_skew` command reports the results of all `set_max_skew` commands in a single report. The report contains each `set_max_skew` command with the worst case slack result followed by the results of each edge matching the criteria set by that `set_max_skew` command. These results are ordered based on the slack value. Example 7–51 shows the `report_max_skew` command and options.

Example 7–51. report_max_skew Command

```
report_max_skew
[-detail <summary|path_only|path_and_clock|full_path>]
[-file <name>]
[-less_than_slack <slack limit>]
[-npaths <number>]
[-panel_name <name>]
[-show_routing]
[-stdout]
```

Table 7–44 describes the options for the report_max_skew command.

Table 7–44. report_max_skew Command Options

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>[-detail <<i>summary|path_only|path_and_clock|full_path</i>>]</td><td>Specifies whether or not the clock path detail is reported:
Path Only: Clock network delay is lumped together
Summary: Lists each individual path
Path and Clock: Clock network delay is shown in detail
Full Path: More clock network details, in particular for generated clocks</td></tr>
<tr><td>[-file <<i>name</i>>]</td><td>Sends the results to an ASCII or HTML file. The extension specified in the file name determines the file type—either *.txt or *.html.</td></tr>
<tr><td>[-less_than_slack <<i>slack limit</i>>]</td><td>Limits the paths reported to the <<i>slack limit</i>> value.</td></tr>
<tr><td>[-npaths <<i>number</i>>]</td><td>Specifies the number of paths to report.</td></tr>
<tr><td>[-panel_name <<i>name</i>>]</td><td>Sends the results to the panel and specifies the name of the new panel.</td></tr>
<tr><td>[-show_routing]</td><td>Displays detailed routing in the path.</td></tr>
<tr><td>[-stdout]</td><td>Indicates the report will be sent to stdout.</td></tr>
</table>

No results are displayed if the -from/-from_clock and -to/-to_clock are applied to less than two paths.

check_timing

Use the check_timing command to generate a report on any potential problem with the design or applied constraints. Not all check_timing results are serious issues. The results should be examined to see if the results are desired. Example 7–52 shows the check_timing command and options.

Example 7–52. check_timing Command

```
check_timing
[-append]
[-file <name>]
[-include <check_list>]
[-stdout]
[-panel_name <name>]
```

Table 7–45 describes the options for the check_timing command.

Table 7–45. check_timing Command Options

Option	Description
-append	Specifies that the current report be appended to the file specified by the -file option.
-file <*name*>	Indicates that the current report is written to the file <*name*>.
-include	Indicates that a check is to be performed with the <*check_list*>. Refer to Table 7–46 for a list of checks.
-stdout	Indicates the report will be sent to stdout.
-panel_name <*name*>	Sends the results to the panel and specifies the name of the new panel.

Table 7–46 describes the possible checks.

Table 7–46. Possible Checks (Part 1 of 2)

Option	Description
no_clock	Checks that registers have at least one clock at their clock pin and ensures that ports determined to be clocks have a clock assigned to them.
multiple_clock	Checks that registers have at most one clock at their clock pin. When multiple clocks reach a register's clock pin, both clocks are used for analysis.
generated_clock	Checks that generated clocks are valid. Generated clocks must have a source that is clocked by a valid clock. They must also not depend on each other in a loop (clk1 cannot have clk2 as a source if clk2 already uses clk1 as a source).
no_input_delay	Checks that every input port that is not determined to be a clock has an input delay set on it.
no_output_delay	Checks that every output port has an output delay set on it.
partial_input_delay	Checks that input delays are complete. Makes sure that input delays have a rise-min, fall-min, rise-max, and fall-max portion set.
partial_output_delay	Checks that output delays are complete. Makes sure that output delays have a rise-min, fall-min, rise-max, and fall-max portion set.
reference_pin	Checks if reference pins specified in set_input_delay and set_output_delay using the reference_pin option are valid. A reference_pin is valid if the clock option specified in the same set_input_delay/set_output_delay command matches the clock that is in the direct fan-in of the reference_pin. Being in the direct fan-in of the reference_pin means that there must be no keepers between the clock and reference_pin.
latency_override	Ensures that the clock latency constraint applied on a port or pin overrides the more generic clock latency set on a clock. Clock latency set to a clock applies to all keepers clocked by the clock. Clock latency set to a pin or a port applies to registers in the fan-out of the port or pin.
loops	Checks that there are no strongly connected components in the design. These loops prevent a design from being properly analyzed. Indicates that loops exist but were marked so that they would not be traversed.
latches	Checks whether there are latches in the design. The Quartus II TimeQuest Timing Analyzer warns the user that the latches exist and cannot be properly analyzed.
pos_neg_clock_domain	Checks whether any register is clocked by both the rising and falling edges of the same clock. if this scenario is necessary, such as in a clock multiplexer, create two separate clocks that have similar settings and are assigned to the same node.

Table 7–46. Possible Checks (Part 2 of 2)

Option	Description
pll_cross_check	Checks the clocks that are assigned to a PLL against the PLL setting defined in the user's design files. Inconsistent setting or an unmatched number of clocks associated with the PLL are reported to the user.
no_uncertainty	Checks that each clock-to-clock transfer has a clock uncertainty assignment set between the two clocks.
virtual_clock	Checks that each virtual clock is referenced.
partial_multicycle	Checks that each setup multicycle assignment has a corresponding hold multicycle assignment, and each hold multicycle assignment has a corresponding setup multicyle assignment.
multicycle_consistency	Checks that all of the multicycle cases in which a setup multicycle does not equal one greater than the hold multicycle. Hold multicycle assignments are usually one cycle fewer than setup multicycle assignments.
partial_min_max_delay	Verifies that each minimum delay assignment has a corresponding maximum delay assignment, and that each maximum delay assignment has a corresponding minimum delay assignment.
clock_assignments_ on_output_ports	Checks that reports all of the clock assignments that have been applied to output ports.
generated_io_delay	Checks that all of the I/O delays with no reference pin, generated -clock, or no -source_atencey_included.

Example 7–53 shows how the check_timing command can be used.

Example 7–53. The check_timing Command

```
# Constrain design
create_clock -name clk -period 4.000 -waveform { 0.000 2.000 } \
[get_ports clk]
set_input_delay -clock clk2 1.5 [get_ports in*]
set_output_delay -clock clk 1.6 [get_ports out*]
set_false_path -from [get_keepers in] -through [get_nets r1] -to \
[get_keepers out]

# Check if there were any problems for combinational loops, latches, or
# misssing or incomplete input delays
check_timing -include {loops latches no_input_delay
partial_input_delay}
```

report_clock_fmax_summary

Use the report_clock_fmax_summary to report potential f_{MAX} for every clock in the design, regardless of the user-specified clock periods. f_{MAX} is only computed for paths where the source and destination registers or ports are driven by the same clock. Paths of different clocks, including generated clocks, are ignored. For paths between a clock and its inversion, f_{MAX} is computed as if the rising and falling edges are scaled along with f_{MAX}, such that the duty cycle (in terms of a percentage) is maintained.

Example 7–54 shows the `report_clock_fmax_summary` command and options.

Example 7–54. report_clock_fmax_summary Command

```
report_clock_fmax_summary
[-append]
[-file <name>]
[-panel_name <name>]
[-stdout]
```

Table 7–47 describes the options for the `report_clock_fmax_summary` command.

Table 7–47. report_clock_fmax_summary Command Options

Option	Description
`-append`	Specifies that the current report be appended to the file specified by the `-file` option.
`-file <name>`	Indicates that the current report is written to the file *<name>*.
`-stdout`	Indicates the report will be sent to `stdout`.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.

The f_{MAX} Summary report contains four columns: f_{MAX}, Restricted f_{MAX}, Clock Name, and Note. The description of each column is given in Table 7–48.

Table 7–48. f_{MAX} Summary Report Column

Column Name	Description
f_{MAX}	Shows the fastest possible frequency at which the internal register-to-register can run. This is not the fastest the clock port can be driven.
Restricted f_{MAX}	Shows fastest possible frequency at which the clock port can run. This number may be lower than the f_{MAX} column for various reasons, including hold timing requirements, I/O edge rate limits for clocks (which also depends on I/O standards), minimum pulse width checks for registers, and minimum period checks for RAM and DSP registers.
Clock Name	Shows the clock name.
Note	Shows any notes related to the clock.

create_timing_summary

Reports the worst-case clock setup and clock hold slacks and endpoint total negative slack (TNS) per clock domain. Total negative slack is the sum of all slacks less than zero for each destination register or port in the clock domain.

Example 7–55 shows the `create_timing_summary` command and options.

Example 7–55. create_timing_summary Command

```
create_timing_summary
[-append]
[-file <name>]
[-hold]
[-panel_name <name>]
[-recovery]
[-removal]
[-setup]
[-stdout]
```

Table 7–49 describes the options for the `create_timing_summary` command.

Table 7–49. create_timing_summary Command Options

Option	Description
`-append`	Specifies that the current report be appended to the file specified by the `-file` option.
`-file <name>`	Indicates that the current report is written to the file *<name>*.
`-hold`	Generates a clock hold check summary report.
`-panel_name <name>`	Sends the results to the panel and specifies the name of the new panel.
`-recovery`	Generates a recovery check summary report.
`-removal`	Generates a removal check summary report.
`-setup`	Generates a clock setup check summary report.
`-stdout`	Indicates the report will be sent to `stdout`.

Timing Analysis Features

The TimeQuest Timing Analyzer supports many features that enhances and provides a through analysis of your designs. This section covers the many features available in the TimeQuest Timing Analyzer.

Multi-Corner Analysis

Multi-corner analysis allows a design to be verified under a variety of operating conditions (voltage, process, and temperature) while performing a static timing analysis on the design.

Use the `set_operating_conditions` command to change the operating conditions or speed grade of the device used for static timing analysis.

Example 7–56 shows the `set_operating_conditions` command and options.

Example 7–56. set_operating_conditions Command

```
set_operating_conditions
[-model <fast|slow>]
[-speed <speed grade>]
[-temperature <value in °C>]
[-voltage <value in mV>]
[<operating condition Tcl object>]
```

Table 7–50 describes the options for the `set_operating_conditions` command.

Table 7–50. set_operating_conditions Command Options

Option	Description
`-model <fast|slow>`	Specifies the timing model.
`-speed <speed grade>`	Specifies the device speed grade.
`-temperature <value in °C>`	Specifies the operating temperature.
`-voltage <value in mV>`	Specifies the operating voltage.
`<operating condition Tcl object>`	Specifies the operating condition Tcl object that specifies the operating conditions.

If an operating condition Tcl object *is* used, the model, speed, temperature, and voltage options are not required. If an operating condition Tcl object *is not* used, the model must be specified, and the -speed, -temperature, and -voltage options are optional, using the appropriate defaults for the device where applicable.

Table 7–51 shows a few of the available operating conditions that can be set for each device family.

Table 7–51. Device Family Operating Conditions

Device Family	Available Conditions				Operating Condition Tcl Objects
	Speed Grade	Model	Voltage (mV)	Temp (°C)	
Stratix III	4	Slow Slow Fast	1100 1100 1100	85 0 0	`4_slow_1100mv_85c` `4_slow_1100mv_0c` `MIN_fast_1100mv_0c`
Cyclone® III	7	Slow Slow Fast	1200 1200 1200	85 0 0	`7_slow_1200mv_85c` `7_slow_1200mv_0c` `MIN_fast_1200mv_0c`
Stratix II	4	Slow Fast	—	—	`4_slow` `MIN_fast`
Cyclone II	6	Slow Fast	—	—	`6_slow` `MIN_fast`

Use the `get_available_operating_conditions-all` command to obtain a list of available operating conditions for the target device.

Table 7–52 shows the operating conditions of each model for device families that support only two operating conditions, that is, fast and slow.

Table 7–52. Operating Conditions for Fast and Slow Models

Model	Speed Grade	Voltage	Temperature
Slow	Slowest speed grade in device density	V_{cc} minimum supply *(1)*	Maximum T_J *(1)*
Fast	Fastest speed grade in device density	V_{cc} maximum supply *(1)*	Minimum T_J *(1)*

Note to Table 7–52:

(1) Refer to the DC & Switching Characteristics chapter of the applicable device Handbook for V_{CC} and T_J.

A static timing analysis should be performed under all available operating conditions. This ensures that no violations will occur under various conditions during the device operation.

Example 7–57 shows how to set the operating conditions for a Stratix III design to the slow model, 1100 mV, and 85° C.

Example 7–57. Setting Operating Conditions with Individual Values

```
set_operating_conditions -model slow -temperature 85 -voltage 1100
```

Alternatively, you can set the operating conditions in Example 7–57 with the Tcl object as shown in Example 7–58.

Example 7–58. Setting Operating Conditions with a Tcl Object

```
set_operating_conditions 4_slow_1100mv_85c
```

Advanced I/O Timing and Board Trace Model Assignments

The Quartus II TimeQuest Timing Analyzer is able to use Advanced I/O Timing and Board Trace Model assignments to model I/O buffer delays in your design.

To turn the Advanced I/O feature on or off, in the **Settings** dialog box, under the **TimeQuest Timing Analyzer** option, choose **on** or **off**.

If you turn the **Advanced I/O Timing** on or off or change Board Trace Model assignments and do not recompile before you analyze timing, you must use the `-force_dat` command when you create the timing netlist. Type the following command in the Tcl console of the Quartus II TimeQuest Timing Analyzer:

```
create_timing_netlist -force_dat ↵
```

If you turn the **Advanced I/O Timing** or change Board Trace Model assignments on or off and recompile before you analyze timing, you do not have to use the `-force_dat` command when you create the timing netlist. You can create the timing netlist with the `create_timing_netlist` command, or with the **Create Timing Netlist** task in the **Tasks** pane.

For more information about the Advanced I/O Timing feature, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Wildcard Assignments and Collections

To simplify the task of applying constraints to many nodes in a design, the Quartus II TimeQuest Timing Analyzer accepts the "*" and "?" wildcard characters. Use these wildcard characters to reduce the number of individual constraints you must specify in your design.

The "*" wildcard character matches any string. For example, given an assignment made to a node specified as `reg*`, the Quartus II TimeQuest Timing Analyzer searches for and applies the assignment to all design nodes that match the prefix `reg` with none, one, or several characters following, such as `reg1`, `reg[2]`, `regbank`, and `reg12bank`.

The "?" wildcard character matches any single character. For example, given an assignment made to a node specified as `reg?`, the Quartus II TimeQuest Timing Analyzer searches and applies the assignment to all design nodes that match the prefix `reg` and any single character following; for example, `reg1`, `rega`, and `reg4`.

Both the collection commands `get_cells` and `get_pins` have three options that allow you to refine searches that include the wildcard character. To refine your search results, select the default behavior, the `-hierarchical` option, or the `-compatibility` option.

The pipe character is used to separate one hierarchy level from the next in the Quartus II TimeQuest Timing Analyzer. For example, <*absolute full cell name*>|<*pin suffix*> represents a hierarchical pin name with the "|" separating the hierarchy from the pin name.

When you use the collection commands `get_cells` and `get_pins` without an option, the default search behavior is performed on a per-hierarchical level of the pin name; that is, the search is performed level by level. A full hierarchical name may contain multiple hierarchical levels where a "|" is used to separate the hierarchical levels, and each wildcard character represents only one hierarchical level. For example,"*" represents the first hierarchical level and "*|*" represents the first and second hierarchical levels.

When you use the collection commands `get_cells` and `get_pins` with the `-hierarchical` option, a recursive match is performed on the relative hierarchical path name of the form <*short cell name*>|<*pin name*>. The search is performed on the node name; for example, the last hierarchy of the name and not the hierarchy path. Unlike the default behavior, this option does not limit the search to each hierarchy level represented by the pipe character.

The pipe character cannot be used in the search with the `get_cells -hierarchical` option. However, the pipe character can be used with the `get_pins` collection search.

When you use the collection commands `get_cells` and `get_pins` with the `-compatibility` option, the search performed is similar to that of the Quartus II Classic Timing Analyzer. This option searches the entire hierarchical path and pipe characters are not treated as special characters.

Assuming the following cells exist in a design:

```
foo
foo|bar
```

and the following pin names:

```
foo|dataa
foo|datab
foo|bar|datac
foo|bar|datad
```

Table 7–53 shows the results of using these search strings.

Table 7–53. Sample Search Strings and Search Results (Part 1 of 2)

Search String	Search Result
`get_pins *|dataa`	`foo|dataa`
`get_pins *|datac`	<*empty*>
`get_pins *|*|datac`	`foo|bar|datac`
`get_pins foo*|*`	`foo|dataa, foo|datab`
`get_pins -hierarchical *|*|datac`	<*empty*> *(1)*
`get_pins -hierarchical foo|*`	`foo|dataa, foo|datab`
`get_pins -hierarchical *|datac`	`foo|bar|datac`
`get_pins -hierarchical foo|*|datac`	<*empty*> *(1)*

Table 7–53. Sample Search Strings and Search Results (Part 2 of 2)

Search String	Search Result
`get_pins -compatibility *|datac`	`foo|bar|datac`
`get_pins -compatibility *|*|datac`	`foo|bar|datac`

Note to Table 7–53:

(1) Due to the additional `*|*|` in the search string, the search result is <*empty*>.

Resetting a Design

Use the `reset_design` command to remove all timing constraints and exceptions from the design under analysis. The command removes all clocks, generated clocks, derived clocks, input delays, output delays, clock latency, clock uncertainty, clock groups, false paths, multicycle paths, min delays, and max delays.

This command provides a convenient way to return to the initial state of analysis without the need to delete and re-create a new timing netlist.

Cross-Probing

The cross-probing feature allows you to locate paths and elements from the TimeQuest Timing Analyzer to various tools available in the Quartus II software (and vice versa).

From the TimeQuest GUI, you can right-click any path in the **View** pane and select either **Locate Path** or **Locate**.

The source is the element in the **From Node** column and the destination is the element in the **To Node** column.

The **Locate Path** option allows you to located the data arrival path, default, of the currently selected row. To locate the data required time path select a row in the data required path panel.

The **Locate Required Path** command is available only when there is a path to show; unless the user reports the clock path as well, there is probably only a single node in the required path. In this case, the command is not available.

The **Locate** option allows you to locate the highlighted element.

The **Locate Path** and **Locate** commands can cross-probe to either the Chip Planner, Technology Map Viewer, or Resource Property Editor. Additionally, the **Locate Path** option can cross-probe to Critical Path Settings.

From the **Critical Path Settings** dialog box in the Chip Planner, you can cross-probe to the TimeQuest Timing Analyzer to report critical paths in the design.

locate

Use the `locate` command in the **Console** pane to cross-probe to the Chip Editor, Critical Path Settings, Resource Property Editor, and the Technology Map Viewer.

Example 7–59 shows the `locate` command and options.

Example 7–59. locate Command

```
locate
[-chip]
[-color <black|blue|brown|green|grey|light_grey|orange|purple|red|white>]
[-cps]
[-label <label>]
[-rpe]
[-tmv]
<items>
```

Table 7–54 describes the options for the `locate` command.

Table 7–54. locate Options

Option	Description
`-chip`	Locates the object in the Chip Planner.
`-color <black|blue|brown|green|grey|light_grey|orange|purple|red|white>`	Identifies the objects you are locating.
`-cps`	Locates the object in the Critical Path Settings dialog of the Chip Planner.
`-label <label>`	Specifies a label used to identify the objects you are locating.
`-rpe`	Locates in the Resource Property Editor.
`-tmv`	Locates the object in the Technology Map Viewer.
`<items>`	Items to locate. Any collection or object (such as paths, points, nodes, nets, keepers, registers, etc.) may be located by passing a reference to the corresponding collection or object.

Example 7–60 shows how to cross-probe ten paths from TimeQuest Timing Analyzer to the Chip Editor and locate all data ports in the Technology Map Viewer.

Example 7–60. Cross-probing from TimeQuest

```
# Locate all of the nodes in the longest ten paths
# into the Chip Editor
locate [get_path -npaths 10] -chip

# locate all ports that begin with data to the Tech Map Viewer

locate [get_ports data*] -tmv
```

The TimeQuest Timing Analyzer GUI

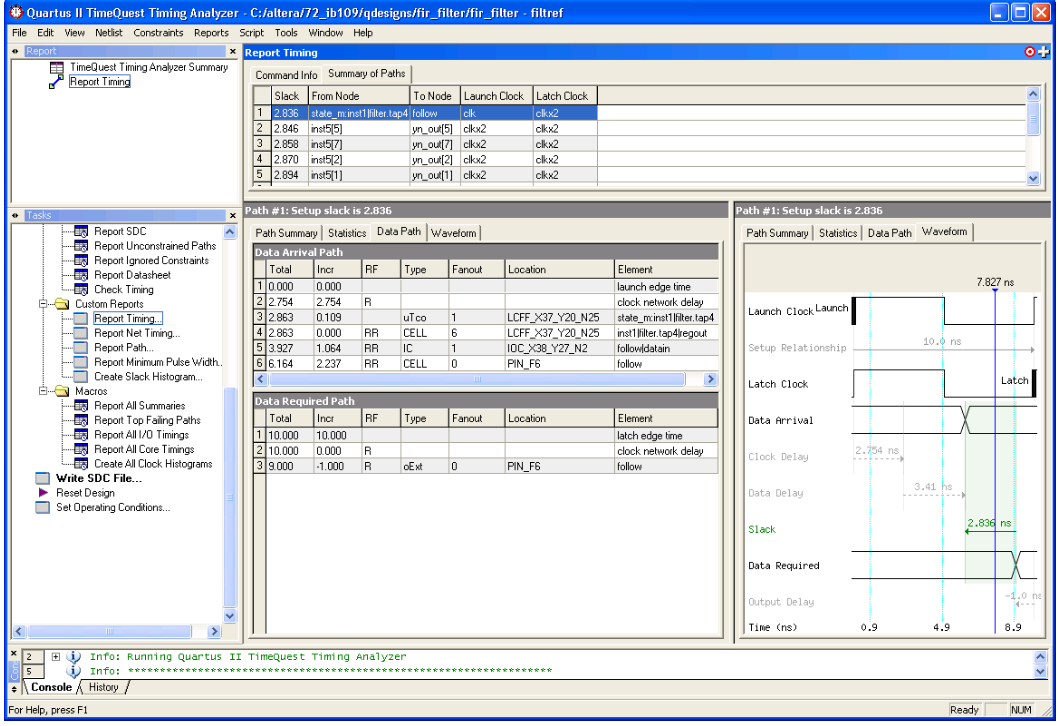
The Quartus II TimeQuest Timing Analyzer provides an intuitive and easy-to-use GUI that allows you to efficiently constrain and analyze your designs. The GUI consists of the following panes:

- "The Quartus II Software Interface and Options" described on page 7–80
- "View Pane" described on page 7–81
- "Tasks Pane" described on page 7–83

- "Console Pane" described on page 7–85
- "Report Pane" described on page 7–85
- "Constraints" described on page 7–85
- "Name Finder" described on page 7–87
- "Target Pane" described on page 7–88
- "SDC Editor" described on page 7–89

Each pane provides features that enhance productivity (Figure 7–35).

Figure 7–35. The TimeQuest GUI



The Quartus II Software Interface and Options

The Quartus II software allows you to configure various options for the Quartus II TimeQuest Timing Analyzer report generation that are generated in the Compilation Report for the design.

The TimeQuest Timing Analyzer settings, in the **Settings** dialog box, allow you to configure the options shown in Table 7–55.

Table 7–55. The Quartus II TimeQuest Timing Analyzer Settings (Part 1 of 2)

Options	Description
.sdc files to include in the project	Adds and removes **.sdc** files associated with the project.
Enable Advanced I/O Timing	Generates advanced I/O timing results from board trace models specified for each pin.
Enable multicorner timing analysis during compilation	Generates multiple reports for all available operating conditions of the target device.

Table 7–55. The Quartus II TimeQuest Timing Analyzer Settings (Part 2 of 2)

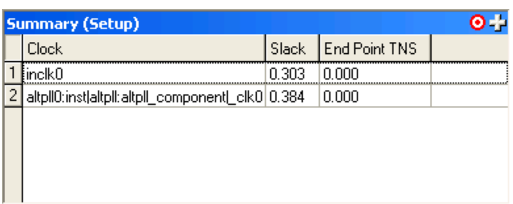
Options	Description
Report worst-case paths during compilation	Generates worst-case path reports per clock domain.
Tcl Script File for customizing report during compilation	Specifies any custom scripts to be sourced for any custom report generation.

The options shown in Table 7–55 control only the reports generated in the Compilation Report, and do not control the reports generated in the Quartus II TimeQuest Timing Analyzer.

View Pane

The **View** pane is the main viewing area for the timing analysis results. Use the **View** pane to view summary reports, custom reports, or histograms. Figure 7–36 shows the **View** pane after you select the **Summary (Setup)** report from the Report pane.

Figure 7–36. Summary (Setup) Report

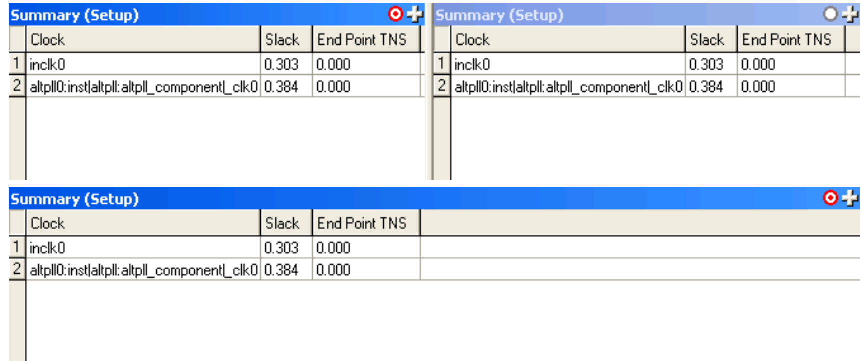


View Pane: Splitting

For analyzing the timing results properly, comparing multiple reports is extremely important. To facilitate multiple report viewing, the **View** pane supports window splitting. Window splitting divides the **View** pane into multiple windows, allowing you to view different reports side-by-side.

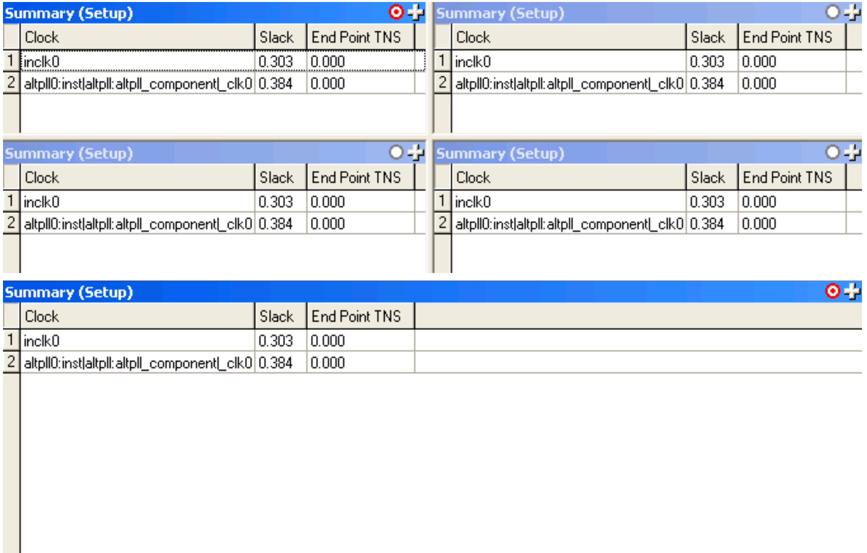
You can split the **View** pane into multiple windows using the split icon located in the upper right corner of the **View** pane. Drag the icon in different directions to generate additional window views in the **View** pane. For example, if you drag the split icon to the left, the **View** pane creates a new window to the right of the current window (Figure 7–37).

Figure 7–37. Splitting the View Pane to the Left (Before and After Split Left)



If you drag the split icon diagonally, the **View** pane creates three new windows in the **View** pane (Figure 7–38).

Figure 7–38. Splitting the View Pane Diagonally (Before and After Diagonal Split)

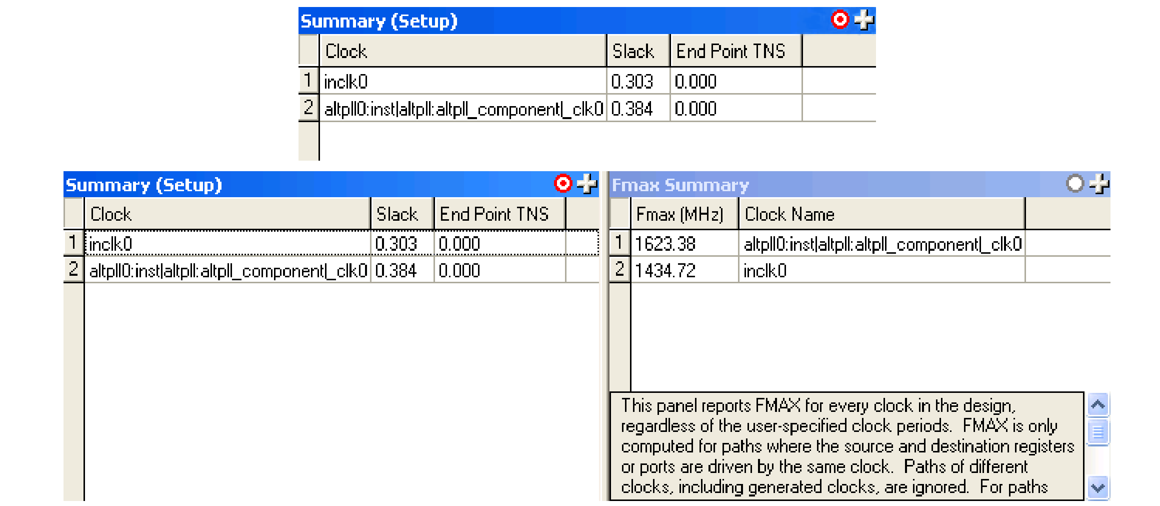


Drag the split icon downward to create a new window directly below the current window.

View Pane: Removing Split Windows

You can remove windows that you create in the View pane using the split icon by dragging the border of the window over the window you wish to remove (Figure 7–39).

Figure 7–39. Removing a Split Window (Before and After Split is Removed)



Tasks Pane

Use the **Tasks** pane to access common commands such as netlist setup and report generation.

The following common commands are located in the **Tasks** pane: Open Project, Set Operating Conditions, and Reset Design. The other commands, including timing netlist setup and report generation, are contained in the following folders:

- Netlist Setup
- Reports

Each command in the **Tasks** pane has an equivalent Tcl command that is displayed in the **Console** pane when the command runs.

Opening a Project and Writing a Synopsys Design Constraints File

To open a project in the Quartus II TimeQuest Timing Analyzer, double-click the **Open Project** task. If you launch the Quartus II TimeQuest Timing Analyzer from the Quartus II software GUI, the project opens automatically.

You can add or remove constraints from the timing netlist after the Quartus II TimeQuest Timing Analyzer reads the initial **.sdc** file. After the file is read, the initial **.sdc** file becomes outdated compared to the constraints in the Quartus II TimeQuest Timing Analyzer. Use the **Write SDC File** command to generate an **.sdc** file that is up-to-date and reflects the current state of constraints in the Quartus II TimeQuest Timing Analyzer.

Netlist Setup Folder

The Netlist Setup folder contains tasks that are used to set up the timing netlist for timing analysis. The three tasks located in this folder are Create Timing Netlist, Read SDC File, and Update Timing Netlist.

Use the Create Timing Netlist task to create a netlist that the Quartus II TimeQuest Timing Analyzer uses to perform static timing analysis. This netlist is used only for timing analysis by the Quartus II TimeQuest Timing Analyzer.

You must always create a timing netlist before you perform an analysis with the Quartus II TimeQuest Timing Analyzer.

Use the Read SDC File command to apply constraints to the timing netlist. By default, the Read SDC File command reads the <*current revision*>**.sdc** file.

Use the `read_sdc` command to read an **.sdc** file that is not associated with the current revision of the design.

Use the Update Timing Netlist command to update the timing netlist after you enter constraints or read an **.sdc** file. You should use this command if any constraints are added or removed from the design.

Reports Folder

The Reports folder contains commands to generate timing summary reports of the static timing analysis results. The twelve commands located in this folder are summarized in Table 7–56.

Table 7–56. Reports Folder Commands

Report Task	Description
Report Fmax Summary	Generates a f_{MAX} summary report for all clocks in the design.
Report Setup Summary	Generates a clock setup summary report for all clocks in the design.
Report Hold Summary	Generates a clock hold summary report for all clocks in the design.
Report Recovery Summary	Generates a recovery summary report for all clocks in the design.
Report Removal Summary	Generates a removal summary report for all clocks in the design.
Report Clocks	Generates a summary report of all created clocks in the design.
Report Clock Transfers	Generates a summary report of all clock transfers detected in the design.
Report Minimum Pulse Width	Generates a summary report of all minimum pulse widths in the design.
Report SDC	Generates a summary report of the constraints read from the **.sdc** file.
Report Unconstrained Paths	Generates a summary report of all unconstrained paths in the design.
Report Ignored Constraints	Generates a summary report of all ignored SDC constraints for the design.
Report Datasheet	Generates a datasheet report for the design.

Macros Folder

The Macros folder contains commands that perform custom tasks available in the Quartus II TimeQuest Timing Analyzer utility package. These commands are: Report All Summaries, Report Top Failing Paths, and Create All Clock Histograms.

Table 7–57 describes the commands available in the Macros folder.

Table 7–57. Macros Folder Commands

Macro Task	Description
Report All Summaries	This command runs the Report Setup Summary, Report Hold Summary, Report Recovery Summary, Report Removal Summary, and Minimum Pulse Width commands to generate all summary reports.
Report Top Failing Paths	This command generates a report containing a list of top failing paths.
Create All Clock Histograms	This command runs the Create Slack Histogram command to generate a clock histogram for all clocks in the design.
Report All I/O Timings	This command generates a report of all timing paths that start or end at a device port.
Report All Core Timings	This command generates a report of all timing paths that start and end at the device register.

Console Pane

The **Console** pane is both a message center for the Quartus II TimeQuest Timing Analyzer and an interactive Tcl console. The **Console** pane has two tabs: the **Console** tab and the **History** tab. The **Console** tab shows all messages, such as info and warning messages. Also, the Console tab allows you to enter and run Synopsys design constraints and Tcl commands. The Console tab shows the Tcl equivalent of all commands that you run in the **Tasks** pane. The History tab records all the Synopsys design constraints and Tcl commands that are run.

To run the commands located in the History tab after the timing netlist has been updated, right-click the command and click **Rerun**.

You can copy Tcl commands from the Console and History tabs to easily generate Tcl scripts to perform timing analysis.

Report Pane

Use the **Report** pane to access all reports generated from the **Tasks** pane, and by any custom report commands. When you select a report in the **Report** pane, the report is shown in the active window in the **View** pane.

If a report is out-of-date with respect to the current constraints, a "?" icon is shown next to the report.

Constraints

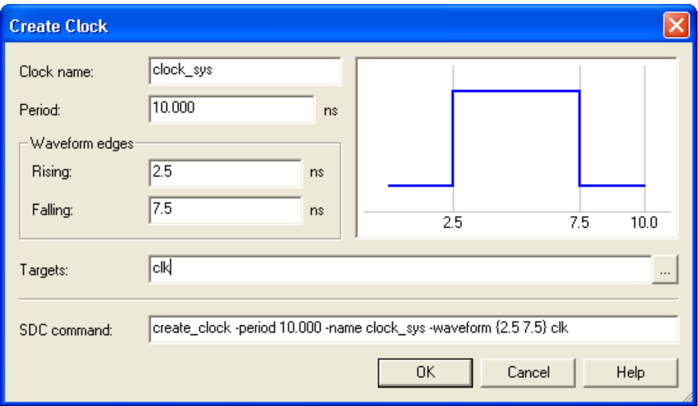
Use the Constraints menu to access commonly used constraints, exceptions, and commands. You can access this menu from the toolbar, click **Edit** and then click **Constraint menu**.

The following commands are available on the Constraints menu:

- **Create Clock**
- **Create Generated Clock**
- **Set Clock Latency**
- **Set Clock Uncertainty**
- **Set Clock Groups**
- **Remove Clock**

For example, use the **Create Clock** dialog box to create clocks in your design. Figure 7–40 shows the **Create Clock** dialog box.

Figure 7–40. Create Clock Dialog Box



The following commands specify timing exceptions and are available on the Constraints menu:

- **Set False Path**
- **Set Multicycle Path**
- **Set Maximum Delay**
- **Set Minimum Delay**

All the dialog boxes used to specify timing constraints or exceptions from commands have an SDC command field. This field contains the SDC command that is run when you click OK.

All commands and constraints created in the Quartus II TimeQuest Timing Analyzer user interface are echoed in the **Console** pane.

The constraints specified with Constraints menu commands are not saved to the current **.sdc** file automatically. You must run the **Write SDC File** command to save your constraints.

The following commands are available on the Constraints menu in the Quartus II TimeQuest Timing Analyzer:

- **Generate SDC File from QSF**
- **Read SDC File**
- **Write SDC File**

The **Generate SDC File from QSF** command runs a Tcl script that converts the Quartus II Classic Timing Analyzer constraints in a QSF file to an **.sdc** file for the Quartus II TimeQuest Timing Analyzer. The file *<current revision>***.sdc** is created by this command.

For information about the **Generate SDC File from QSF** command, refer to the *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

The Generate SDC File from QSF command attempts to convert all timing constraints and exceptions in the QSF file to their equivalent **.sdc** file constraints. However, not all QSF file constraints are convertible to **.sdc** file constraints. Review the **.sdc** file after it is created to ensure that all constraints have been successfully converted.

The **Read SDC File** command reads the <*current revision*>**.sdc** file.

When you select the **Write SDC File** command, an up-to-date **.sdc** file that reflects the current state of constraints in the Quartus II TimeQuest Timing Analyzer is generated.

Name Finder

Use the **Name Finder** dialog box to select the target for any constraints or exceptions in the Quartus II TimeQuest Timing Analyzer GUI. The **Name Finder** dialog box allows you to specify collections, filters, and filter options. The Collections field in the **Name Finder** dialog box allows you to specify the type of name to select. To select the type, in the **Collection** list, select the desired collection API from the following list:

- `get_cells`
- `get_clocks`
- `get_keepers`
- `get_nets`
- `get_nodes`
- `get_pins`
- `get_ports`
- `get_registers`

For more information about the various collection APIs, refer to “Collections” on page 7–21.

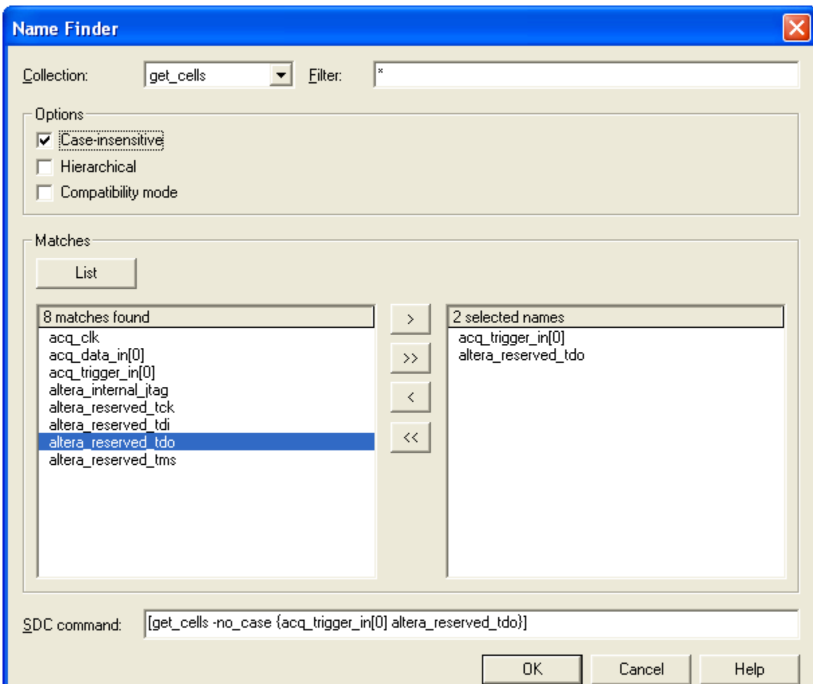
The Filter field allows you to filter names based on your own criteria, including wildcard characters. You can further refine your results using the following filter options:

- Case-insensitive
- Hierarchical
- Compatibility mode

For more information about the filter options, refer to “Wildcard Assignments and Collections” on page 7–76.

The Name Finder dialog box also provides an SDC command field that displays the currently selected name search command. You can copy the value from this field and use it for other constraint target fields. The **Name Finder** dialog box is shown in Figure 7–41.

Figure 7–41. Name Finder Dialog Box



Target Pane

When using the TimeQuest GUI, you can split the **View** pane into multiple windows. The splitting feature allows you to display multiple reports in the **View** pane. After splitting the **View** pane, the last active window is updated with any new reports. You can change this behavior by changing the state of each split window. To change the window state, click the target circle in the upper right corner (Figure 7–42). Table 7–58 describes the state of each window.

Figure 7–42. Target Pane

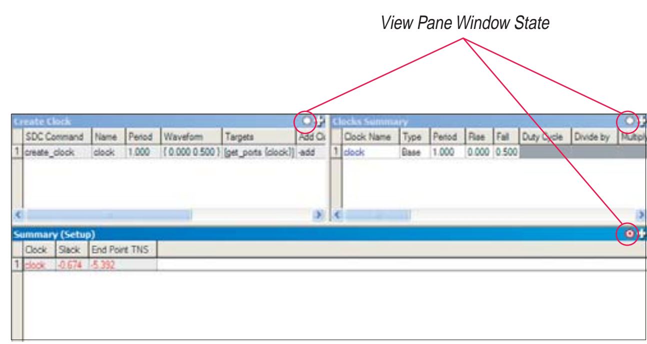


Table 7–58. View Pane Window State

State	Description
Partially Filled Red Circle	Indicates that the active window displays any new reports.
Fully Filled Red Circle	Indicates that the window, independent of it being the active window, displays any new reports.
Empty Circle	Indicates that the window does not display any new reports.

Clicking on the circle in the upper right corner of an active window changes the state of the window.

SDC Editor

The TimeQuest Timing Analyzer GUI also provides an SDC editor. The SDC editor provides an easy and convenient way to write, edit, and view **.sdc** files. The SDC editor is context sensitive. After an SDC constraint or exception has been entered, a tooltip appears that shows the options and format for the constraint or exception.

The SDC editor also provides helpful tools, including SDC templates and SDC templates for common design structures. To find these templates, when the SDC editor is active, look on the **Edit** menu.

On the menu bar, the Constraints menu opens the **Constraints** dialog box. After you have finished entering all required parameters, the **.sdc** file is inserted at the current cursor position.

Conclusion

The Quartus II TimeQuest Timing Analyzer addresses the requirements of complex designs, resulting in increased productivity and efficiency through its intuitive user interface, support of industry-standard constraints format, and scripting capabilities. The Quartus II TimeQuest Timing Analyzer is a next-generation timing analysis tool that supports the industry-standard SDC format and allows designers to create, manage, and analyze complex timing constraints and to perform advanced timing verification.

Referenced Documents

This chapter references the following documents:

- *AN 481: Applying Multicycle Exceptions in the TimeQuest Timing Analyzer*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *Managing Metastability with the Quartus II Software* chapter of volume 1 in the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer Cookbook*
- *SDC and TimeQuest API Reference Manual*
- *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *TimeQuest Quick Start Tutorial*

- *Understanding Metastability in FPGAs* White Paper
- *Volume 4: SOPC Builder* in the *Quartus II Handbook*

Document Revision History

Table 7–59 shows the revision history for this chapter.

Table 7–59. Document Revision History (Part 1 of 2)

Date and Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Updated Table 7–1 ■ Updated "Metastability" on page 7–15 ■ Updated "Delay and Skew Specifications" on page 7–42 ■ Added "set_max_skew" on page 7–43 ■ Added "report_exceptions" on page 7–55 ■ Updated "report_metastability" on page 7–57 ■ Added "report_partitions" on page 7–66 ■ Added "report_max_skew" on page 7–69 ■ Updated "Multi-Corner Analysis" on page 7–74 ■ Added Table 7–52 ■ Removed report_path section ■ Minor editorial updates	Updated for the Quartus II software version 9.0 release.

Table 7–59. Document Revision History (Part 2 of 2)

Date and Version	Changes Made	Summary of Changes

November 2008 v8.1.0	Updated for the Quartus II software version 8.1, including:
	■ Added the following sections:
	  ■ "set_net_delay" on page 7–42
	  ■ "Annotated Delay" on page 7–49
	  ■ "report_net_delay" on page 7–66
	■ Updated the descriptions of the `-append` and `-file` <*name*> options in tables throughout the chapter
	■ Updated entire chapter using 8½" × 11" chapter template
	■ Minor editorial updates
		Medium update for the Quartus II software version 8.1 release.

May 2008 v8.0.0	Updated for the Quartus II software version 8.0, including: ■ Changed the heading "Specify Design Timing Requirements" to "The Quartus II TimeQuest Timing Analyzer Flow Guidelines" on page 7–26 ■ In "SDC Constraint Files" on page 7–29, added information about order-sensitivity ■ Added a new section on "Metastability" on page 7–19 ■ Added a new section on "Common Clock Path Pessimism" on page 7–22 ■ Removed information about Asynchronous clocks from "Clock Groups" on page 7–43 ■ Updated information in Example 7–28 ■ Added three entries to Table 7–22 ■ Added information to Table 7–24 ■ Added information about the RSKM to "report_rskm" on page 7–80, including a formulaic equation (Equation 12) ■ Added the section "Clock Groups" on page 7–43 ■ Added Table 7–44 to "report_clock_fmax_summary" on page 7–86 ■ Added qualifier to introduction of Table 7–46 ■ Added Speed Grade information to Table 7–46 ■ Removed [-dtw] and added [-add] to information about the derive_clock_uncertainty command ("Derive Clock Uncertainty" on page 7–47) ■ Added the section "report_metastability" on page 7–68 ■ Added a new information about RSKM to "report_rskm" on page 7–80 ■ Added the section "Cross-Probing" on page 7–93 ■ Minor editorial updates ■ Added hyperlinks to referenced documents throughout chapter	Significant update for the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

8. Best Practices for the Quartus II TimeQuest Timing Analyzer

QII53024-9.0.0

Introduction

Timing constraints and exceptions are vital to all designs that target FPGAs. Timing constraints and exceptions allow designers to specify requirements and verify timing of their systems or FPGAs. This chapter provides the steps necessary to fully constrain an FPGA design with the Quartus® II TimeQuest Timing Analyzer.

This chapter assumes that you are familiar with the Quartus II TimeQuest Timing Analyzer.

This chapter is divided into three major sections. The ordering of the sections also shows the recommended flow to apply timing constraints and exceptions in the Quartus II TimeQuest Timing Analyzer.

This chapter contains the following sections:

1. "Clock Requirements"
2. "I/O Requirements" on page 8–4
3. "Exceptions" on page 8–5

For more information about constraints and exceptions supported by the Quartus II TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter of volume 3 in the *Quartus II Handbook* or the *SDC and TimeQuest API Reference Manual*.

Clock Requirements

Clocks play a major role in timing constraints. The Quartus II TimeQuest Timing Analyzer supports three different types of clocks:

- Base clocks
- Derived clocks
- Virtual clocks

Clocks are used to specify register-to-register requirements for synchronous transfers and guide the Fitter optimization algorithms to achieve the best possible placement and routes for the design.

Clocks should be the first constraints specified in any design's SDC files. This is important because the Quartus II TimeQuest Timing Analyzer reads SDC constraints and exceptions from the top of the file to the bottom of the file. Also, many constraints reference clocks and, therefore, the clocks must be defined first.

Base Clocks

Base clocks are the primary input clocks generated into the FPGA. These clocks are usually generated from off-chip oscillators or forwarded from an external device. Base clocks are not clocks derived within the FPGA, such as PLLs. Base clocks should be defined first, because derived clocks and other constraints can reference them.

Use the SDC command `create_clock` to constrain all primary input clocks. The target for `create_clock` should use the collection `get_ports`. For example, specifying a 100-MHz requirement on an input clock port `clk_sys` is shown in Example 8–1.

Example 8–1. create_clock Command

```
create_clock -period 10 [get_ports clk_sys]
```

With the `-add` option, you can apply multiple clocks on the same clock node. For example, if two oscillators can drive the same clock port on the FPGA, the following SDC commands can be used:

```
create_clock -period 10 [get_ports clk_sys] ↵
create_clock -period 5 [get_ports clk_sys] -add ↵
```

Derived Clocks

Derived clocks are clocks generated internally in the FPGA. These clocks modify or synthesize a source clock signal. For example, derived clocks can be PLLs or register clock dividers. Derived clocks should be constrained after all base clocks have been constrained. This ensures that all properties of the base clock are properly accounted for in the derived clock.

Use the SDC command `create_generated_clock` to constrain all generated clocks in the design. The source of the `create_generated_clock` should be a node in the design and not a previously constrained clock.

TimeQuest supports the command `derive_pll_clocks` to automatically constrain all PLL outputs used in the FPGA. This removes the requirement to manually constrain each PLL output clock.

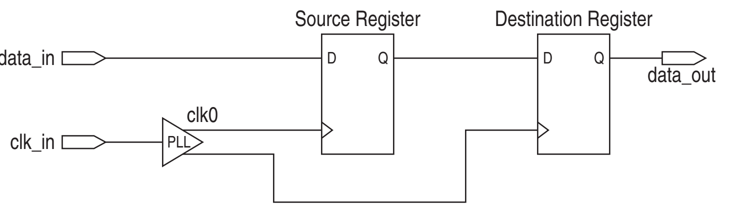
Virtual Clocks

A virtual clock is a clock that does not have a real source in the design or that does not interact directly with the design. Virtual clocks are created with the `create_clock` command, but no targets are specified. For example, to create a 10 ns virtual clock us the following command:

```
create_clock -period 10 -name my_virt_clk ↵
```

Figure 8–1 and Example 8–2 show when a virtual clock should be used. If an FPGA interfaces with an external device and both the FPGA and external device have different clock sources, the clock source for the external device must be modeled with a virtual clock.

Figure 8–1. Inter-Clock Transfer



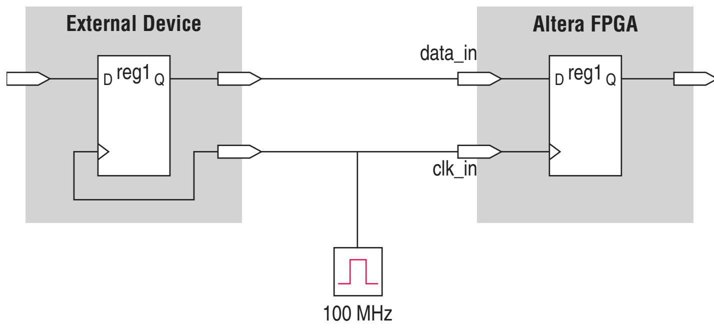
Example 8–2. Virtual Clock Example 1

```
#create base clock for the design
create_clock -period 5 [get_ports system_clk]
#create the virtual clock for the external register
create_clock -period 10 -name virt_clk -waveform { 0 5 }
#set the output delay referencing the virtual clock
set_output_delay -clock virt_clk -max 1.5 [get_ports dataout]
```

Altera recommends that you use virtual clocks when modeling external delays using the `set_input_delay` and `set_output_delay` constraints. This is important when using the command `derive_clock_uncertainty` for the design. The virtual clock prevents the `derive_clock_uncertainty` command from applying clock uncertainties for either intra- or inter-clock transfers on an I/O interface clock transfer.

For each clock in the design that feeds an input or output port, an equivalent virtual clock should be created. Figure 8–2 and Example 8–3 show an example of this.

Figure 8–2. I/O Interface Specifications



Example 8–3. SDC Commands to Constrain the I/O Interface

```
# Create the base clock for the clock port
create_clock -period 10 -name clk_in [get_ports clk_in]
# Create a virtual clock with the same properties of the base clock
# driving the source register
create_clock -period 10 -name virt_clk_in
# Create the input delay referencing the virtual clock and not the base
# clock
# DO NOT use set_input_delay -clock clk_in <delay_value>
# [get_ports data_in]
set_input_delay -clock virt_clk_in <delay value> [get_ports data_in]
```

I/O Requirements

To fully analyze a design, all timing requirements must be specified. This includes specifying internal timing requirements as well as external timing requirements. With external timing requirements specified, the I/O interface or periphery of the FPGA can be verified against any system specification. The Quartus II TimeQuest Timing Analyzer supports two types of external delay modeling: input and output.

I/O requirements should be specified after all clocks in the design have been constrained. Also, when specifying I/O requirements, a virtual clock should be referenced in the constraints.

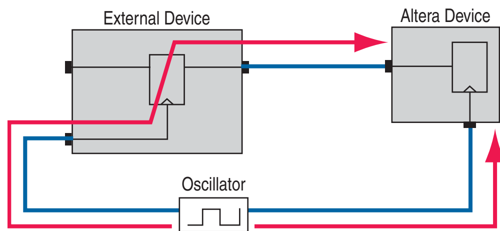
Either the input or output requirements can be specified after the clock constraints.

Input Requirements

The input requirements allow all external delays feeding into the FPGA to be specified. The requirements need to be specified for all input ports in the design.

Use the SDC command `set_input_delay` to specify external input delay requirements. The `set_input_delay` should reference a virtual clock for the `-clock` option. The virtual clock defines the launching clock for the input port. The latching clock inside the chip that captures the input data is automatically determined because all clocks in the chip have been defined. Figure 8–3 shows an example of an input delay referencing a virtual clock.

Figure 8–3. Set Input Delay

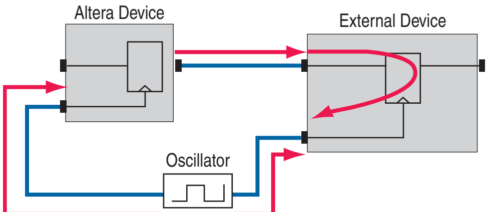


Output Requirements

The output requirements allow all external delays from the FPGA to be specified. The requirements must be specified for all output ports in the design.

Use the SDC command `set_output_delay` to specify external output delay requirements. The `set_output_delay` should reference a virtual clock for the `-clock` option. The virtual clock defines the latching clock for the output port. The launching clock inside the chip that launches the output data is automatically determined because all clocks in the chip have been defined. Figure 8–4 shows an example of an output delay referencing a virtual clock.

Figure 8–4. Output Delay



Exceptions

Timing exceptions in the Quartus II TimeQuest Timing Analyzer provide a way to modify the default timing analysis behavior to match the analysis required by your design. The Quartus II TimeQuest Timing Analyzer supports these three major categories of exceptions:

- False paths
- Minimum and maximum delays
- Multicycles

Because timing exceptions modify the default analysis, they should be specified after the clocks and input and output delay constraints.

False Paths

By specifying a false path in your design, you are removing the specified path from analysis. This path can be point-to-point or clock-to-clock. An example is a static configuration register that is written once during power up initialization, but will never change state again. These signals often cross clock domains, but because some data may transfer across those clock domains, you may not want to cut the clock transfers, but instead, selectively cut the path from the static configuration register to all endpoints.

Example 8–4 shows how to cut the path from all registers beginning with **A** to all registers beginning with **B**.

Example 8–4. False Path

```
set_false_path -from [get_pins A*] -to [get_pins B*]
```

The Quartus II TimeQuest Timing Analyzer assumes all clocks are related until you specify otherwise. Setting clock groups is an efficient way of cutting false clock-to-clock timing relationships in the design. It requires fewer line entries to cut the paths between clocks, compared to writing multiple `set_false_path` exceptions between every clock transfer to be cut. Use the `set_clock_groups` command to collect groups of signals related to each other, and use `-asynchronous` to specify that each group of clocks is asynchronous with each other. In the case of multiple clocks applied to the same port for multi-mode operation, use `set_clock_groups` with `-exclusive` to declare these clocks are placed into separate groups and mutually exclusive to each other. In other words, the clocks cannot physically exist in the design at the same time.

Minimum and Maximum Delays

Asynchronous signals that do not have a specific clock relationship in the design, but need a bounded maximum and minimum path delay, can be accomplished with the use of the `set_max_delay` and `set_min_delay` constraints. Typically, this timing exception is used for paths that go from port to port through the FPGA without a register stage in the path. When using this timing exception to constrain the path delay, specify both the maximum and minimum delay of the path. Do not constrain only the maximum the minimum value. The `set_max_delay` and `set_min_delay` commands modify the setup and hold relationship equal to the constraint's value.

Alternatively, you can use the `set_net_delay` constraint to specify the maximum, minimum, or skew for any edge in your design. This constraint is used in situations where no clock relationships are defined or required.

For more information about the `set_net_delay` constraint, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Multicycles

Multicycle paths are often difficult to identify in a design. This requires intimate knowledge of the design functionality to know if a signal is updated or sampled on the default clock edge relationships derived by the TimeQuest Timing Analyzer. Thorough coverage of multicycle paths in the design can greatly increase performance of the Fitter and provide better quality of results in compiles because it relaxes some invalid setup and hold edge relationships.

An example of a potential multicycle path would be long combinational paths where the latching register does not require data stability on every clock edge, but only on every second clock edge. This is dependent on the endpoint register's usage of that signal. In this case, a `set_multicycle_path -setup` 2 states that data is stable at the endpoint every two clock cycles of the endpoint latch clock.

When specifying a multicycle path, both the setup and hold multicycle relationships should be defined. For the preceding example, though it can take two clock cycles to set data at the endpoint, the minimum hold time relationship must be defined with a multicycle as well. Typically, the `set_multicycle_path -hold` value is (N – 1), where N is equal to the `set_multicycle_path -setup` value, for a register-to-register path within the same clock domain. However, if data is crossing across different clock domains, the phase and period of the launch and latch clock may change the proper `-setup` and `-hold` values to something different than `-setup` N and `-hold` (N – 1). Use these with caution and examine the timing paths carefully in the TimeQuest Timing Analyzer before and after applying the multicycle to determine if the launch and latch clock edges are in the proper relationship.

For more information about multicycles in the TimeQuest Timing Analyzer, refer to *AN 481: Applying Multicycle Exceptions in the TimeQuest Timing Analyzer*.

Conclusion

Specifying all timing constraints and exceptions for your design is one of the most important aspects of design implementation in the Quartus II software. Constraints and exceptions allow the Quartus II Fitter to focus on the critical paths in the design and reduce the amount of time spent on non-critical parts of the design. Also, constraints and exceptions provide an easy method to verify the design's timing requirements. Following the guidelines and flow in this chapter provides you with an easy path to a successful design implementation in the Quartus II software.

Referenced Documents

This chapter references the following documents:

- *AN 481: Applying Multicycle Exceptions in the TimeQuest Timing Analyzer*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *SDC and TimeQuest API Reference Manual*

Document Revision History

Table 8–1 shows the revision history for this chapter.

Table 8–1. Document Revision History

Date and Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Initial release.	—

9. Switching to the Quartus II TimeQuest Timing Analyzer

QII53019-9.0.0

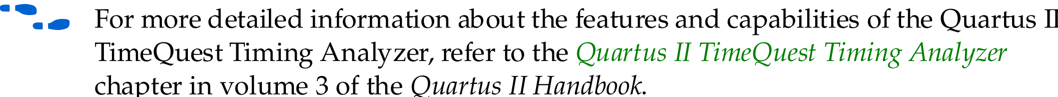
Introduction

The Quartus II TimeQuest Timing Analyzer provides more powerful timing analysis features than the Quartus II Classic Timing Analyzer. This chapter describes the benefits of switching to the Quartus II TimeQuest Timing Analyzer, the differences between the Quartus II TimeQuest and Quartus II Classic Timing Analyzers, and the process you should follow to switch a design from using the Quartus II Classic Timing Analyzer to the Quartus II TimeQuest Timing Analyzer.

Benefits of Switching to the Quartus II TimeQuest Timing Analyzer

Increasing design complexity requires a timing analysis tool with greater capabilities and flexibility. The Quartus II TimeQuest Timing Analyzer offers the following benefits:

- Industry-standard Synopsys Design Constraint (SDC) support increases productivity.
- Simple, flexible reporting uses industry-standard terminology and makes timing sign-off faster.



For more detailed information about the features and capabilities of the Quartus II TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

These features ease constraint and analysis of modern, complex designs. SDC constraints support complex clocking schemes, high-speed interfaces, and more logic. An example includes designs that have multiplexed clocks, regardless of whether they are switched on or off chip. Designs with source-synchronous interfaces, such as DDR memory interfaces, are much simpler to constrain and analyze with the Quartus II TimeQuest Timing Analyzer.

There are three main differences between the Quartus II Classic and Quartus II TimeQuest Timing Analyzers. Unlike the Quartus II Classic Timing Analyzer, the Quartus II TimeQuest Timing Analyzer has the following three benefits:

- All clocks are related by default. (Refer to "Related and Unrelated Clocks" on page 9–11.)
- The default hold multicycle value is zero. (Refer to "Hold Multicycle" on page 9–20.)
- You must constrain all ports and ripple clocks. (Refer to "Automatic Clock Detection" on page 9–16.)

Chapter Contents

"Switching to the Quartus II TimeQuest Timing Analyzer" describes the four-step process you should follow to switch a design to the Quartus II TimeQuest Timing Analyzer.

"Differences Between Quartus II TimeQuest and Quartus II Classic Timing Analyzers" on page 9–4 covers terminology, constraints, clocks, hold multicycle, and other differences.

"Timing Assignment Conversion" on page 9–26 is a comprehensive guide to converting Quartus II Classic QSF timing assignments to Quartus II TimeQuest SDC constraints.

"Conversion Utility" on page 9–44 describes a utility that helps you convert Classic QSF timing assignments to the Quartus II TimeQuest SDC constraints.

"Notes" on page 9–54 includes notes about support for specific features in the current version of the Quartus II TimeQuest Timing Analyzer.

Switching to the Quartus II TimeQuest Timing Analyzer

You should use the following process to switch a design from the Quartus II Classic Timing Analyzer to the Quartus II TimeQuest Timing Analyzer. The process is composed of the following steps, which are described in detail in the next sections:

1. Compile your design and perform timing analysis with the Quartus II Classic Timing Analyzer (page 9–2).
2. Create a Synopsys Design Constraints (**.sdc**) file that contains timing constraints (page 9–2).
3. Perform timing analysis with the Quartus II TimeQuest Timing Analyzer and examine the reports (page 9–3).
4. Set the default timing analyzer to TimeQuest (page 9–4).

Compile Your Design

To begin, compile your design with the Quartus® II software. You should run the Quartus II Classic Timing Analyzer during compilation because it is easier to convert your assignments to SDC constraints when you create an **.sdc** file. To run the Quartus II Classic Timing Analyzer in the Quartus II GUI, on the Processing menu, click **Start**, then click **Start Timing Analyzer**. To run the Quartus II Classic Timing Analyzer if you are a command-line user, type `quartus_tan` <*project*> ↵ at a system command prompt.

Create an .sdc File

The Quartus II TimeQuest Timing Analyzer supports SDC format constraints. If you are familiar with SDC terminology, you can create an **.sdc** file with any text editor and skip to "Perform Timing Analysis with the Quartus II TimeQuest Timing Analyzer" on page 9–3. Name the **.sdc** file <*revision*>**.sdc** (<*revision*> is the current revision of your project) and save it in your project directory.

Refer to the *SDC and TimeQuest Tcl API Reference Manual* for a TimeQuest SDC command reference.

Alternately, you can use a Quartus II TimeQuest conversion utility to help you convert the timing assignments in an existing Quartus II Settings File (**.qsf**) to corresponding SDC constraints.

Conversion Utility

To run the Quartus II TimeQuest conversion utility, click **Generate SDC file from QSF** on the Constraints menu. You can also run the conversion utility by typing either of the following commands at a system command prompt:

- `quartus_tan --qsf2sdc` <*project name*> ↵

or

- `quartus_sta --qsf2sdc` <*project name*> ↵

The **.sdc** file created by the conversion utility is named <*revision*>**.sdc**.

For information about how to run the Quartus II TimeQuest Timing Analyzer, refer to "Run the Quartus II TimeQuest Timing Analyzer".

If you use the conversion utility, you must review the **.sdc** file to ensure it is correct and complete, and make changes if necessary. Refer to "Constraint File Priority" on page 9–8 for the recommended way to make changes.

The conversion utility cannot convert some types of Quartus II Classic Timing Analyzer assignments for the following reasons:

- No corresponding SDC constraint exists
- Multiple SDC constraints are valid, so correct conversion requires knowledge of the intended function of your design

You must manually convert any such assignments based on the guidelines in "Timing Assignment Conversion" on page 9–26.

Perform Timing Analysis with the Quartus II TimeQuest Timing Analyzer

When your **.sdc** file is complete, use the reporting capabilities in the Quartus II TimeQuest Timing Analyzer. If you use the Quartus II TimeQuest GUI, double-click any of the reports listed in the Tasks pane. You can also type commands in the Quartus II TimeQuest Tcl shell to generate reports.

You should also review "Notes" on page 9–54 to ensure the Quartus II TimeQuest Timing Analyzer supports all stages of your design flow.

For complete information about how to use the Quartus II TimeQuest Timing Analyzer, and descriptions of commands and reports, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*, and the *SDC and TimeQuest Tcl API Reference Manual*.

Run the Quartus II TimeQuest Timing Analyzer

If you are using the Quartus II software, to open the Quartus II TimeQuest GUI, on the Tools menu, click **TimeQuest Timing Analyzer**. The Quartus II TimeQuest GUI automatically opens the project you have open in the Quartus II GUI.

If you use the system command prompt to open the Quartus II TimeQuest Timing Analyzer, type `quartus_staw` ↵ to open the Quartus II TimeQuest GUI, or type `quartus_sta -s` ↵ to start the Quartus II TimeQuest Timing Analyzer in Tcl shell mode. Use the **project_open** command to open your project, or, on the File menu, click **Open Project**.

Set the Default Timing Analyzer

To use the Quartus II TimeQuest Timing Analyzer as the default timing analyzer for your project, turn on **Use TimeQuest Timing Analyzer during compilation**. In the Quartus II GUI, on the Assignments menu, click **Settings**, then select the **Timing Analysis Settings** category, and turn on **Use TimeQuest Timing Analyzer during compilation**. You can make the same setting in your project's **.qsf** file with the following Tcl command:

```
set_global_assignment -name USE_TIMEQUEST_TIMING_ANALYZER ON
```

This setting directs the Quartus II software to use the Quartus II TimeQuest Timing Analyzer instead of the Quartus II Classic Timing Analyzer.

The setting to make the Quartus II TimeQuest Timing Analyzer the default timing analyzer is specific to each project, so you can decide on a per-project basis whether to use the Quartus II TimeQuest Timing Analyzer or the Quartus II Classic Timing Analyzer.

If you want to use the Quartus II Classic Timing Analyzer instead of the Quartus II TimeQuest Timing Analyzer, ensure **Use Classic Timing Analyzer during compilation** is selected. You can delete the *<revision>***.sdc** file, because the Quartus II Classic Timing Analyzer does not use it.

In the Quartus II software, a timing analyzer performs two functions:

- Processing timing constraints and exceptions that affect how your design is placed and routed
- Reporting after place and route is complete so you know whether the design meets timing requirements

Although you can use one timing analyzer to process timing constraints during place and route and the other for reporting, you should use the same timing analyzer for both. The Quartus II Classic Timing Analyzer uses assignments in the **.qsf** file, and the Quartus II TimeQuest Timing Analyzer uses constraints in the **.sdc** file. Any differences between the timing assignments in the two files may cause inconsistent results.

Differences Between Quartus II TimeQuest and Quartus II Classic Timing Analyzers

The Quartus II TimeQuest Timing Analyzer is different from the Quartus II Classic Timing Analyzer in the following ways:

- "Terminology" on page 9–5
- "Constraints" on page 9–6
- "Clocks" on page 9–11
- "Hold Multicycle" on page 9–20
- "Fitter Behavior" on page 9–22
- "Reporting" on page 9–22
- "Scripting API" on page 9–25

Terminology

This section introduces the industry-standard SDC terminology that the Quartus II TimeQuest Timing Analyzer uses.

For more detailed information about this terminology, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Netlist

The Quartus II TimeQuest Timing Analyzer uses SDC naming conventions for netlists. Netlists consist of cells, pins, nets, ports, and clocks.

- Cells are instances of fundamental hardware elements in Altera® FPGAs (such as logic elements, look-up tables, and registers).
- Pins are inputs and outputs of cells.
- Nets are connections between output pins and input pins.
- Ports are top-level module inputs and outputs (device inputs and outputs).
- Clocks are abstract objects outside the netlist.

The terminology of pins and ports is opposite to that of the Quartus II Classic Timing Analyzer. In the Quartus II Classic Timing Analyzer, ports are inputs and outputs of cells, and pins are top-level module inputs and outputs (device inputs and outputs).

Figure 9–1 shows a simple design, and Figure 9–2 shows the Quartus II TimeQuest netlist representation of the design. Netlist elements in Figure 9–2 are labeled to illustrate the SDC terminology.

Figure 9–1. Sample Design

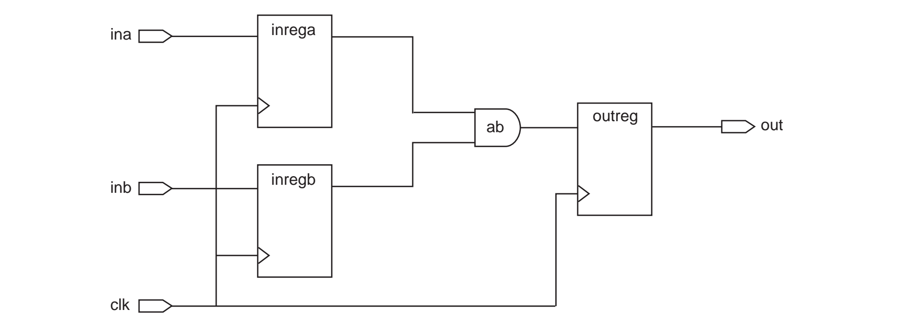
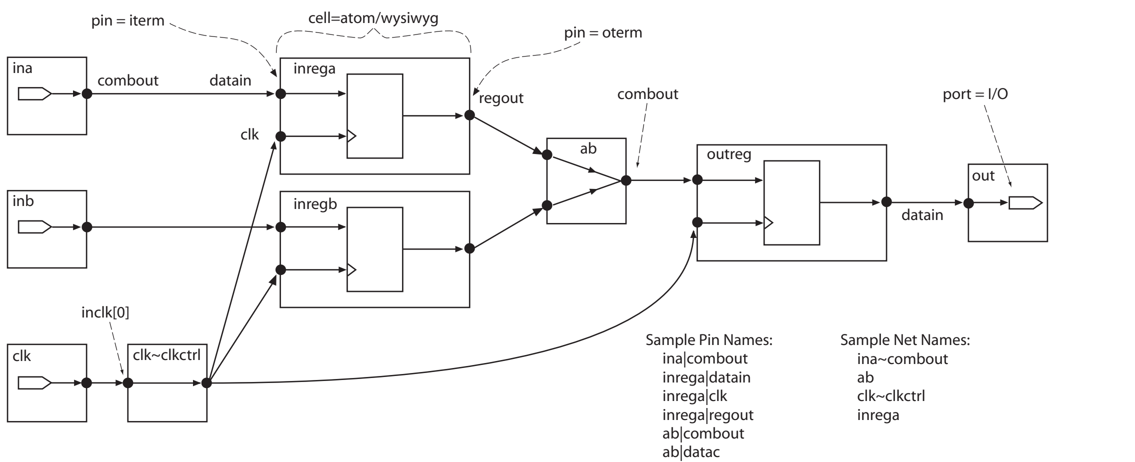


Figure 9–2. Quartus II TimeQuest Timing Analyzer Netlist



Collections

In addition to standard SDC collections, the Quartus II TimeQuest Timing Analyzer supports the following Altera-specific collection types:

- **Keepers**—Non-combinational nodes in a netlist
- **Nodes**—Nodes can be combinational, registers, latches, or ports (device inputs and outputs)
- **Registers**—Registers or latches in the netlist

You can use the **get_keepers**, **get_nodes**, or **get_registers** commands to access these collections.

Constraints

The Quartus II Classic and Quartus II TimeQuest Timing Analyzers store constraints in different files, support different methods for constraint entry, and prioritize constraints differently. The following sections detail these differences.

Constraint Files

The Quartus II TimeQuest Timing Analyzer stores all SDC timing constraints in **.sdc** files. The Quartus II Classic Timing Analyzer stores all timing assignments in your project's **.qsf** file. The **.qsf** file contains all your project's assignments and settings except for the Quartus II TimeQuest Timing Analyzer constraints. The Quartus II TimeQuest Timing Analyzer ignores the timing assignments in your **.qsf** file except when the conversion utility converts Quartus II Quartus II Classic QSF timing assignments to Quartus II TimeQuest SDC constraints. There is no automatic process that keeps timing constraints synchronized between your **.qsf** and **.sdc** files. If you want to keep the constraints synchronized, you must convert them manually.

Constraint Entry

In the Quartus II Classic Timing Analyzer, you enter timing assignments with the **Settings** dialog box, the Assignment Editor, or with commands in Tcl scripts. The Quartus II TimeQuest Timing Analyzer does not use the Assignment Editor for its constraints, and you cannot use the Assignment Editor to enter SDC constraints. You must use one of the following methods to enter Quartus II TimeQuest constraints:

- Enter constraints at the Tcl prompt in the Quartus II TimeQuest Timing Analyzer
- Enter constraints in an **.sdc** file with a text editor or SDC editor
- Use the constraint entry commands on the Constraints menu in the Quartus II TimeQuest Timing Analyzer GUI

You can enter timing assignments for the Quartus II Classic Timing Analyzer even if no timing netlist exists for your design. The Quartus II TimeQuest Timing Analyzer requires that a netlist exist for interactive constraint entry. Each Quartus II TimeQuest Timing Analyzer constraint is a Tcl command evaluated in real-time, if entered directly in the Tcl console. As part of this evaluation, the Quartus II TimeQuest Timing Analyzer validates all names. To do this, SDC commands can only be evaluated after a netlist is created. An **.sdc** file can be created at any time using the Quartus II TimeQuest Timing Analyzer or any other text editor, but a netlist is required before an **.sdc** file can be sourced. You must create a timing netlist in the Quartus II TimeQuest Timing Analyzer before you can enter constraints with either of the following interactive methods:

- At the Tcl console of the Quartus II TimeQuest Timing Analyzer
- With commands on the Constraints menu in the Quartus II TimeQuest Timing Analyzer GUI

If you enter constraints with a text editor separate from the Quartus II TimeQuest Timing Analyzer, no timing netlist is required.

To create a timing netlist in the Quartus II TimeQuest Timing Analyzer, use the **create_timing_netlist** command, or double-click **Create Timing Netlist** in the Tasks pane of the Quartus II TimeQuest GUI.

If you have never compiled your design, and you want to use the Quartus II TimeQuest Timing Analyzer to enter constraints interactively, you must synthesize your design before you create a timing netlist. To synthesize your design, type the following command at a system command prompt:

```
quartus_map <project name> ↵
```

If you use the Quartus II GUI, ensure that your project is open, then click **Start** on the Processing menu, and click **Start Analysis and Synthesis**.

To create the netlist, open the Quartus II TimeQuest Timing Analyzer. Then, on the Netlist menu, click **Create Timing Netlist...**, select **Post-map**, and click **OK**. Alternately, at the TCL console, type the following command:

```
create_timing_netlist -post_map ↵
```

Time Units

Enter time values are in default time units of nanoseconds (ns) with up to three decimal places. Note that the Quartus II TimeQuest Timing Analyzer does not display the default time unit when it displays time values.

You can specify a different default time unit with the **set_time_format -unit** <*default time unit*> command, or specify another unit when you enter a time value, for example, `300ps`.

Specifying time units with the value is not part of the standard SDC format. This is a Quartus II TimeQuest Timing Analyzer extension.

You can specify clock constraints with period or frequency in the Quartus II TimeQuest Timing Analyzer. For example, you can use either of the following constraints:

- `create_clock -period 10.000`
  (assuming default units and decimal places)
- `create_clock -period "100 MHz"`
- `create_clock -period "10 ns"`

MegaCore Functions

If you change any MegaCore function settings and regenerate the core after you convert your timing assignments to SDC constraints, you must manually update the SDC constraints or reconvert your assignments. You must update or reconvert, because changes to MegaCore function settings can affect timing assignments embedded in the hardware description language files of the core. The timing assignments are not converted automatically when the core settings change.

You should make a backup copy of your **.sdc** file before reconverting assignments. If you made changes to the **.sdc** file, you can manually copy the updated MegaCore timing constraints to your **.sdc** file.

Bus Name Format

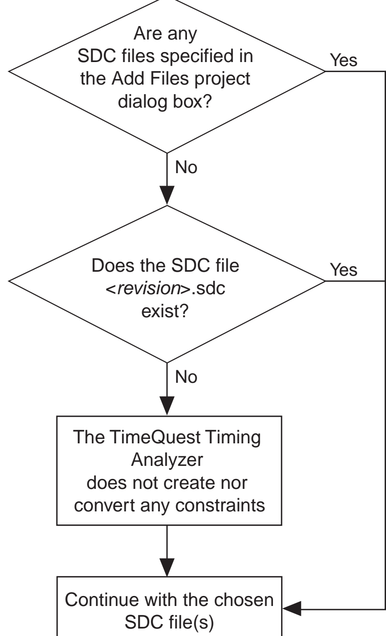
In the Quartus II Classic Timing Analyzer, you can make a timing assignment to all bits in a bus with the bus name (or the bus name followed by an asterisk enclosed in square brackets) as the target. For example, to make an assignment to all bits of a bus called address, use `address` or `address[*]` as the target of the assignment.

In the Quartus II TimeQuest Timing Analyzer, you must use the bus name followed by square brackets enclosing an asterisk, like this: `address[*]`.

Constraint File Priority

The Quartus II TimeQuest Timing Analyzer searches for **.sdc** files with a specific priority, as shown in Figure 9–3.

Figure 9–3. .sdc File Search Order



If you specify constraints in multiple **.sdc** files, or if you use a single **.sdc** file with a name other than <*revision*>**.sdc**, you must add the files to your project so the Quartus II TimeQuest Timing Analyzer can find them. If you use the Quartus II software, click **Add/Remove Files in Project** on the Project menu, and add the appropriate **.sdc** files. You can also add **.sdc** files to your project with the following Tcl command in your **.qsf** file, repeated once for each **.sdc** file:

```
set_global_assignment -name SDC_FILE <SDC file name>
```

The Quartus II TimeQuest Timing Analyzer reads constraint files from the files list in the order they are listed, first to last.

☞ If you use an **.sdc** file created by the conversion utility, you should place it before all other **.sdc** files in the list of files. When conflicting constraints apply to the same node, the last constraint has the highest priority. Therefore, **.sdc** files with your additions or changes should be listed after the **.sdc** file created by the conversion utility, so your constraints have higher priority.

Beginning with version 6.1, the Quartus II TimeQuest Timing Analyzer does not run the conversion utility automatically when it cannot find an **.sdc** file according to the priority shown in Figure 9–3. It may prompt you to run the conversion utility from the Constraints menu in the Quartus II TimeQuest GUI.

☞ You must review the **.sdc** file as you would when manually running the conversion utility. Refer to "Reviewing Conversion Results" on page 9–51 for information about how to review the converted constraints.

If no **.sdc** file exists when you run the Quartus II Fitter, and you have turned on **Use TimeQuest Timing Analyzer during compilation**, the Fitter does not create an **.sdc** file automatically, but it attempts to meet a default 1 GHz constraint on all clocks in your design.

Constraint Priority

The Quartus II Classic Timing Analyzer prioritizes assignments based on the specificity of the nodes to which they are assigned. The more specific an assignment is, the higher its priority. The Quartus II TimeQuest Timing Analyzer simplifies these precedence rules. When overlaps occur in the nodes to which the constraints apply, constraints at the bottom of the file take priority over constraints at the top of the file.

As an example, in the Quartus II Classic Timing Analyzer, point-to-point multicycle assignments have higher priority than single point multicycle assignments. The two assignments in Example 9–1 result in a multicycle assignment of 2 between `A_reg` and all nodes beginning with B, including `B_reg`. The single point assignment does not apply to paths from `A_reg` to `B_reg`, because the specific point-to-point assignment takes priority over the general single point assignment.

Example 9–1. Quartus II Classic Timing Analyzer Multicycle Assignments

```
set_instance_assignment -name MULTICYCLE -from A_reg -to B* 2
set_instance_assignment -name MULTICYCLE -to B_reg 3
```

Example 9–2 shows SDC versions of the preceding Quartus II Classic Timing Analyzer timing assignments. However, the Quartus II TimeQuest Timing Analyzer evaluates the constraints from top to bottom (regardless of point-to-point or single point), so the path from `A_reg` to `B_reg` receives a multicycle exception of 3 because it is second in order.

Example 9–2. Quartus II TimeQuest Timing Analyzer Multicycle Exceptions

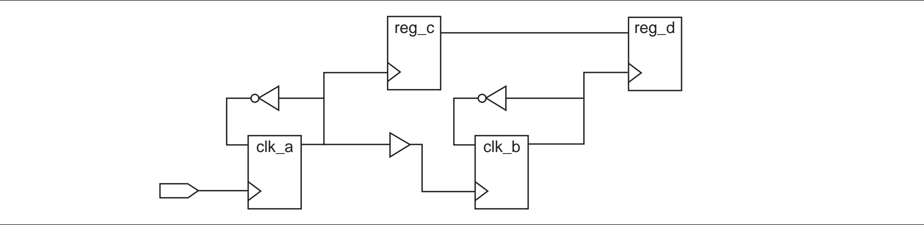
```
set_multicycle_path -from [get_keepers A_reg] -to [get_keepers B*] 2
set_multicycle_path -to [get_keepers B_reg] 3
```

Ambiguous Constraints

Because of new capabilities in the Quartus II TimeQuest Timing Analyzer, some Quartus II Classic assignments can be ambiguous after conversion by the conversion utility. These assignments require manual updating based on your knowledge of your design.

Figure 9–4 shows a ripple clock circuit. The explanation that follows shows an ambiguous constraint for that circuit, and how to edit the constraint to remove the ambiguity in the Quartus II TimeQuest Timing Analyzer.

Figure 9–4. Ripple Clock Circuit



In the Quartus II Classic Timing Analyzer, the following QSF multicycle assignment from `clk_a` to `clk_b` with a value of 2 applies to paths transferring data from the `clk_a` domain to the `clk_b` domain:

```
set_instance_assignment -name MULTICYCLE -from clk_a -to clk_b 2
```

In Figure 9–4, this assignment applies to the path from `reg_c` to `reg_d`. In the Quartus II TimeQuest Timing Analyzer, the use of the clock node names in the following equivalent multicycle exception is ambiguous:

```
set_multicycle_path -setup -from clk_a -to clk_b 2
```

The exception could apply to the path between `clk_a` and `clk_b`, or it could apply to paths from one ripple clock domain to the other ripple clock domain (`reg_c` to `reg_d`).

The Quartus II TimeQuest exceptions shown in Example 9–3 are not ambiguous because they use collections to explicitly specify the targets of the exception.

Example 9–3. Unambiguous Quartus II TimeQuest Timing Analyzer Exceptions

```
# Applies to path from reg_c to reg_d
set_multicycle_path -setup -from [get_clocks clk_a] \
   -to [get_clocks clk_b] 2
# Applies to path from clk_a to clk_b
set_multicycle_path -setup -from [get_registers clk_a] \
   -to [get_registers clk_b] 2
```

Clocks

The Quartus II Classic and Quartus II TimeQuest Timing Analyzers detect, analyze, and report clocks differently. The following sections describe these differences.

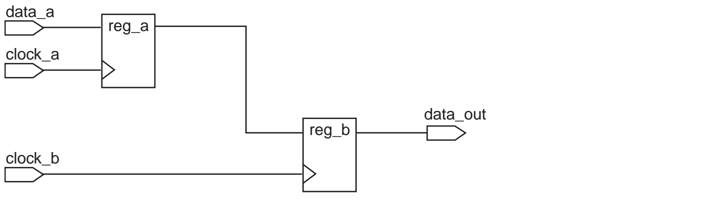
Related and Unrelated Clocks

In the Quartus II TimeQuest Timing Analyzer, all clocks are related by default, and you must add assignments to indicate unrelated clocks. However, in the Quartus II Classic Timing Analyzer, all base clocks are unrelated by default. All derived clocks of a base clock are related to each other, but are unrelated to other base clocks and their derived clocks.

You can change the default behavior of the Quartus II Classic Timing Analyzer to treat all clocks as related clocks. On the Assignments menu, click **Timing Analysis Settings**. Click **More Settings** and then select **Cut paths between unrelated clock domains**. Ensure that the setting is off.

Figure 9–5 on page 9–12 shows a simple circuit with a path between two clock domains. The Quartus II TimeQuest Timing Analyzer analyzes the path from `reg_a` to `reg_b` because all clocks are related by default. The Quartus II Classic Timing Analyzer does not analyze the path from `reg_a` to `reg_b` by default.

Figure 9–5. Cross Clock Domain Path



To make clocks unrelated in the Quartus II TimeQuest Timing Analyzer, use the **set_clock_groups** command with the `-exclusive` option. For example, the following command makes `clock_a` and `clock_b` unrelated, so the Quartus II TimeQuest Timing Analyzer does not analyze paths between the two clock domains.

```
set_clock_groups -exclusive -group {clock_a} -group {clock_b}
```

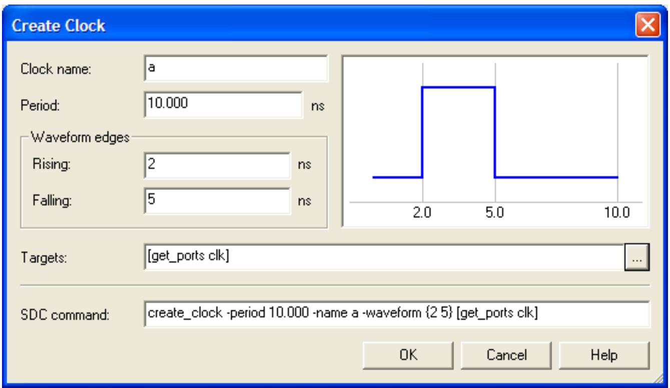
Clock Offset

In the Quartus II TimeQuest Timing Analyzer, clocks can have non-zero values for the rising edge of the waveform, a feature that the Quartus II Classic Timing Analyzer does not support. To specify an offset for your clock, use the waveform option for the **create_clock** command to specify the rising and falling edge times, as shown in this example:

```
-waveform {<rising edge time> <falling edge time>}
```

Figure 9–6 shows a clock constraint with an offset in the Quartus II TimeQuest Timing Analyzer GUI.

Figure 9–6. Create Clock Screen



Clock offset affects setup and hold relationships. Launch and latch edges are evaluated after offsets are applied. Depending on the offset, the setup relationship can be the offset value, or the difference between the period and offset. You should not use clock offset to emulate latency. You should use the clock latency constraint instead. Refer to “Offset and Latency Example” on page 9–13 for an example that illustrates the different effects of offset and latency.

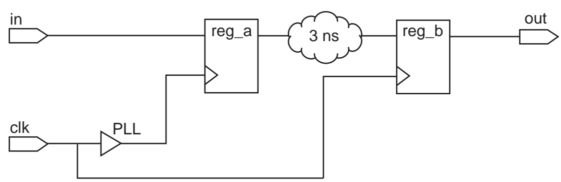
Clock Latency

The Quartus II TimeQuest Timing Analyzer does not ignore early clock latency and late clock latency differences when the clock source is the same, as the Quartus II Classic Timing Analyzer does. When you specify latencies, you should take common clock path pessimism into account and use uncertainty to control pessimism differences for clock-to-clock data transfers. Unlike clock offset, clock latency affects skew, and launch and latch edges are evaluated before latencies are applied, so the setup relationship is always equal to the period.

Offset and Latency Example

Figure 9–7 shows a simple register-to-register circuit used to illustrate the different effects of offset and latency. The examples show why you should not use clock offset to emulate clock latency. You should always turn on the **Enable Clock Latency** option in the Quartus II Classic Timing Analyzer. This option is in the **More Settings** box of the **Timing Settings** dialog box.

Figure 9–7. Simple Circuit for Offset and Latency Examples

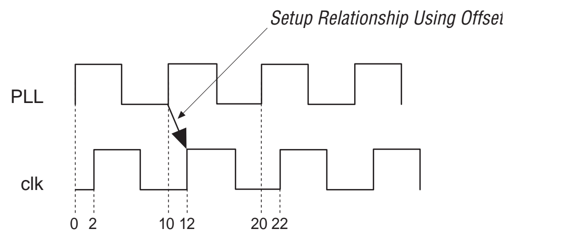


The period for `clk` is 10 ns, and the period for the PLL output is 10 ns. The PLL compensation value is –2 ns. The network delay from the PLL to `reg_a` equals the network delay from `clk` to `reg_b`. Finally, the delay from `reg_a` to `reg_b` is 3 ns.

Clock Offset Scenario

Treat the PLL compensation value of –2 ns as a clock offset of –2 ns with a clock skew of 0 ns. Launch and latch edges are evaluated after offsets are applied, so the setup relationship is 2 ns (Figure 9–8).

Figure 9–8. Setup Relationship Using Offset



Equation 9–1 shows how to calculate the slack value for the path from `reg_a` to `reg_b`.

Equation 9–1.

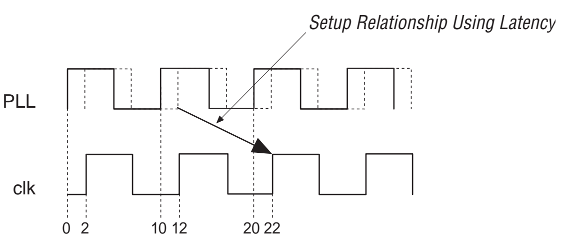
$$\text{slack} = \text{clock arrival} - \text{data arrival}$$
$$\text{slack} = \text{setup relationship} + \text{clock skew} - \text{reg_to_reg delay}$$
$$\text{slack} = 2\text{ns} + 0\text{ns} - 3\text{ns}$$
$$\text{slack} = -1\text{ns}$$

The negative slack requires a multicycle assignment with a value of 2 and a hold multicycle assignment with a value of 1 to correct. With these assignments from `reg_a` to `reg_b`, the setup relationship is then 12 ns, resulting in a slack of 9 ns.

Clock Latency Scenario

Treat the PLL compensation value of –2 ns as latency with a clock skew of 2 ns. Because launch and latch edges are evaluated before latencies are applied, the setup relationship is 10 ns (the period of `clk` and the PLL) (Figure 9–9).

Figure 9–9. Setup Relationship Using Latency



Equation 9–2 shows how to calculate the slack value for the path from `reg_a` to `reg_b`.

Equation 9–2.

$$\text{slack} = \text{clock arrival} - \text{data arrival}$$
$$\text{slack} = \text{setup relationship} + \text{clock skew} - \text{reg_to_reg delay}$$
$$\text{slack} = 10\text{ns} + 2\text{ns} - 3\text{ns}$$
$$\text{slack} = 9\text{ns}$$

The slack of 9 ns is identical to the slack computed in the previous example, but because this example uses latency instead of offset, no multicycle assignment is required.

Clock Uncertainty

The Quartus II Classic Timing Analyzer ignores **Clock Setup Uncertainty** and **Clock Hold Uncertainty** assignments when you specify a setup or hold relationship between two clocks. However, the Quartus II TimeQuest Timing Analyzer does not ignore clock uncertainty when you specify a setup or hold relationship between two clocks. Figure 9–10 and Figure 9–11 illustrate the different behavior between the Quartus II TimeQuest and Quartus II Classic Timing Analyzers.

In both figures, the constraints are identical. There is a 20-ns period for `clk_a` and `clk_b`. There is a setup relationship (a **set_max_delay** exception in the Quartus II TimeQuest Timing Analyzer) of 7 ns from `clk_a` to `clk_b`, and a clock setup uncertainty constraint of 1 ns from `clk_a` to `clk_b`. The actual setup relationship in the Quartus II TimeQuest Timing Analyzer is 1 ns less than in the Quartus II Classic Timing Analyzer because of the way they analyze clock uncertainty.

Figure 9–10. Quartus II Classic Timing Analyzer Behavior

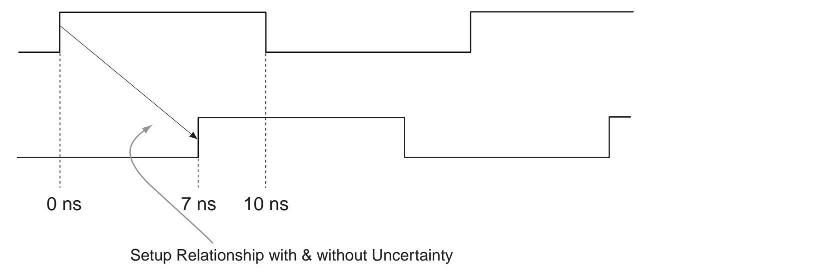
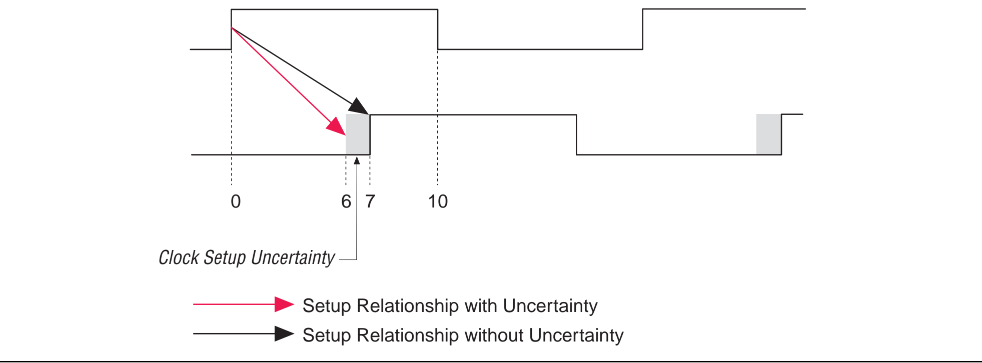


Figure 9–11. Quartus II TimeQuest Timing Analyzer Behavior



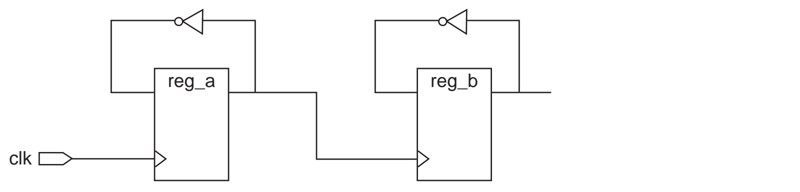
Derived and Generated Clocks

Generated clocks in the Quartus II TimeQuest Timing Analyzer are different than derived clocks in the Quartus II Classic Timing Analyzer. In the Quartus II Classic Timing Analyzer, the source of a derived clock must be a base clock. However, in the Quartus II TimeQuest Timing Analyzer, the source of a generated clock can be any other clock in the design (including virtual clocks), or any node to which a clock propagates through the clock network. Because generated clocks are related through the clock network, you can specify generated clocks for isolated modules, such as IP, without knowing the details of the clocks outside of the module.

In the Quartus II TimeQuest Timing Analyzer, you can specify generated clocks relative to specific edges and edge shifts of a master clock, a feature that the Quartus II Classic Timing Analyzer does not support.

Figure 9–12 shows a simple ripple clock that you should constrain with generated clocks in the Quartus II TimeQuest Timing Analyzer.

Figure 9–12. Generated Clocks Circuit



The Quartus II TimeQuest Timing Analyzer constraints shown in Example 9–4 constrain the clocks in the circuit above. Note that the source of each generated clock can be the input pin of the register itself, not the name of another clock.

Example 9–4. Generated Clock Constraints

```
create_clock -period 10 -name clk clk
create_generated_clock -divide_by 2 -source reg_a|CLK -name reg_a reg_a
create_generated_clock -divide_by 2 -source reg_b|CLK -name reg_b reg_b
```

Automatic Clock Detection

The Quartus II Classic and Quartus II TimeQuest Timing Analyzers identify clock sources of registers that do not have a defined clock source. The Quartus II Classic Timing Analyzer traces back along the clock network, through registers and logic, until it reaches a top-level port in your design. The Quartus II TimeQuest Timing Analyzer also traces back along the clock network, but it stops at registers.

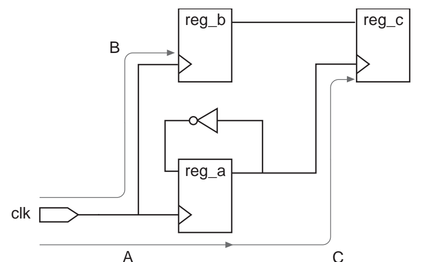
You can use two SDC commands in the Quartus II TimeQuest Timing Analyzer to automatically detect and create clocks for unconstrained clock sources:

- **derive_clocks**—creates clocks on sources of clock pins that do not already have at least one clock sourcing the clock pin
- **derive_pll_clocks**—identifies PLLs and creates generated clocks on the clock output pins

derive_clocks Command

Figure 9–13 shows a simple circuit with a divide-by-2 register and indicates the clock network delays as A, B, and C. The following explanation describes how the Quartus II Classic and Quartus II TimeQuest Timing Analyzers detect the clocks in Figure 9–13.

Figure 9–13. Circuit for derive_clocks Example



The Quartus II Classic Timing Analyzer detects that `clk` is the clock source for registers `reg_a`, `reg_b`, and `reg_c`. It detects that `clk` is the clock source for `reg_c` because it traces back along the clock network for `reg_c` through `reg_a`, until it finds the `clk` port. The Quartus II Classic Timing Analyzer computes the clock arrival time for `reg_c` as $A + C$.

The **derive_clocks** command in the Quartus II TimeQuest Timing Analyzer creates two base clocks, one on the `clk` port and one on `reg_a`, because the command does not trace through registers on the clock network. The clock arrival time for `reg_c` is C because the clock starts at `reg_a`.

To make the Quartus II TimeQuest Timing Analyzer compute the same clock arrival time ($A + C$) as the Quartus II Classic Timing Analyzer for `reg_c`, make the following modifications to the clock constraints created by the **derive_clocks** command:

- Change the base clock named `reg_a` to a generated clock
- Make the source the clock pin of `reg_a` (`reg_a|clk`) or the port `clk`
- Add a `-divide_by 2` option

These modifications cause the clock arrival times to `reg_c` to match between the Quartus II Classic Timing Analyzer and the Quartus II TimeQuest Timing Analyzer. However, the clock for `reg_c` is shown as `reg_a` instead of `clk`, and the launch and latch edges may change for some paths due to the divide-by-2.

You can use the **derive_clocks** command at the beginning of your design cycle when you do not know all of the clock constraints for your design, but you should not use it during timing sign-off. Instead, you should constrain each clock in your design with the **create_clock** or **create_generated_clocks** commands.

The **derive_clocks** command detects clocks in your design using the following rules:

1. An input clock port is detected as a clock only if there are no other clocks feeding the destination registers.
   a. Input clock ports are not detected automatically if they feed only other base clocks.
   b. If other clocks feed the port's register destinations, the port is assumed to be an enable or data port for a gated clock.
   c. When no clocks are defined, and multiple clocks feed a destination register, the auto-detected clock is selected arbitrarily.
2. All ripple clocks (registers in a clock path) are detected as clocks automatically using the same rules for input clock ports. If both an input port and a register feed register clock pins, the input port is selected as the clock.

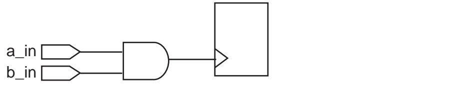
The following examples show how the **derive_clocks** command detects clocks in the simple circuit shown in Figure 9–14.

Figure 9–14. Simple Circuit 1

- If you do not make any clock settings, and then you run **derive_clocks**, it detects `a_in` as a clock according to rule 1, because there are no other clocks feeding the register.
- If you create a clock with `b` as its target, and then you run **derive_clocks**, it does not detect `a_in` as a clock according to rule 1a, because `a_in` feeds only another clock.

The following examples show how the **derive_clocks** command detects clocks in the simple circuit shown in Figure 9–15.

Figure 9–15. Simple Circuit 2



- If you do not make any clock settings and then you run **derive_clocks**, it selects a clock arbitrarily, according to rule 1c.
- If you create a clock with `a_in` as its target and then you run **derive_clocks**, it does not detect `b_in` as a clock according to rule 1b, because another clock (`a_in`) feeds the register.

derive_pll_clocks Command

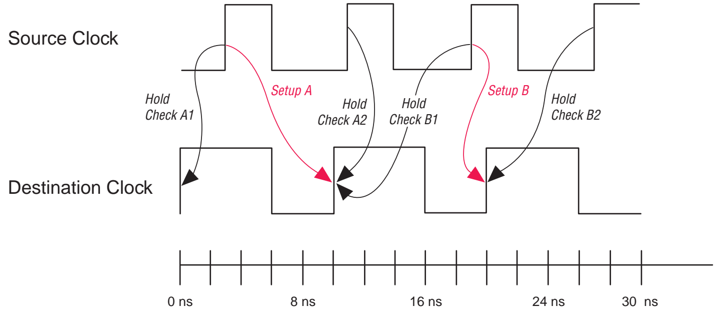
The **derive_pll_clocks** command names the generated clocks according to the names of the PLL output pins by default, and you cannot change these generated clock names. If you want to use your own clock names, you must use the **create_generated_clock** command for each PLL output clock and specify the names with the `-name` option.

If you use the PLL clock-switchover feature, the **derive_pll_clocks** command creates additional generated clocks on each output clock pin of the PLL based on the secondary clock input to the PLL. This may require **set_clock_groups** or **set_false_path** commands to cut the primary and secondary clock outputs. For information about how to make clocks unrelated, refer to “Related and Unrelated Clocks” on page 9–11.

Hold Relationship

The Quartus II TimeQuest and Quartus II Classic Timing Analyzers choose the worst-case hold relationship differently. Refer to Figure 9–16 for sample waveforms to illustrate the different effects.

Figure 9–16. Worst-Case Hold



The Quartus II Classic Timing Analyzer first identifies the worst-case setup relationship. The worst-case setup relationship is **Setup B**. Then the Quartus II Classic Timing Analyzer chooses the worst-case hold relationship (Hold Check B1 or Hold Check B2) for that specific setup relationship, Setup B. The Quartus II Classic Timing Analyzer chooses Hold Check B2 for the worst-case hold relationship.

However, the Quartus II TimeQuest Timing Analyzer calculates worst-case hold relationships for all possible setup relationships and chooses the absolute worst-case hold relationship. The Quartus II TimeQuest Timing Analyzer checks two hold relationships for every setup relationship:

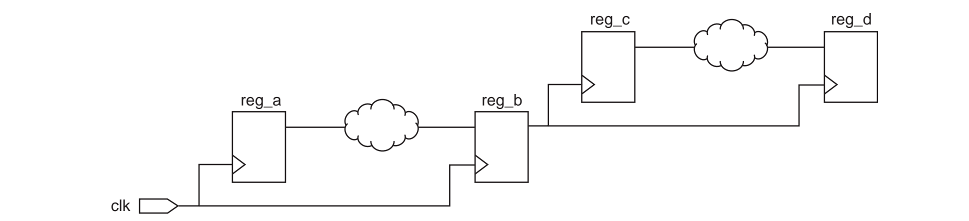
- Data launched by the current launch edge not captured by the previous latch edge (Hold Check A1 and Hold Check B1)
- Data launched by the next launch edge not captured by the current latch edge (Hold Check A2 and Hold Check B2)

The Quartus II TimeQuest Timing Analyzer chooses Hold Check A2 as the absolute worst-case hold relationship.

Clock Objects

The Quartus II Classic Timing Analyzer treats nodes with clock settings assigned to them as special objects in the timing netlist. Any node in the timing netlist with a clock setting is treated as a clock object, regardless of its actual type, such as a register. When a register has a clock setting assigned to it, the Quartus II Classic Timing Analyzer does not analyze register-to-register paths beginning or ending at that register. Figure 9–17 shows a circuit that illustrates this situation.

Figure 9–17. Clock Objects



With no clock assignments on any of the registers, the Quartus II Classic Timing Analyzer analyzes timing on the path from `reg_a` to `reg_b`, and from `reg_c` to `reg_d`. If you make a clock setting assignment to `reg_b`, `reg_b` is no longer considered a register node in the netlist, and the path from `reg_a` to `reg_b` is no longer analyzed.

In the Quartus II TimeQuest Timing Analyzer, clocks are abstract objects that are associated with nodes in the timing netlist. The Quartus II TimeQuest Timing Analyzer analyzes the path from `reg_a` to `reg_b` even if there is a clock assigned to `reg_b`.

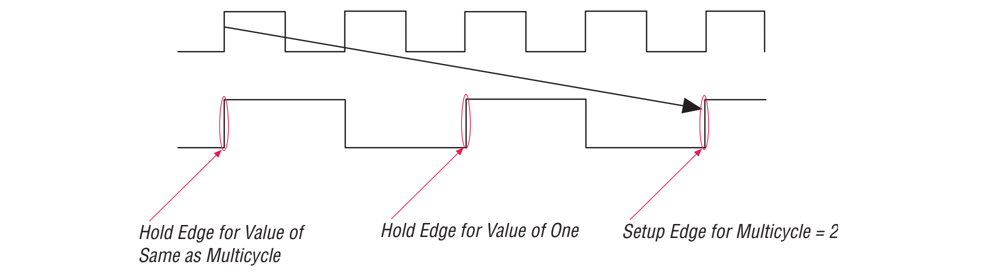
Hold Multicycle

The hold multicycle value numbering scheme is different in the Quartus II Classic and Quartus II TimeQuest Timing Analyzers. Also, you can choose between two values for the default hold multicycle value in the Quartus II Classic Timing Analyzer but you cannot change the default value in the Quartus II TimeQuest Timing Analyzer. The hold multicycle value specifies which clock edge is used for hold analysis when you change the latch edge with a multicycle assignment.

In the Quartus II Classic Timing Analyzer, the hold multicycle value is based on 1, and is the number of clock cycles away from the setup edge. In the Quartus II TimeQuest Timing Analyzer, the hold multicycle value is based on zero, and is the number of clock cycles away from the default hold edge. In the Quartus II TimeQuest Timing Analyzer, the default hold edge is one edge before or after the setup edges. Subtract 1 from any hold multicycle value in the Quartus II Classic Timing Analyzer to compute the equivalent value for the Quartus II TimeQuest Timing Analyzer.

In the Quartus II Classic Timing Analyzer, you can set the default value of the hold multicycle assignment to **One** or **Same as Multicycle**. The default value applies to any multicycle assignment in your design that does not also have a multicycle hold assignment. Figure 9–18 illustrates the difference between **One** and **Same as Multicycle** for a multicycle assignment of 2 using the Quartus II Classic Timing Analyzer.

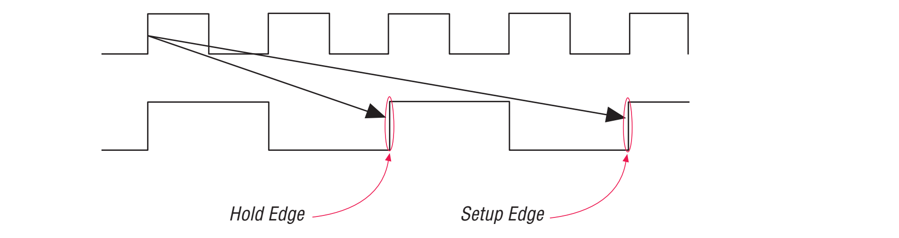
Figure 9–18. Difference Between One and Same As Multicycle



If the default value is **One**, the Quartus II Classic Timing Analyzer uses the clock edge one before the setup edge for hold analysis. If the default value is **Same as Multicycle**, the Quartus II Classic Timing Analyzer uses the clock edge that is <*value of multicycle assignment*> edges back from the setup edge.

Figure 9–19 shows simple waveforms for a cross-clock domain transfer with the indicated setup and hold edges.

Figure 9–19. First Relationship Example



In the Quartus II TimeQuest Timing Analyzer, only a multicycle exception of 2 is required to constrain the design for the indicated setup and hold relationships.

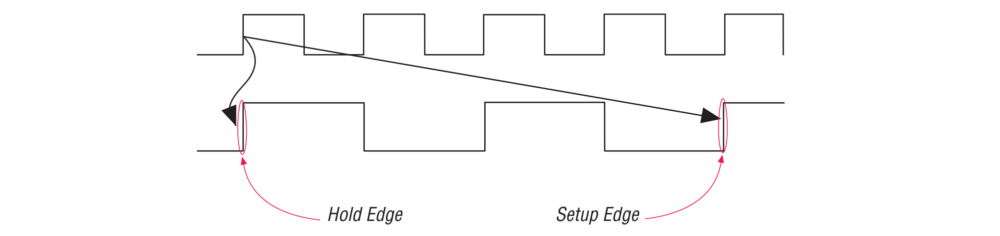
In the Quartus II Classic Timing Analyzer, if the **Default Hold Multicycle** value is **One**, only a multicycle assignment of 2 is required to constrain the design.

In the Quartus II Classic Timing Analyzer, if the **Default Hold Multicycle** value is **Same as Multicycle**, you must make two assignments to constrain the design:

- A multicycle assignment of 2
- A hold multicycle assignment of 1 to override the default value

Figure 9–20 shows simple waveforms for a different cross-clock domain transfer with indicated setup and hold edges. The following explanation shows what exceptions to apply to achieve the desired setup and hold relationships.

Figure 9–20. Second Relationship Example



In the Quartus II TimeQuest Timing Analyzer, you must use the following two exceptions:

- A multicycle exception of 2
- A hold multicycle exception of 1, because the hold edge is one edge behind the default hold edge, which is one edge after the setup edge.

In the Quartus II Classic Timing Analyzer, if the **Default Hold Multicycle** value is **One**, you must make two assignments to constrain the design:

- A multicycle assignment of 2
- A hold multicycle assignment of 2 to override the default value

In the Quartus II Classic Timing Analyzer, if the **Default Hold Multicycle** value is **Same as Multicycle**, only a multicycle assignment of 2 is required to constrain the design.

You should always add a hold multicycle assignment for every multicycle assignment to ensure the correct exceptions are applied regardless of the timing analyzer you use, or, for the Quartus II Classic Timing Analyzer, the **Default Hold Multicycle** setting.

Fitter Behavior

The behavior for one value of the **Optimize hold time** Fitter assignment differs between the Quartus II TimeQuest Timing Analyzer and the Quartus II Classic Timing Analyzer. When you set the Quartus II TimeQuest Timing Analyzer as the default timing analyzer, the **I/O Paths and Minimum TPD Paths** value directs the Fitter to optimize all hold time paths, which has the same affect as the **All Paths** value.

Fitter Performance

If you use the Quartus II TimeQuest Timing Analyzer as your default timing analyzer, the Fitter memory use and compilation time may increase. However, the timing analysis time may decrease.

Reporting

The Quartus II TimeQuest Timing Analyzer provides a more flexible and powerful interface for reporting timing analysis results than the Quartus II Classic Timing Analyzer. Although the interface and constraints are more flexible and powerful, both analyzers use the same device timing models, except for device families that support rise/fall analysis. The Quartus II Classic Timing Analyzer does not support rise/fall analysis, but the Quartus II TimeQuest Timing Analyzer does. Therefore, you may see slightly different delays on identical paths in device families that support rise/fall analysis if you analyze timing in both analyzers.

This means that both analyzers report identical delays along identically constrained paths in your design. The Quartus II TimeQuest Timing Analyzer allows you to constrain some paths that you could not constrain with the Quartus II Classic Timing Analyzer. Differently constrained paths result in different reported values, but for identical paths in your design that are constrained the same way, the delays are exactly the same. Both timing analyzers use the same timing models.

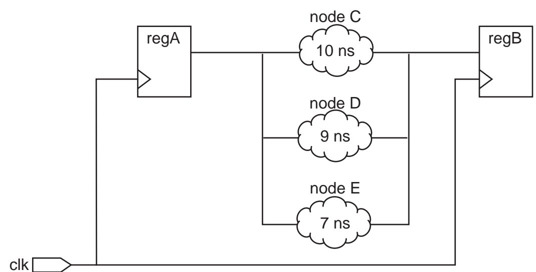
For information about reporting with the Quartus II TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Paths and Pairs

In reporting, the most significant difference between the two analyzers is that the Quartus II TimeQuest Timing Analyzer reports paths, while the Quartus II Classic Timing Analyzer reports pairs. Path reporting means that the analyzer separately reports every path between two registers. Pair reporting means that the analyzer reports only the worst-case path between two registers. One benefit of path reporting over pair reporting is that you can more easily identify common points in failing paths that may be good targets for optimization.

If your design does not meet timing constraints, this reporting difference can give the impression that there are many more timing failures when you use the Quartus II TimeQuest Timing Analyzer. Figure 9–21 shows a sample circuit followed by a description of the differences between path and pair reporting.

Figure 9–21. Failing Paths



There is an 8-ns period constraint on `clk`, resulting in two paths that fail timing: `regA` → `C` → `regB` and `regA` → `D` → `regB`. The Quartus II Classic Timing Analyzer reports only worst-case path `regA` → `C`→ `regB`. The Quartus II TimeQuest Timing Analyzer reports both failing paths `regA` → `C` → `regB` and `regA` → `D` → `regB`. It also reports path `regA` → `E` → `regB` with positive slack.

Default Reports

The Quartus II TimeQuest Timing Analyzer generates only a small number of reports by default, as compared to the Quartus II Classic Timing Analyzer, which generates every report by default. With the Quartus II TimeQuest Timing Analyzer, you generate desired reports on demand.

To learn how to create custom reports, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the Quartus II Handbook.

Netlist Names

The Quartus II Classic Timing Analyzer uses register names in reporting, but the Quartus II TimeQuest Timing Analyzer uses register pin names (with the exception of port names of the top-level module). Buried nodes or register names are used when necessary.

Example 9–5 shows how register names are used in Quartus II Classic Timing Analyzer reports.

Example 9–5. Netlist Names in the Quartus II Classic Timing Analyzer

```
Info: + Shortest register to register delay is 0.538 ns
    Info: 1: + IC(0.000 ns) + CELL(0.000 ns) = 0.000 ns; Loc. =
LCFF_X1_Y5_N1;
Fanout = 1; REG Node = 'inst'
    Info: 2: + IC(0.305 ns) + CELL(0.149 ns) = 0.454 ns; Loc. =
LCCOMB_X1_Y5_N20; Fanout = 1; COMB Node = 'inst3~feeder'
    Info: 3: + IC(0.000 ns) + CELL(0.084 ns) = 0.538 ns; Loc. =
LCFF_X1_Y5_N21; Fanout = 1; REG Node = 'inst3'
    Info: Total cell delay = 0.233 ns ( 43.31 % )
    Info: Total interconnect delay = 0.305 ns ( 56.69 % )
```

Example 9–6 shows the same information as presented in a Quartus II TimeQuest Timing Analyzer report. In this example, register pin names are used in place of register names.

Example 9–6. Netlist Names in the Quartus II TimeQuest Timing Analyzer

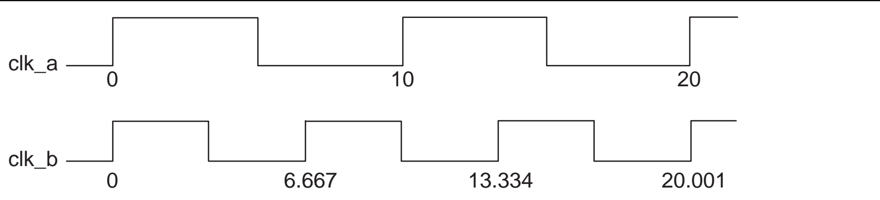
```
Info:       3.788       0.250     uTco  inst
Info:       3.788       0.000 RR  CELL  inst|regout
Info:       4.093       0.305 RR    IC  inst3~feeder|datad
Info:       4.242       0.149 RR  CELL  inst3~feeder|combout
Info:       4.242       0.000 RR    IC  inst3|datain
Info:       4.326       0.084 RR  CELL  inst3
```

Non-Integer Clock Periods

In some cases when related clock periods are not integer multiples of each other, a lack of precision in clock period definition in the Quartus II TimeQuest Timing Analyzer can result in reported setup or hold relationships of a few picoseconds. In addition, launch and latch times for the relationships can be very large. If you experience this, use the **set_max_delay** and **set_min_delay** exceptions to specify the correct relationships. The Quartus II Classic Timing Analyzer can maintain additional information about clock frequency that mitigates the lack of precision in clock period definition.

When the clock period cannot be expressed as an integer in terms of picoseconds, you have the problem detailed in Figure 9–22. This figure shows two clocks: `clk_a` has a 10 ns period, and `clk_b` has a 6.667 ns period.

Figure 9–22. Very Small Setup Relationship



There is a 1 ps setup relationship at 20 ns because you cannot specify the 6.667 ns period beyond picosecond precision. You should apply the maximum and minimum delay exceptions shown in Example 9–7 between the two clocks to specify the correct relationships.

Example 9–7. Minimum and Maximum Delay Exceptions

```
set_max_delay -from [get_clocks clk_a] -to [get_clocks clk_b] 3.333
set_min_delay -from [get_clocks clk_a] -to [get_clocks clk_b] 0
```

Other Features

The Quartus II TimeQuest Timing Analyzer reports time values without units. By default, the units are nanoseconds, and three decimal places are displayed. You can change the default time unit and decimal places with the **set_time_format** command.

When you use the Quartus II TimeQuest Timing Analyzer in a Tcl shell, output is ASCII-formatted, and columns are aligned for easy reading on 80-column consoles. Example 9–8 shows sample output from a **report_timing** command from the Quartus II TimeQuest Timing Analyzer.

Example 9–8. ASCII-Formatted Quartus II TimeQuest Timing Analyzer Report

```
tcl> report_timing -from inst -to inst5
Info: Report Timing: Found 1 setup paths (0 violated).  Worst case slack
is 3.634
Info: -from [get_keepers inst]
Info: -to [get_keepers inst5]
Info: Path #1: Slack is 3.634
Info:
===================================================================
Info: From Node    : inst
Info: To Node      : inst5
Info: Launch Clock : clk_a
Info: Latch Clock  : clk_b
Info:
Info: Data Arrival Path:
Info:
Info: Total (ns)  Incr (ns)     Type  Node
Info: ==========  ========= ==  ====
===================================
Info:      0.000      0.000           launch edge time
Info:      2.347      2.347  R        clock network delay
Info:      2.597      0.250     uTco  inst
Info:      2.597      0.000 RR  CELL  inst|regout
Info:      3.088      0.491 RR    IC  inst6|datac
Info:      3.359      0.271 RR  CELL  inst6|combout
Info:      3.359      0.000 RR    IC  inst5|datain
Info:      3.443      0.084 RR  CELL  inst5
Info:
Info: Data Required Path:
Info:
Info: Total (ns)  Incr (ns)     Type  Node
Info: ==========  ========= ==  ====
===================================
Info:      4.000      4.000           latch edge time
Info:      7.041      3.041  R        clock network delay
Info:      7.077      0.036     uTsu  inst5
Info:
Info: Data Arrival Time  :     3.443
Info: Data Required Time :     7.077
Info: Slack              :     3.634
Info:
===================================================================
Info:
1 3.634
```

Scripting API

In versions of the Quartus II software earlier than 6.0, the **::quartus::project** Tcl package contained the following SDC-like commands for making timing assignments:

- **create_base_clock**
- **create_relative_clock**
- **get_clocks**
- **set_clock_latency**
- **set_clock_uncertainty**
- **set_input_delay**
- **set_multicycle_assignment**

- **set_output_delay**
- **set_timing_cut_assignment**

These commands are not SDC-compliant. Beginning with version 6.0, these commands are in a new package called **::quartus::timing_assignment**. To ensure backward compatibility with existing Tcl scripts, the **::quartus::timing_assignment** package is loaded by default in the following executables:

- **quartus**
- **quartus_sh**
- **quartus_cdb**
- **quartus_sim**
- **quartus_stp**
- **quartus_tan**

The **::quartus::timing_assignment** package is not loaded by default in the **quartus_sta** executable. The **::quartus::sdc** Tcl package includes SDC-compliant versions of the commands listed above. That package is available only in the **quartus_sta** executable, and it is loaded by default.

Timing Assignment Conversion

This section describes Quartus II Classic QSF timing assignments and their equivalent Quartus II TimeQuest constraints. You can convert many Quartus II Classic timing assignments to SDC constraints. Some Quartus II Classic timing assignments can be converted to two different SDC constraints, and you must understand the intended functionality of the design to make an appropriate conversion. You cannot convert some Quartus II Classic timing assignments because there is no equivalent SDC constraint.

This section includes the following topics, arranged alphabetically:

- "Clock Enable Multicycle" on page 9–30
- "Clock Latency" on page 9–27
- "Clock Uncertainty" on page 9–28
- "Cut Timing Path" on page 9–41
- "Default Required f_{MAX} Assignment" on page 9–29
- "Hold Relationship" on page 9–27
- "Input and Output Delay" on page 9–31
- "Inverted Clock" on page 9–28
- "Maximum Clock Arrival Skew" on page 9–42
- "Maximum Data Arrival Skew" on page 9–42
- "Maximum Delay" on page 9–41
- "Minimum Delay" on page 9–42
- "Minimum t_{CO} Requirement" on page 9–38

- "Minimum t_{PD} Requirement" on page 9–41
- "Multicycle" on page 9–30
- "Not a Clock" on page 9–28
- "Setup Relationship" on page 9–27
- "t_{CO} Requirement" on page 9–36
- "t_{H} Requirement" on page 9–34
- "t_{PD} Requirement" on page 9–40
- "t_{SU} Requirement" on page 9–32
- "Virtual Clock Reference" on page 9–29

Setup Relationship

The **Setup Relationship** assignment overrides the setup relationship between two clocks. By default, the Quartus II Classic Timing Analyzer automatically calculates the setup relationship based on your clock settings. The QSF variable for the **Setup Relationship** assignment is SETUP_RELATIONSHIP. In the Quartus II TimeQuest Timing Analyzer, use the **set_max_delay** command to specify the maximum setup relationship for a path.

The setup relationship value is the time between latch and launch edges before the Quartus II TimeQuest Timing Analyzer accounts for clock latency, source μt_{CO}, or destination μt_{SU}.

Hold Relationship

The **Hold Relationship** assignment overrides the hold relationship between two clocks. By default, the Quartus II Classic Timing Analyzer automatically calculates the hold relationship based on your clock settings. The QSF variable for the **Hold Relationship** assignment is HOLD_RELATIONSHIP. In the Quartus II TimeQuest Timing Analyzer, use the **set_min_delay** command to specify the minimum hold relationship for a path.

Clock Latency

Table 9–1 shows the equivalent SDC constraints for each of these Quartus II Classic assignments.

Table 9–1. Quartus II Classic and SDC Equivalent Constraints

Quartus II Classic Timing Assignment		
Assignment Name	**QSF Variable**	**SDC Constraint**
Early Clock Latency	EARLY_CLOCK_LATENCY	set_clock_latency -source -late
Late Clock Latency	LATE_CLOCK_LATENCY	set_clock_latency -source -early

For more information about clock latency support in the Quartus II TimeQuest Timing Analyzer, refer to "Clock Latency" on page 9–13.

Clock Uncertainty

This section describes the conversion for the following Quartus II Classic Timing Analyzer assignments:

- Clock Setup Uncertainty
- Clock Hold Uncertainty

Table 9–2 shows the equivalent SDC constraints for each of these Quartus II Classic Timing Analyzer assignments.

Table 9–2. Quartus II Classic and SDC Equivalent Constraints

Quartus II Classic Timing Analyzer Timing Assignment		SDC Constraint
Assignment Name	**QSF Variable**	
Clock Setup Uncertainty	CLOCK_SETUP_UNCERTAINTY	set_clock_uncertainty -setup
Clock Hold Uncertainty	CLOCK_HOLD_UNCERTAINTY	set_clock_uncertainty -hold

Inverted Clock

The Quartus II Classic Timing Analyzer detects inverted clocks automatically when the clock inversion occurs at the input of the LCELL that contains the register specified in the assignment. You must make an **Inverted Clock** assignment in all other situations for Quartus II Classic Timing Analyzer analysis. The QSF variable for the **Inverted Clock** assignment is INVERTED_CLOCK. The Quartus II TimeQuest Timing Analyzer detects inverted clocks automatically, regardless of the type of inversion circuit, in designs that target device families that support unateness: Stratix® II, Cyclone® II, and HardCopy® II. For designs that target any other device family, you must create a generated clock with the `-invert` option on the output of the cell that inverts the clock.

For more information about unateness, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Not a Clock

The **Not a Clock** assignment directs the Quartus II Classic Timing Analyzer that the specified node is not a clock source when it would normally be detected as a clock because of a global f_{MAX} requirement. The QSF variable for the **Not a Clock** assignment is NOT_A_CLOCK. This assignment is not supported in the Quartus II TimeQuest Timing Analyzer and there is no equivalent constraint. Appropriate clock constraints are created in the Quartus II TimeQuest Timing Analyzer only.

Default Required f_{MAX} Assignment

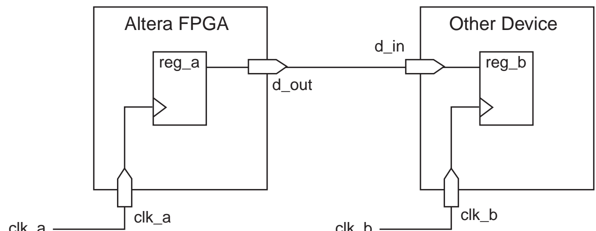
The **Default Required** $\mathbf{f_{MAX}}$ assignment allows you to specify an f_{MAX} requirement for the Quartus II Classic Timing Analyzer for all unconstrained clocks in your design. The QSF variable for the **Default Required** $\mathbf{f_{MAX}}$ assignment is FMAX_REQUIREMENT. You can use the **derive_clocks** command to create clocks on sources of clock pins in your design that do not already have clocks assigned to them. You should constrain each individual clock in your design with the **create_clock** or **created_generated_clock** command, not the **derive_clocks** command. Refer to "Automatic Clock Detection" on page 9–16 to learn why you should constrain individual clocks in your design.

Virtual Clock Reference

The **Virtual Clock Reference** assignment allows you to define timing characteristics of a reference clock outside the FPGA. The QSF variable for the **Virtual Clock Reference** assignment is VIRTUAL_CLOCK_REFERENCE. The Quartus II TimeQuest Timing Analyzer supports virtual clocks by default, while the Quartus II Classic Timing Analyzer requires the **Virtual Clock Reference** assignment to indicate that a clock setting is for a virtual clock. To create a virtual clock in the Quartus II TimeQuest Timing Analyzer, use the **create_clock** or **create_generated_clock** commands with the `-name` option and no targets.

Figure 9–23 shows a simple circuit that requires a virtual clock, and the following example shows how to constrain the circuit. The circuit shows data transfer between an Altera FPGA and another device, and the clocks for the two devices are not related. You can constrain the path with an output delay assignment, but that assignment requires a virtual clock that defines the clock characteristics of the destination device.

Figure 9–23. Virtual Clock Sample Circuit



Assume the circuit has the following assignments in the Quartus II Classic Timing Analyzer:

- Clock period of 10 ns on `system_clk` (clock for the Altera FPGA)
- Clock period of 8 ns on `virt_clk` (clock for the other device)
- Virtual Clock Reference setting for `virt_clk` is on (indicates that `virt_clk` is a virtual clock)
- Output Maximum Delay of 5 ns on `dataout` with respect to `virt_clk` (constrains the path between the two devices)

The SDC commands shown in Example 9–9 constrain the circuit the same way.

Example 9–9. SDC Constraints

```
# Clock for the Altera FPGA
create_clock -period 10 -name system_clk [get_ports system_clk]
# Virtual clock for the other device, with no targets
create_clock -period 8 -name virt_clk
# Constrains the path between the two devices
set_output_delay -clock virt_clk 5 [get_ports dataout]
```

Clock Settings

The Quartus II Classic Timing Analyzer includes a variety of assignments to describe clock settings. These include duty cycle, f_{MAX}, offset, and others. In the Quartus II TimeQuest Timing Analyzer, use the **create_clock** and **create_generated_clock** commands to constrain clocks.

Multicycle

Table 9–3 shows the equivalent SDC exceptions for each of these Quartus II Classic Timing Analyzer timing assignments.

Table 9–3. Quartus II Classic and SDC Equivalent Exceptions

Quartus II Classic Timing Assignment		SDC Exception
Assignment Name	**QSF Variable**	
Multicycle *(1)*	MULTICYCLE	set_multicycle_path -setup -end
Source Multicycle *(2)*	SRC_MULTICYCLE	set_multicycle_path -setup -start
Multicycle Hold *(3)*	HOLD_MULTICYCLE	set_multicycle_path -hold -end
Source Multicycle Hold	SRC_HOLD_MULTICYCLE	set_multicycle_path -hold -start

Notes to Table 9–3:

(1) A multicycle assignment is also known as a “destination multicycle setup” assignment.

(2) A source multicycle assignment is also known as a “source multicycle setup” assignment.

(3) A multicycle hold is also known as a “destination multicycle hold “assignment.

The default value and numbering scheme for the hold multicycle value is different in the Quartus II Classic and Quartus II TimeQuest Timing Analyzers. Refer to “Hold Multicycle” on page 9–20 for more information about the difference between the default value and numbering scheme for the hold multicycle value in the Quartus II Classic and Quartus II TimeQuest Timing Analyzers.

For more information about how to convert the hold multicycle value, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Clock Enable Multicycle

The Quartus II Classic Timing Analyzer supports the following clock enable multicycle assignments. Corresponding types of multicycle assignments are applied to all registers enabled by the targets of the specified clock.

- Clock Enable Multicycle
- Clock Enable Source Multicycle

- Clock Enable Multicycle Hold
- Clock Enable Source Multicycle Hold

The Quartus II TimeQuest Timing Analyzer supports clock-enabled multicycle constraints with the **get_fanouts** command. Use the **get_fanouts** command to create a collection of nodes that have a common source signal, such as a clock enable.

I/O Constraints

FPGA I/O timing assignments have typically been made with FPGA-centric t_{SU} and t_{CO} requirements for the Quartus II Classic Timing Analyzer. However, the Quartus II Classic Timing Analyzer also supports input and output delay assignments to accommodate industry-standard, system-centric timing constraints. Where possible, you should use system-centric constraints to constrain your designs for the Quartus II TimeQuest Timing Analyzer. Table 9–4 includes Quartus II Classic I/O assignments, the equivalent FPGA-centric SDC constraints, and recommended system-centric SDC constraints.

For setup checks (t_{SU} and t_{CO}), <*latch – launch*> equals the clock period for same-clock transfers. For hold checks (t_H and Minimum t_{CO}), <*latch – launch*> equals 0 for same-clock transfers. Conversions from Quartus II Classic assignments to `set_input_delay` and `set_output_delay` constraints work well only when the source and destination registers' clocks are the same (same clock and polarity). If the source and destination registers' clocks are different, the conversion may not be straightforward and you should take extra care when converting to `set_input_delay` and `set_output_delay` constraints.

Table 9–4. Quartus II Classic and Quartus II TimeQuest Timing Analyzers Equivalent I/O Constraints

Classic	FPGA-centric SDC	System-centric SDC
t_{SU} Requirement	set_max_delay <*t_{SU} requirement*>	set_input_delay -max <*latch – launch – t_{SU} requirement*>
t_H Requirement	set_min_delay – <*t_H requirement*> *(1)*	set_input_delay -min <*latch – launch + t_H requirement*>
t_{CO} Requirement	set_max_delay <*t_{CO} requirement*>	set_output_delay -max <*latch – launch – t_{CO} requirement*>
Minimum t_{CO} Requirement	set_min_delay <*minimum t_{CO} requirement*>	set_output_delay -min <*latch – launch – minimum t_{CO} requirement*>
t_{PD} Requirement	set_max_delay <*t_{PD} requirement*>	*(2)*
Minimum t_{PD} Requirement	set_min_delay <*minimum t_{PD} requirement*>	*(2)*

Notes to Table 9–4:

(1) Refer to "t_H Requirement" on page 9–34 for an explanation about why this exception uses the negative t_H requirement.

(2) The input and output delays can be used for t_{PD} paths, such that they will be analyzed as a system f_{MAX} path. This is a feature unique to the Quartus II TimeQuest Timing Analyzer.

Input and Output Delay

Table 9–5 shows the equivalent SDC exceptions for each of these Quartus II Classic Timing Analyzer timing assignments.

Table 9–5. Quartus II Classic and SDC Equivalent Exceptions

Quartus II Classic Timing Assignment		SDC Exception
Assignment Name	**QSF Variable**	
Input Maximum Delay	INPUT_MAX_DELAY	set_input_delay -max
Input Minimum Delay	INPUT_MIN_DELAY	set_input_delay -min
Output Maximum Delay	OUTPUT_MAX_DELAY	set_output_delay -max
Output Minimum Delay	OUTPUT_MIN_DELAY	set_output_delay -min

In some circumstances, you may receive the following warning message when you update the SDC netlist:

```
Warning: For set_input_delay/set_output_delay, port "<port>" does not
have delay for flag (<rise|fall>, <min|max>)
```

This warning occurs whenever port constraints have maximum or minimum delay assignments, but not both. In the Quartus II Classic Timing Analyzer, device inputs can have **Input Maximum Delay** assignments, **Input Minimum Delay** assignments, or both, and device outputs can have **Output Maximum Delay** assignments, **Output Minimum Delay** assignments, or both.

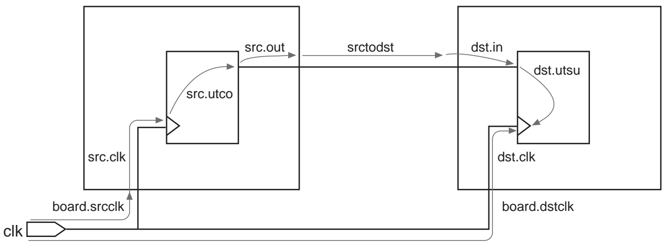
To avoid receiving the warning, your **.sdc** file must specify both the `-max` and `-min` options for each port, or specify neither. If a device I/O in your design includes both the maximum and minimum delay assignments in the Quartus II Classic Timing Analyzer, the conversion utility converts both, and no warning appears about that device I/O. If a device I/O has only maximum or minimum delay assignments in the Quartus II Classic Timing Analyzer, you have the following options:

- Add the missing minimum or maximum delay assignment to the device I/O before performing the conversion.
- Modify the SDC constraint after the conversion to add appropriate `-max` or `-min` values.
- Modify the SDC constraint to remove the `-max` or `-min` option so the value is used for both by default.

t_{SU} Requirement

The **t_{SU} Requirement** assignment specifies the maximum acceptable clock setup time for the input (data) pin. The QSF variable for the **t_{SU} Requirement** assignment is TSU_REQUIREMENT. You can convert the **t_{SU} Requirement** assignment to the **set_max_delay** command or the **set_input_delay** command with the `-max` option. The delay value for the **set_input_delay** command is <*latch – launch – t_{SU} requirement*>. Refer to the labeled paths in Figure 9–24 to understand the names in Equation 9–3 and Equation 9–4.

Figure 9–24. Path Names



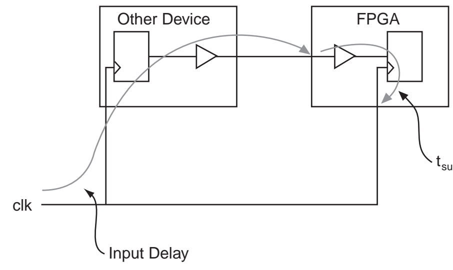
Equation 9–3 shows the derivation of this conversion.

Equation 9–3.

$$\text{required} - \text{arrival} > 0$$
$$\text{required} = \text{latch} + \text{board.dstclk} + \text{dst.clk} - \text{dst.utsu}$$
$$\text{arrival} = \text{launch} + \text{board.srcclk} + \text{src.clk} + \text{src.utco} + \text{src.out} + \text{srctodst} + \text{dst.in}$$
$$\text{input_delay} = \text{board.srcclk} + \text{src.clk} + \text{src.utcu} + \text{src.out} + \text{srctodst} - \text{board.dstclk}$$
$$\text{required} = \text{latch} + \text{dst.clk} - \text{dst.utsu}$$
$$\text{arrival} = \text{launch} + \text{input_delay} + \text{dst.in}$$
$$(\text{latch} + \text{dst.clk} - \text{dst.utsu}) - (\text{launch} + \text{input_delay} + \text{dst.in}) > 0$$
$$t_{su}\,\text{requirement} - \text{actual}\ t_{su} > 0$$
$$\text{actual}\ t_{su} = \text{dst.in} + \text{dst.utsu} - \text{dst.clk}$$
$$t_{su}\,\text{requirement} - (\text{dst.in} + \text{dst.utsu} - \text{dst.clk}) > 0$$
$$t_{su}\,\text{requirement} = \text{latch} - \text{launch} - \text{input_delay}$$
$$\text{input_delay} = \text{latch} - \text{launch} - t_{su}\,\text{requirement}$$

The delay value is the difference between the period of the clock source of the register and the t_{SU} **Requirement** value, as shown in Figure 9–25.

Figure 9–25. t_{SU} Requirement



The delay value for the **set_max_delay** command is the t_{SU} **Requirement** value. Equation 9–4 shows the derivation of this conversion.

Equation 9–4.

$$\text{required} - \text{arrival} > 0$$
$$\text{required} = \text{latch} + \text{board.dstclk} + \text{dst.clk} - \text{dst.utsu}$$
$$\text{arrival} = \text{launch} + \text{board.srcclk} + \text{src.clk} + \text{src.utco} + \text{src.out} + \text{srctodst} + \text{dst.in}$$
$$\text{max_delay} = \text{latch} + \text{board.dstclk} + - \text{launch} - \text{board.srcclk} - \text{src.clk} - \text{src.out} - \text{srctodst}$$
$$\text{required} = \text{max_delay} + \text{dst.clk} - \text{dst.utsu}$$
$$\text{arrival} = \text{dst.in}$$
$$(\text{max_delay} + \text{dst.clk} - \text{dst.utsu}) - (\text{dst.in}) > 0$$
$$t_{su}\ \text{requirement} - t_{su} > 0$$
$$\text{actual}\ t_{su} = \text{dst.in} + \text{dst.utsu} - \text{dst.clk}$$
$$t_{su}\ \text{requirement} - (\text{dst.in} + \text{dst.utsu} - \text{dst.clk}) > 0$$
$$\text{set_max_delay} = t_{su}\ \text{requirement}$$

Table 9–6 shows the different ways you can make t_{SU} assignments in the Quartus II Classic Timing Analyzer, and the corresponding options for the **set_max_delay** exception.

Table 9–6. t_{SU} Requirement and set_max_delay Equivalence

t_{SU} Requirement Options	set_max_delay Options
-to <*pin*>	-from [get_ports <*pin*>] -to [get_registers *]
-to <*clock*>	-from [get_ports *] -to [get_clocks <*clock*>]
-to <*register*>	-from [get_ports *] -to [get_registers <*register*>]
-from <*pin*> -to <*register*>	-from [get_ports <*pin*>] -to [get_registers <*register*>]
-from <*clock*> -to <*pin*>	-from [get_ports <*pin*>] -to [get_clocks <*clock*>] *(1)*

Notes to Table 9–6:

(1) If the pin in this assignment feeds registers clocked by the same clock, it is equivalent to the first option, `-to` <*pin*>. If the pin feeds registers clocked by different clocks, use **set_input_delay** to constrain the paths properly.

To convert a global t_{SU} assignment to an equivalent SDC exception, use the command shown in Example 9–10.

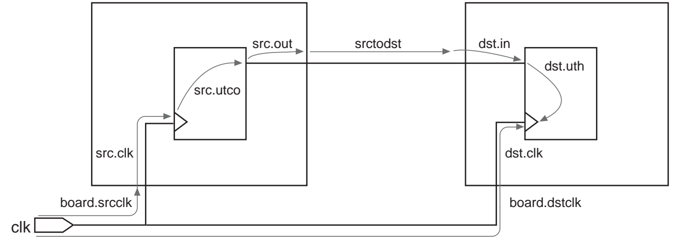
Example 9–10. Converting a Global t_{SU} Assignment to an Equivalent SDC Exception

```
set_max_delay -from [all_inputs] -to [all_registers] <tSU value>
```

t_H Requirement

The **t_H Requirement** specifies the maximum acceptable clock hold time for the input (data) pin. The QSF variable for the **t_H Requirement** assignment is TH_REQUIREMENT. You can convert the **t_H Requirement** assignment to the **set_min_delay** command, or the **set_input_delay** command with the `-min` option. The delay value for the **set_input_delay** command is <*latch – launch + t_H requirement*>. Refer to the labeled paths in Figure 9–26 to understand the names in Equation 9–5 and Equation 9–6.

Figure 9–26. Path Names



Equation 9–5 shows the derivation of this calculation.

Equation 9–5.

$$\text{arrival} - \text{required} > 0$$
$$\text{arrival} = \text{launch} + \text{board.srcclk} + \text{src.clk} + \text{src.utco} + \text{src.out} + \text{srctodst} + \text{dst.in}$$
$$\text{required} = \text{latch} + \text{board.dstclk} + \text{dst.clk} + \text{dst.uth}$$
$$\text{input_delay} = \text{board.srcclk} + \text{src.clk} + \text{srcutcu} + \text{src.out} + \text{srctodst} - \text{board.dstclk}$$
$$\text{arrival} = \text{launch} + \text{input_delay} + \text{dst.in}$$
$$\text{required} = \text{latch} + \text{dst.clk} + \text{dst.uth}$$
$$(\text{launch} + \text{input_delay} + \text{dst.in}) - (\text{latch} + \text{dst.clk} + \text{dst.uth}) > 0$$
$$t_H \text{ requirement} - \text{actual } t_H > 0$$
$$\text{actual } t_H = \text{dst.clk} + \text{dst.uth} - \text{dst.in}$$
$$t_H \text{ requirement} - (\text{dst.clk} + \text{dst.uth} - \text{dst.in}) > 0$$
$$t_H \text{ requirement} = \text{launch} - \text{latch} + \text{input_delay}$$
$$\text{input_delay} = \text{latch} - \text{launch} + t_H \text{ requirement}$$

The delay value for the **set_min_delay** command is the $\mathbf{t_H}$ **Requirement** value. Equation 9–6 shows the derivation of this conversion.

Equation 9–6.

$$\text{arrival} - \text{required} > 0$$
$$\text{arrival} = \text{dst.in}$$
$$\text{required} = \text{min_delay} + \text{dst.clk} + \text{dst.uth}$$
$$\text{dst.in} - (\text{min_delay} + \text{dst.clk} + \text{dst.uth})$$
$$t_H \text{ requirement} - \text{actual } t_H > 0$$
$$\text{actual } t_H = \text{dst.clk} + \text{dst.uth} - \text{dst.in}$$
$$t_H \text{ requirement} - (\text{dst.clk} + \text{dst.uth} - \text{dst.in}) > 0$$
$$\text{set_min_delay} = -t_H \text{ requirement}$$

Table 9–7 shows the different ways you can make t_H assignments in the Quartus II Classic Timing Analyzer, and the corresponding options for the **set_min_delay** command.

Table 9–7. t_H Requirement and set_min_delay Equivalence

t_H Requirement Options	set_min_delay Options
-to <*pin*>	-from [get_ports <*pin*>] -to [get_registers *]
-to <*clock*>	-from [get_ports *] -to [get_clocks <*clock*>]
-to <*register*>	-from [get_ports *] -to [get_registers <*register*>]
-from <*pin*> -to <*register*>	-from [get_ports <*pin*>] -to [get_registers <*register*>]
-from <*clock*> -to <*pin*>	-from [get_ports <*pin*>] -to [get_clocks <*clock*>] *(1)*

Note to Table 9–7:

(1) If the pin in this assignment feeds registers clocked by the same clock, it is equivalent to the first option, `-to` <*pin*>. If the pin feeds registers clocked by different clocks, use **set_input_delay** to constrain the paths properly. Refer to "Input and Output Delay" on page 9–31 for additional information.

To convert a global t_H assignment to an equivalent SDC exception, use the command shown in Example 9–11.

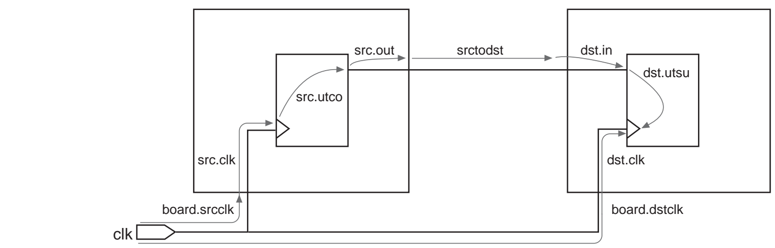
Example 9–11. Converting a Global t_H Assignment to an Equivalent SDC Exception

```
set_min_delay -from [all_inputs] -to [all registers] <negative tH value>
```

t_{CO} Requirement

The **t_{CO} Requirement** assignment specifies the maximum acceptable clock to output delay to the output pin. The QSF variable for the **t_{CO} Requirement** assignment is TCO_REQUIREMENT. You can convert the **t_{CO} Requirement** assignment to the **set_max_delay** command or the **set_output_delay** with the `-max` option. The delay value for the **set_output_delay** command is <*latch – launch + tCO requirement*>. Refer to the labeled paths in Figure 9–27 to understand the names in Equation 9–7 and Equation 9–8.

Figure 9–27. Path Names



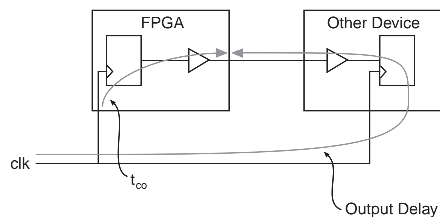
Equation 9–7 shows the derivation of this conversion.

Equation 9–7.

$$\text{required} - \text{arrival} > 0$$
$$\text{required} = \text{latch} - \text{output_delay}$$
$$\text{arrival} = \text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$\text{output_delay} = \text{srctodst} + \text{dst.in} + \text{dst.utsu} - \text{dst.clk} - \text{board.dstc.k} + \text{board.srcclk}$$
$$(\text{latch} - \text{output_delay}) - (\text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}) > 0$$
$$t_{CO}\ \text{requirement} - \text{actual}\ t_{CO} > 0$$
$$\text{actual}\ t_{CO} = \text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$t_{CO}\ \text{requirement} - (\text{src.clk} + \text{src.utco} + \text{src.out}) > 0$$
$$t_{CO}\ \text{requirement} = \text{latch} - \text{launch} - \text{output_delay}$$
$$\text{output_delay} = \text{latch} - \text{launch} - t_{CO}\ \text{requirement}$$

The delay value is the difference between the period of the clock source of the register and the t_{CO} **Requirement** value, as illustrated in Figure 9–28.

Figure 9–28. t_{CO} Requirement



The delay value for the **set_max_delay** command is the t_{CO} **Requirement** value. Equation 9–8 shows the derivation of this conversion.

Equation 9–8.

$$\text{required} - \text{arrival} > 0$$
$$\text{required} = \text{set_max_delay}$$
$$\text{arrival} = \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$\text{set_max_delay} - (\text{src.clk} + \text{src.utco} + \text{src.out}) > 0$$
$$t_{CO}\ \text{requirement} - \text{actual}\ t_{CO} > 0$$
$$\text{actual}\ t_{CO} = \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$t_{CO}\ \text{requirement} - (\text{src.clk} + \text{src.utco} + \text{src.out}) > 0$$
$$\text{set_max_delay} = t_{CO}\ \text{requirement}$$

Table 9–8 shows the different ways you can make t_{CO} assignments in the Quartus II Classic Timing Analyzer, and the corresponding options for the **set_max_delay** exception.

Table 9–8. t_{CO} Requirement and set_max_delay Equivalence

t_{CO} Requirement Options	set_max_delay Options
-to <*pin*>	-from [get_registers *] -to [get_ports <*pin*>]
-to <*clock*>	-from [get_clocks <*clock*>] -to [get_ports *]
-to <*register*>	-from [get_registers <*register*>] -to [get_ports *]
-from <*register*> -to <*pin*>	-from [get_registers <*register*>] -to [get_ports <*pin*>]
-from <*clock*> -to <*pin*>	-from [get_clocks <*clock*>] -to [get_ports <*pin*>] *(1)*

Note to Table 9–8:

(1) If the pin in this assignment feeds registers clocked by the same clock, it is equivalent to the first option, `-to` <*pin*>. If the pin feeds registers clocked by different clocks, you should use **set_output_delay** to constrain the paths properly.

To convert a global t_{CO} assignment to an equivalent SDC exception, use the command in Example 9–12.

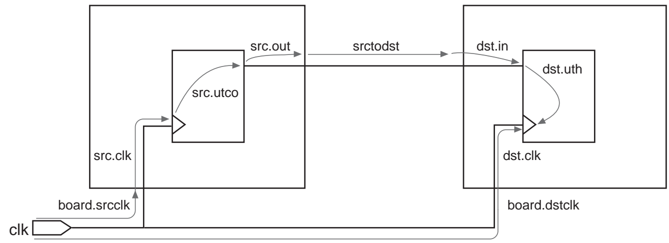
Example 9–12. Converting a Global t_{CO} Assignment to an Equivalent SDC Exception

```
set_max_delay -from [all registers] -to [all_outputs] <tCO value>
```

Minimum t_{CO} Requirement

The **Minimum t_{CO} Requirement** assignment specifies the minimum acceptable clock to output delay to the output pin. The QSF variable for the **Minimum t_{CO} Requirement** assignment is MIN_TCO_REQUIREMENT. You can convert the **Minimum t_{CO} Requirement** assignment to the **set_min_delay** command or the **set_output_delay** command with the `-min` option. The delay value for the **set_output_delay** command is <*latch – launch + minimum t_{CO} requirement*>. Refer to the labeled paths in Figure 9–29 to understand the names in Equation 9–9 and Equation 9–10.

Figure 9–29. Path Names



Equation 9–9 shows the derivation of this conversion.

Equation 9–9.

$$arrival + required > 0$$
$$arrival = launch + src.clk + src.utco + src.out$$
$$required = latch - output_delay$$
$$output_delay = srctodst + dst.in - dst.uth - dst.clk - board.dstclk + board.srcclk$$
$$(launch + src.clk + src.utco + src.out) - (latch - output_delay) > 0$$
$$minimum\ t_{co} - minimum\ t_{co} requirement > 0$$
$$minimum\ t_{co} = launch + src.clk + src.utco + src.out$$
$$(launch + src.clk + src.utco + src.out) - minimum\ t_{co}\ requirement > 0$$
$$minimum\ t_{co}\ requirement = latch - launch - output_delay$$
$$output_delay = latch - launch - minimum\ t_{co}\ requirement$$

The delay value for the **set_min_delay** command is the **Minimum t_{CO} Requirement**. Equation 9–10 shows the derivation of this conversion.

Equation 9–10.

$$arrival - required > 0$$
$$arrival = src.clk + src.utco + src.out$$
$$required = min_delay$$
$$(src.clk + src.utco + src.out) - (set_min_delay) > 0$$
$$minimum\ t_{co} - minimum\ t_{co}\ requirement > 0$$
$$minimum\ t_{co} = src.clk + src.utco + src.out$$
$$(src.clk + src.utco + src.out) - minimum\ t_{co} requirement > 0$$
$$set_min_delay = minimum\ t_{co}\ requirement$$

Table 9–9 shows the different ways you can make minimum t_{CO} assignments in the Quartus II Classic Timing Analyzer, and the corresponding options for the **set_min_delay** exception.

Table 9–9. Minimum t_{CO} Requirement and set_min_delay Equivalence

Minimum t_{co} Requirement Options	set_min_delay Options
-to <*pin*>	-from [get_registers *] -to [get_ports <*pin*>]
-to <*clock*>	-from [get_clocks <*clock*>] -to [get_ports *]
-to <*register*>	-from [get_registers <*register*>] -to [get_ports *]
-from <*register*> -to <*pin*>	-from [get_registers <*register*>] -to [get_ports <*pin*>]
-from <*clock*> -to <*pin*>	-from [get_clocks <*clock*>] -to [get_ports <*pin*>] *(1)*

Note to Table 9–9:

(1) If the pin in this assignment feeds registers clocked by the same clock, it is equivalent to the first option, `-to <pin>`. If the pin feeds registers clocked by different clocks, use **set_output_delay** to constrain the paths properly.

To convert a global **Minimum t_{CO} Requirement** to an equivalent SDC exception, use the command shown in Example 9–13.

Example 9–13. Converting a Global Minimum t_{CO} Requirement to an Equivalent SDC Exception

```
set_min_delay -from [all_registers] -to [all_outputs] <minimum tCO value>
```

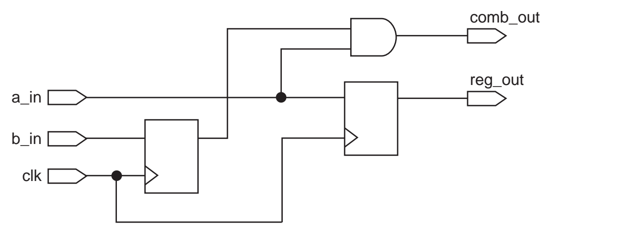
t_{PD} Requirement

The **t_{PD} Requirement** assignment specifies the maximum acceptable input to non-registered output delay, that is, the time required for a signal from an input pin to propagate through combinational logic and appear at an output pin. The QSF variable for the **t_{PD} Requirement** assignment is TPD_REQUIREMENT. You can use the **set_max_delay** command in the Quartus II TimeQuest Timing Analyzer as an equivalent constraint as long as you account for input and output delays. The **t_{PD} Requirement** assignment does not take into account input and output delays, but the **set_max_delay** exception does, so you must modify the **set_max_delay** value to take into account input and output delays.

Combinational Path Delay Scenario

Figure 9–30 shows a simple circuit followed by an example of a **t_{PD} Requirement** to **set_max_delay** conversion.

Figure 9–30. t_{PD} Example



Assume the circuit has the following assignments in the Quartus II Classic Timing Analyzer:

- Clock period of 10 ns
- **t_{PD} Requirement** from `a_in` to `comb_out` of 10 ns
- **Input Max Delay** on `a_in` relative to `clk` of 2 ns
- **Output Max Delay** on `comb_out` relative to `clk` of 2 ns

The path from `a_in` to `comb_out` is not affected by the input and output delays. The slack is equal to the <*t_{PD} Requirement from* `a_in` *to* `comb_out`> – <*path delay from* `a_in` *to* `comb_out`>.

Assume the circuit has the SDC constraints shown in Example 9–14 in the Quartus II TimeQuest Timing Analyzer.

Example 9–14. SDC Constraints

```
create_clock -period 10 -name clk [get_ports clk]
set_max_delay -from a_in -to comb_out 10
set_input_delay -clk clk 2 [get_ports a_in]
set_output_delay -clk clk 2 [get_ports comb_out]
```

The path from `a_in` to `comb_out` is affected by the input and output delays. The slack is equal to:

```
<set_max_delay value from a_in to comb_out>    <input delay>    <output delay>    <path delay from a_in to comb_out>
```

To convert a global **t_{PD} Requirement** assignment to an equivalent SDC exception, use the command shown in Example 9–15. You should add the input and output delays to the value of your converted **t_{PD} Requirement** (**set_max_delay** exception value) to achieve an equivalent SDC exception.

Example 9–15. Converting a Global t_{PD} Requirement Assignment to an Equivalent SDC Exception

```
set_max_delay -from [all_inputs] -to [all_outputs] <value>
```

Minimum t_{PD} Requirement

The **Minimum t_{PD} Requirement** assignment specifies the minimum acceptable input to non-registered output delay, that is, the minimum time required for a signal from an input pin to propagate through combinational logic and appear at an output pin. The QSF variable for the **Minimum t_{PD} Requirement** assignment is MIN_TPD_REQUIREMENT. You can use the **set_min_delay** command in the Quartus II TimeQuest Timing Analyzer as an equivalent constraint as long as you account for input and output delays. The **Minimum t_{PD} Requirement** assignment does not take into account input and output delays, but the **set_min_delay** exception does.

Refer to "Combinational Path Delay Scenario" on page 9–40 to see how input and output delays affect minimum and maximum delay exceptions.

To convert a global **Minimum t_{PD} Requirement** assignment to an equivalent SDC exception, use the command shown in Example 9–16.

Example 9–16. Converting a Global Minimum t_{PD} Requirement Assignment to an Equivalent SDC Exception

```
set_min_delay -from [all_inputs] -to [all_outputs] <value>
```

Cut Timing Path

The **Cut Timing Path** assignment in the Quartus II Classic Timing Analyzer is equivalent to the **set_false_path** command in the Quartus II TimeQuest Timing Analyzer. The QSF variable for the **Cut Timing Path** assignment is CUT.

Maximum Delay

The **Maximum Delay** assignment specifies the maximum required delay for the following types of paths:

- Pins to registers
- Registers to registers
- Registers to pins

The QSF variable for the **Maximum Delay** assignment is MAX_DELAY. This overwrites the requirement computed from the clock setup relationship and clock skew. There is no equivalent constraint in the Quartus II TimeQuest Timing Analyzer.

The **Maximum Delay** assignment for the Quartus II Classic Timing Analyzer is not related to the **set_max_delay** exception in the Quartus II TimeQuest Timing Analyzer.

Minimum Delay

The **Minimum Delay** assignment specifies the minimum required delay for the following types of paths:

- Pins to registers
- Registers to registers
- Registers to pins

The QSF variable for the **Minimum Delay** assignment is MIN_DELAY. This overwrites the requirement computed from the clock hold relationship and clock skew. There is no equivalent constraint in the Quartus II TimeQuest Timing Analyzer.

The **Minimum Delay** assignment for the Quartus II Classic Timing Analyzer is not related to the **set_min_delay** exception in the Quartus II TimeQuest Timing Analyzer.

Maximum Clock Arrival Skew

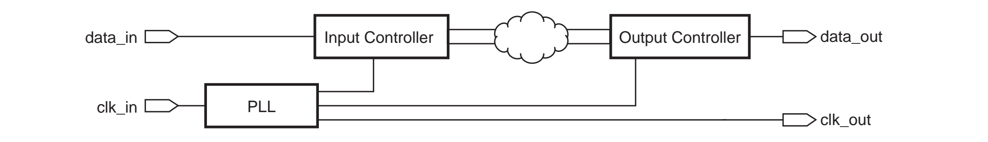
The **Maximum Clock Arrival Skew** assignment specifies the maximum clock skew between a set of registers. The QSF variable for the **Maximum Clock Arrival Skew** assignment is MAX_CLOCK_ARRIVAL_SKEW. In the Quartus II Classic Timing Analyzer, this assignment is specified between a clock node name and a set of registers. **Maximum Clock Arrival Skew** is not supported in the Quartus II TimeQuest Timing Analyzer.

Maximum Data Arrival Skew

The **Maximum Data Arrival Skew** assignment specifies the maximum data arrival skew between a set of registers, pins, or both. The QSF variable for the **Maximum Data Arrival Skew** assignment is MAX_DATA_ARRIVAL_SKEW. In this case, the data arrival delay represents the t_{CO} from the clock to the given register, pin, or both. This assignment is specified between a clock node and a set of registers, pins, or both.

The Quartus II TimeQuest Timing Analyzer does not support a constraint to specify maximum data arrival skew, but you can specify setup and hold times relative to a clock port to constrain an interface like this. Figure 9–31 shows a simplified source-synchronous interface used in the following example.

Figure 9–31. Source-Synchronous Interface Diagram

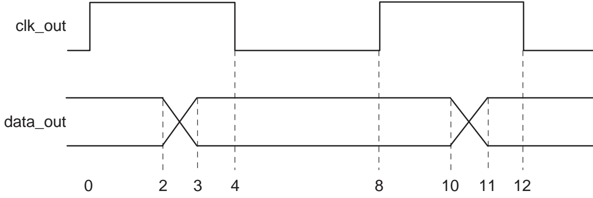


Constraining Skew on an Output Bus

This example constrains the interface so that all bits of the `data_out` bus go off-chip between 2 and 3 ns after the `clk_out` signal. Assume that `clock_in` and `clock_out` have a period of 8 ns.

The following equations and example show how to create timing requirements that satisfy the timing relationships shown in Figure 9–32.

Figure 9–32. Source-Synchronous Timing Diagram



Equation 9–11 shows how to compute the value for the **set_output_delay** `-min` command that creates the 2 ns hold requirement on the destination. For hold requirement calculations in which source and destination clocks are the same, *<latch>* – *<launch>* = 0.

Equation 9–11.

$$\text{latch} - \text{launch} = 0\text{ns}$$
$$\text{output delay} = \text{latch} - \text{launch} - 2\text{ns}$$
$$\text{output delay} = -2\text{ns}$$

Equation 9–12 shows how to compute the value for the **set_output_delay** command that creates the 3 ns setup requirement on the destination. For setup requirement calculations in which source and destination clocks are the same, *<latch>* – *<launch>* = clock period.

Equation 9–12.

$$\text{latch} - \text{launch} = 8\text{ns}$$
$$\text{output delay} = \text{latch} - \text{launch} - 3\text{ns}$$
$$\text{output delay} = 5\text{ns}$$

Refer to "I/O Constraints" on page 9–31 for an explanation of the above equations.

Example 9–17 shows the three constraints together.

Example 9–17. Constraining a DDR Interface

```
set period 8.000
create_clock -period $period \
             -name clock_in \
             clock_in
derive_pll_clocks
set_output_delay -add_delay \
    -clock ddr_pll_1_inst|altpll_component|pll|CLK[0] \
    -reference_pin clk_out \
    -min -2.000 \
    [get_ports data_out*]
set_output_delay -add_delay \
    -clock ddr_pll_1_inst|altpll_component|pll|CLK[0] \
    -reference_pin clk_out \
    -max [expr $period - 3.000] \
    [get_ports data_out*]
```

Conversion Utility

The Quartus II TimeQuest Timing Analyzer includes a conversion utility to help you convert Quartus II Classic timing assignments in a **.qsf** file to SDC constraints in an **.sdc** file. The utility can use information from your project report database (in the **\db** folder), if it exists, so you should compile your design before the conversion.

The conversion utility ignores all disabled QSF assignments. Disabled assignments show **No** in the **Enabled?** column of the Assignment Editor, and include the `-disable` option in the **.qsf** file.

Refer to "Conversion Utility" on page 9–3 to learn how to run the conversion utility.

Unsupported Global Assignments

The conversion utility checks whether any of the global timing assignments in Table 9–10 exist in your project. Any global assignments not supported by the conversion utility are ignored during the conversion. Refer to the indicated page for information about each assignment, and how to manually convert these global assignments to SDC commands.

Table 9–10. Global Timing Assignments

Assignment Name	QSF Variable	More Information
t_{SU} Requirement	TSU_REQUIREMENT	page 9–32
t_{H} Requirement	TH_REQUIREMENT	page 9–34
t_{CO} Requirement	TCO_REQUIREMENT	page 9–36
Minimum t_{CO} Requirement	MIN_TCO_REQUIREMENT	page 9–38
t_{PD} Requirement	TPD_REQUIREMENT	page 9–40
Minimum t_{PD} Requirement	MIN_TPD_REQUIREMENT	page 9–41

Recommended Global Assignments

When any unsupported assignments have been identified, the conversion utility checks the global assignments in Table 9–11 to ensure they match the specified values.

Table 9–11. Recommended Global Assignments

Quartus II Classic Assignment Name	QSF Variable	Value
Cut off clear and preset signal paths	CUT_OFF_CLEAR_AND_PRESET_PATHS	ON
Cut off feedback from I/O pins	CUT_OFF_IO_PIN_FEEDBACK	ON
Cut off read during write signal paths	CUT_OFF_READ_DURING_WRITE_PATHS	ON
Analyze latches as synchronous elements	ANALYZE_LATCHES_AS_SYNCHRONOUS_ELEMENTS	ON
Enable Clock Latency	ENABLE_CLOCK_LATENCY	ON
Display Entity Name	PROJECT_SHOW_ENTITY_NAME	ON

The following assignments are checked to ensure the functionality of the Quartus II Classic Timing Analyzer with the specified values corresponds to the behavior of the Quartus II TimeQuest Timing Analyzer.

- **Cut off clear and preset signal paths**— The Quartus II TimeQuest Timing Analyzer does not support this functionality. You should use Recovery and Removal analysis instead to analyze register control paths. The Quartus II Classic Timing Analyzer does not support this option.
- **Cut off feedback from I/O pins**—The Quartus II TimeQuest Timing Analyzer does not match the functionality of the Quartus II Classic Timing Analyzer when this assignment is OFF.
- **Cut off read during write signal paths**—The Quartus II TimeQuest Timing Analyzer does not match the functionality of the Quartus II Classic Timing Analyzer when this assignment is OFF.
- **Analyze latches as synchronous elements**—The Quartus II TimeQuest Timing Analyzer analyzes latches as synchronous elements by default and does not match the functionality of the Quartus II Classic Timing Analyzer when this assignment is OFF. Beginning with version 5.1 of the Quartus II software, the Quartus II Classic Timing Analyzer analyzes latches as synchronous elements by default.
- **Enable Clock Latency**—The Quartus II TimeQuest Timing Analyzer includes clock latency in its calculations. The Quartus II TimeQuest Timing Analyzer does not match the functionality of the Quartus II Classic Timing Analyzer when this assignment is OFF. Latency on a clock can be viewed as a simple delay on the clock path, and affects clock skew. This is in contrast to an offset, which alters the setup and hold relationship between two clocks. Refer to "Offset and Latency Example" on page 9–13 for an example of the different effects of offset and latency. When you turn on **Enable Clock Latency** in the Quartus II Classic Timing Analyzer, it affects the following aspects of timing analysis:
  - **Early Clock Latency** and **Late Clock Latency** assignments are honored
  - The compensation delay of a PLL is analyzed as latency
  - For clock settings where you do not specify an offset, the automatically computed offset is treated as latency.

- **Display Entity Name**—Any entity-specific assignments are ignored in the Quartus II TimeQuest Timing Analyzer because they do not include the entity name when this option is ON.

If your design meets timing requirements in the Quartus II Classic Timing Analyzer without all of the settings recommended in Table 9–11 on page 9–45, you should perform one of the following actions:

- Change the settings and re-constrain and re-verify as necessary.
  or
- Add or modify SDC constraints as appropriate because analysis in the Quartus II TimeQuest Timing Analyzer may be different after conversion.

Clock Conversion

Next, the conversion utility adds the **derive_pll_clocks** command to the **.sdc** file. This command creates generated clocks on all PLL outputs in your design each time the **.sdc** file is read. The command does not add a clock on the FPGA port that drives the PLL input.

The conversion utility includes the **derive_pll_clocks -use_tan_name** command in the **.sdc** file it creates. The `-use_tan_name` option overrides the default clock naming behavior (the PLL pin name) so the clock name is the same as the net name in the Quartus II Classic Timing Analyzer.

Including the `-use_tan_name` option ensures that the conversion utility converts clock-to-clock exceptions properly. If you remove the `-use_tan_name` option, you must also edit references to the clock names in other SDC commands so they match the PLL pin names.

If your design includes a global f_{MAX} assignment, the assignment is converted to a **derive_clocks** command. The behavior of a global f_{MAX} assignment is different from the behavior of clocks created with the **derive_clocks** command, and you should use the **report_clocks** command when you review conversion results to evaluate the clock settings. Refer to "Automatic Clock Detection" on page 9–16 for an explanation of the differences. As soon as you know the appropriate clock settings, you should use the **create_clock** or **create_generated_clock** command instead of the **derive_clocks** command.

The conversion utility adds a **post_message** command before the **derive_clocks** command to remind you that the clocks are derived automatically. The Quartus II TimeQuest Timing Analyzer displays the reminder the first time it reads the **.sdc** file. Remove or comment out the **post_message** command to prevent the message from displaying.

Next, the conversion utility identifies and converts clock settings in the **.qsf** file. If a project database exists, the utility also identifies and converts any additional clocks in the report file that are not in the **.qsf**, such as PLL base clocks.

If you change the PLL input frequency, you must modify the SDC constraint manually.

The conversion utility ignores clock offsets on generated clocks. Refer to "Clock Offset" on page 9–12 for information about how to use offset values in the Quartus II TimeQuest Timing Analyzer.

Instance Assignment Conversion

Next, the conversion utility converts the instance assignments shown in Table 9–12. Refer to the indicated page for information about each assignment.

Table 9–12. Instance Timing Assignments

Assignment Name	QSF Variable	More Information
Late Clock Latency	LATE_CLOCK_LATENCY	page 9–27
Early Clock Latency	EARLY_CLOCK_LATENCY	
Clock Setup Uncertainty	CLOCK_SETUP_UNCERTAINTY	page 9–28
Clock Hold Uncertainty	CLOCK_HOLD_UNCERTAINTY	
Multicycle *(1)*	MULTICYCLE	page 9–30
Source Multicycle *(2)*	SRC_MULTICYCLE	
Multicycle Hold *(3)*	HOLD_MULTICYCLE	
Source Multicycle Hold	SRC_HOLD_MULTICYCLE	
Clock Enable Multicycle	CLOCK_ENABLE_MULTICYCLE	page 9–30
Clock Enable Source Multicycle	CLOCK_ENABLE_SOURCE_MULTICYCLE	
Clock Enable Multicycle Hold	CLOCK_ENABLE_MULTICYCLE_HOLD	
Clock Enable Source Multicycle Hold	CLOCK_ENABLE_SOURCE_MULTICYCLE_HOLD	
Cut Timing Path	CUT	page 9–41
Input Maximum Delay	INPUT_MAX_DELAY	page 9–31
Input Minimum Delay	INPUT_MIN_DELAY	
Output Maximum Delay	OUTPUT_MAX_DELAY	
Output Minimum Delay	OUTPUT_MIN_DELAY	

Notes to Table 9–12:

(1) A multicycle assignment can also be known as a "destination multicycle setup" assignment.

(2) A source multicycle assignment can also be known as a "source multicycle setup" assignment.

(3) A multicycle hold can also be known as a "destination multicycle hold" assignment.

Depending on input and output delay assignments, you may receive a warning message when the **.sdc** file is read. The message warns that the **set_input_delay** command, **set_output_delay** command, or both are missing the `-max` option, `-min` option, or both. Refer to "Input and Output Delay" on page 9–31 for an explanation of why the warning occurs and how to avoid it.

Beginning in version 7.1 of the Quartus II software, the conversion utility automatically adds multicycle hold exceptions for each multicycle setup assignment. The value of each multicycle hold exception depends on the **Default hold multicycle** assignment value in your project. If the value is **One**, the conversion utility uses a value of 0 (zero) for each multicycle hold exception it adds. If the value is **Same as multicycle**, the conversion utility uses a value one less than the corresponding multicycle setup assignment value for each multicycle hold exception it adds. Refer to "Hold Multicycle" on page 9–20 for more information on hold multicycle differences between the Quartus II Classic and Quartus II TimeQuest Timing Analyzers.

Next, the conversion utility converts the instance assignments shown in Table 9–13. Refer to the indicated page for information about each assignment. If the t_{PD} and minimum t_{PD} assignment targets also have input or output delays that apply to them, the t_{PD} and minimum t_{PD} conversion values may be incorrect. This is described in more detail on the indicated pages for the appropriate assignments.

Table 9–13. Instance Timing Assignments

Assignment Name	QSF Variable	More Information
t_{PD} Requirement *(1)*	TPD_REQUIREMENT	page 9–40
Minimum t_{PD} Requirement *(1)*	MIN_TPD_REQUIREMENT	page 9–41
Setup Relationship	SETUP_RELATIONSHIP	page 9–27
Hold Relationship	HOLD_RELATIONSHIP	page 9–27

Note to Table 9–13:

(1) Refer to "t_{PD} and Minimum t_{PD} Requirement Conversion" on page 9–56 for more information about how the conversion utility converts single-point t_{PD} and minimum t_{PD} assignments.

The conversion utility converts Quartus II Classic I/O timing assignments to FPGA-centric SDC constraints. Table 9–14 includes Quartus II Classic assignments, the equivalent FPGA-centric SDC constraints, and recommended system-centric SDC constraints.

Table 9–14. Quartus II Classic and Quartus II TimeQuest Equivalent Constraints

Quartus II Classic	FPGA-Centric SDC	System-Centric SDC	More Information
t_{SU} Requirement *(1)*	set_max_delay	set_input_delay -max	page 9–32
t_H Requirement *(1)*	set_min_delay	set_input_delay -min	page 9–34
t_{CO} Requirement *(2)*	set_max_delay	set_output_delay -max	page 9–36
Minimum t_{CO} Requirement *(2)*	set_min_delay	set_output_delay -min	page 9–38

Notes to Table 9–14:

(1) Refer to "t_{PD} and Minimum t_{PD} Requirement Conversion" on page 9–56 for more information about how the conversion utility converts this type of assignment.

(2) Refer to "t_{CO} Requirement Conversion" on page 9–49 for more information about how the conversion utility converts this type of assignment.

The conversion utility can convert Quartus II Classic I/O timing assignments only to the FPGA-centric constraints without additional information about your design. Making system-centric constraints requires information about the external circuitry interfacing with the FPGA such as external clocks, clock latency, board delay, and clocking exceptions. You cannot convert Quartus II Classic timing assignments to system-centric constraints without that information. If you use the conversion utility, you can modify the SDC constraints to change the FPGA-centric constraints to system-centric constraints as appropriate.

PLL Phase Shift Conversion

The conversion utility does not account for PLL phase shifts when it converts values of the following FPGA-centric I/O timing assignments:

- **t_{SU} Requirement**
- **t_H Requirement**

- t_{CO} **Requirement**
- **Minimum** t_{CO} **Requirement**

If any of your paths go through PLLs with a phase shift, you must correct the converted values for those paths according to the following formula:

Equation 9–13.

$$\textit{<correct value>} = \textit{<converted value>} - \frac{(\textit{<pll output period>} \times \textit{<phase shift>})}{360}$$

Example 9–18 shows the incorrect conversion result for a t_{CO} assignment and how to correct it. For the example, assume the PLL output frequency is 200 MHz (period is 5 ns), the phase shift is 90 degrees, the t_{CO} **Requirement** value is 1 ns, and it is made to `data[0]`. The **.qsf** file contains the following assignment:

Example 9–18. Assignment

```
set_instance_assignment -name TCO_REQUIREMENT -to data[0] 1.0
```

The conversion utility generates the SDC command shown in Example 9–19.

Example 9–19. SDC Command

```
set_max_delay -from [get_registers *] -to [get_ports data[0]] 1.0
```

To correct the value, use the formula and values above, as shown in the following example:

$$1.0 - \frac{(5 \times 90)}{360} = -0.25$$

Then, change the value so the SDC command looks like Example 9–20.

Example 9–20. SDC Command with Correct Values

```
set_max_delay -from [get_registers *] -to [get_ports data[0]] -0.25
```

t_{CO} Requirement Conversion

The conversion utility uses a special process to convert t_{CO} **Requirement** and **Minimum** t_{CO} **Requirement** assignments. In addition to the **set_max_delay** or **set_min_delay** commands, the conversion utility adds a **set_output_delay** constraint relative to a virtual clock named N/C (Not a Clock). It also creates the virtual clock named N/C with a period of 10 ns. Adding the virtual clock allows you to report timing on the output paths. Without the virtual clock N/C, the clock used for reporting would be blank. Example 9–21 shows how the conversion utility converts a t_{CO} **Requirement** assignment of 5.0 ns to `data[0]`.

Example 9–21. Converting a t_{CO} Requirement Assignment of 5.0 ns to data[0]

```
set_max_delay -from [get_registers *] -to [get_ports data[0]]
set_output_delay -clock "N/C" 0 [get_ports data[0]]
```

Entity-Specific Assignments

Next, the conversion utility converts the entity-specific assignments listed in Table 9–15 that exist in the **Timing Analyzer Settings** report panel. This usually occurs if you have any timing assignments in your Verilog HDL or VHDL source, which can include MegaCore function files. These entity-specific assignments cannot be automatically converted unless your project is compiled and a **\db** directory exists.

Table 9–15. Entity-Specific Timing Assignments

Quartus II Classic	QSF Variable	More Information
Multicycle	MULTICYCLE	page 9–30
Source Multicycle	SRC_MULTICYCLE	
Multicycle Hold	HOLD_MULTICYCLE	
Source Multicycle Hold	SRC_HOLD_MULTICYCLE	
Setup Relationship	SETUP_RELATIONSHIP	page 9–27
Hold Relationship	HOLD_RELATIONSHIP	page 9–27
Cut Timing Path	CUT	page 9–41

You must manually convert all other entity-specific timing assignments.

Paths Between Unrelated Clock Domains

Beginning in version 7.1 of the Quartus II software, the conversion utility can create exceptions that cut paths between unrelated clock domains, which matches the default behavior of the Quartus II Classic Timing Analyzer. When **Cut paths between unrelated clock domains** is on, the conversion utility creates clock groups with the **set_clock_groups** command and uses the `-exclusive` option to cut paths between the clock groups.

Unsupported Instance Assignments

Finally, the conversion utility checks for the unsupported instance assignments listed in Table 9–16 and warns you if any exist. Refer to the indicated page for information about each assignment.

You can manually convert some of the assignments to SDC constraints.

Table 9–16. Instance Timing Assignments

Assignment Name	QSF Variable	More Information
Inverted Clock	INVERTED_CLOCK	page 9–28
Maximum Clock Arrival Skew	MAX_CLOCK_ARRIVAL_SKEW	page 9–42
Maximum Data Arrival Skew	MAX_DATA_ARRIVAL_SKEW	page 9–42
Maximum Delay	MAX_DELAY	page 9–41
Minimum Delay	MIN_DELAY	page 9–42
Virtual Clock Reference	VIRTUAL_CLOCK_REFERENCE	page 9–29

Reviewing Conversion Results

You must review the messages that are generated during the conversion process, and review the **.sdc** file for correctness and completeness. Warning and critical warning messages identify significant differences between the Quartus II Classic Timing Analyzer and Quartus II TimeQuest Timing Analyzer behaviors. In some cases, warning messages indicate that the conversion utility ignored assignments because it could not determine the intended functionality of your design. You must add to or modify the SDC constraints as necessary based on your knowledge of the design.

The conversion utility creates an **.sdc** file with the same name as your current revision, <*revision*>**.sdc**, and it overwrites any existing <*revision*>**.sdc** file. If you use the conversion utility to create an **.sdc** file, you should make additions or corrections in a separate **.sdc** file, or a copy of the **.sdc** file created by the conversion utility. That way, you can re-run the conversion utility later without overwriting your additions and changes. If you have constraints in multiple **.sdc** files, refer to“Constraint File Priority” on page 9–8 to learn how to add constraints to your project.

Warning Messages

The conversion utility may generate any of the following warning messages. Refer to the information provided for each warning message to learn what to do in that instance.

Ignored QSF Variable <*assignment*>

The conversion utility ignored the specified assignment. Determine whether an equivalent constraint is necessary and manually add one if appropriate. Refer to “Timing Assignment Conversion” on page 9–26 for information about conversions for all QSF timing assignments.

Global <*name*> = <*value*>

The conversion utility ignored the global assignment <*name*>. Manually add an equivalent constraint if appropriate. Refer to “Unsupported Global Assignments” on page 9–44 for information about conversions for these assignments.

QSF: Expected <*name*> to be set to <*expected value*> but it is set to <*actual value*>

The behavior of the Quartus II TimeQuest Timing Analyzer is closest to the Quartus II Classic Timing Analyzer when the value for the specified assignment is the expected value. Because the actual assignment value is not the expected value in your project, the Quartus II TimeQuest Timing Analyzer analysis may be different from the Quartus II Classic Timing Analyzer analysis. Refer to “Recommended Global Assignments” on page 9–45 for an explanation about the indicated QSF variable names.

QSF: Found Global Fmax Requirement. Translation will be done using derive_clocks

Your design includes a global f_{MAX} requirement, and the requirement is converted to the **derive_clocks** command. Refer to “Default Required f_{MAX} Assignment” on page 9–29 for information about how to convert to an SDC constraint.

TAN Report Database not found. HDL based assignments will not be migrated

You did not analyze your design with the Quartus II Classic Timing Analyzer before running the conversion utility. As a result, the conversion utility did not convert any timing assignments in your HDL source code to SDC constraints. If you have timing assignments in your HDL source code, you must find and convert them manually, or analyze your design with the Quartus II Classic Timing Analyzer and rerun the conversion utility.

Ignored Entity Assignment (Entity <entity>): *<variable>* = *<value>* -from *<from>* -to *<to>*

The conversion utility ignored the specified entity assignment because the utility cannot automatically convert the assignment. Table 9–15 on page 9–50 lists the entity-specific assignments the script can convert automatically.

Refer to "Timing Assignment Conversion" on page 9–26 for information about how to convert the entity assignment manually.

Ignoring OFFSET_FROM_BASE_CLOCK assignment for clock *<clock>*

In some cases, this assignment is used to work around a limitation in how the Quartus II Classic Timing Analyzer handles some forms of clock inversion. The conversion script ignores the assignment because it cannot determine whether the assignment is used as a workaround. Review your clock setting and add the assignment in your **.sdc** file if appropriate. Refer to "Clock Offset" on page 9–12 for more information about converting clock offset.

Clock *<clock>* has no FMAX_REQUIREMENT - No clock was generated

The conversion utility did not convert the clock named <*clock*> because it has no f_{MAX} requirement. You should add a clock constraint with an appropriate period to your **.sdc** file.

No Clock Settings defined in QSF file

If your **.qsf** file has no clock settings, ignore this message. You must add clock constraints in your **.sdc** file manually.

Clocks

Ensure that the conversion utility converted all clock assignments correctly. Run **report_clocks**, or double-click **Report Clocks** in the Tasks pane in the Quartus II TimeQuest Timing Analyzer GUI. Make sure that the right number of clocks is reported. If any clock constraints are missing, you must add them manually with the appropriate SDC commands (**create_clock** or **create_generated_clock**). Confirm that each option for each clock is correct.

The Quartus II TimeQuest Timing Analyzer can create more clocks, such as:

- `derive_clocks` selecting ripple clocks
- `derive_pll_clocks`, adding
  - Extra clocks for PLL switchover
  - Extra clocks for LVDS pulse-generated clocks (`~load_reg`)

Clock Transfers

After you confirm that all clock assignments are correct, run **report_clock_transfers**, or double-click **Report Clock Transfers** in the Tasks pane in the Quartus II TimeQuest Timing Analyzer GUI. The command generates a summary table with the number of paths between each clock domain. If the number of cross-clock domain paths seems high, remember that all clock domains are related in the Quartus II TimeQuest Timing Analyzer. You must cut unrelated clock domains. Refer to "Related and Unrelated Clocks" on page 9–11 for information about how to cut unrelated clock domains.

Path Details

If you have unexpected differences between the Quartus II Classic and Quartus II TimeQuest Timing Analyzers on some paths, follow these steps to identify the cause of the difference.

1. List the path in the Quartus II Classic Timing Analyzer.
2. Report timing on the path in the Quartus II TimeQuest Timing Analyzer.
3. Compare slack values.
4. Compare source and destination clocks.
5. Compare the launch/latch times in the Quartus II TimeQuest Timing Analyzer to the setup/hold relationship in the Quartus II Classic Timing Analyzer. The times are absolute in the Quartus II TimeQuest Timing Analyzer and relative in the Quartus II Classic Timing Analyzer.
6. Compare clock latency values.

Unconstrained Paths

Next, run `report_ucp`, or double-click **Report Unconstrained Paths** in the Tasks pane in the Quartus II TimeQuest Timing Analyzer GUI. This command generates a series of reports that detail any unconstrained paths in your design. If your design was completely constrained in the Quartus II Classic Timing Analyzer but there are unconstrained paths in the Quartus II TimeQuest Timing Analyzer, some assignments may not have been converted properly. Also, some of the assignments could be ambiguous. The Quartus II TimeQuest Timing Analyzer analyzes more paths than the Quartus II Classic Timing Analyzer does, so any unconstrained paths might be paths you could not constrain in the Quartus II Classic Timing Analyzer.

Bus Names

If your design includes Quartus II Classic Timing Analyzer timing assignments to buses, and the bus names do not include square brackets enclosing an asterisk, such as: `address[*]`, you should review the SDC constraints to ensure the conversion is correct. Refer to "Bus Name Format" on page 9–8 for more information.

Other

Review the notes listed in "Conversion Utility" on page 9–56.

Re-Running the Conversion Utility

You can force the conversion utility to run even if it can find an **.sdc** file according to the priority described in "Constraint File Priority" on page 9–8. Any method described in "Conversion Utility" on page 9–3 forces the conversion utility to run even if it can find an **.sdc** file.

Notes

This section describes notes for the Quartus II TimeQuest Timing Analyzer.

Output Pin Load Assignments

The Quartus II TimeQuest Timing Analyzer takes **Output Pin Load** values into account when it analyzes your design. If you change **Output Pin Load** assignments and do not recompile before you analyze timing, you must use the `-force_dat` option when you create the timing netlist. Type the following command at the Tcl console of the Quartus II TimeQuest Timing Analyzer:

```
create_timing_netlist -force_dat ↵
```

If you change **Output Pin Load** assignments and recompile before you analyze timing, do not use the `-force_dat` option when you create the timing netlist. You can create the timing netlist with the **create_timing_netlist** command, or with the **Create Timing Netlist** task in the Tasks pane.

Also note that the SDC `set_output_load` command is not supported, so you must make all output load assignments in the Quartus II Settings File (**.qsf**).

Constraint Target Types

In version 6.0 of the Quartus II software, the Quartus II TimeQuest Timing Analyzer did not support constraints between clocks and non-clocks. Beginning with version 6.1, the Quartus II TimeQuest Timing Analyzer supports mixed exception types; you can apply an exception of any clock/node combination.

DDR Constraints with the DDR Timing Wizard

The DDR Timing Wizard (DTW) creates an **.sdc** file that contains constraints for a DDR interface. You can use that **.sdc** file with the Quartus II TimeQuest Timing Analyzer to analyze only the DDR interface part of a design.

You should use the **.sdc** file created by DTW for constraining a DDR interface in the Quartus II TimeQuest Timing Analyzer. Additionally, your **.qsf** file should not contain timing assignments for the DDR interface if you plan to use the conversion utility to create an **.sdc** file. You should run the conversion utility before you use DTW, and you should choose not to apply assignments to the **.qsf** file.

However, if you used DTW and chose to apply assignments to a **.qsf** file, before you used the conversion utility, you should remove the DTW-generated QSF timing assignments and re-run the conversion utility. The conversion utility creates some incompatible SDC constraints from the DTW QSF assignments.

HardCopy Stratix Device Handoff

If you target the HardCopy device family, you should not use the Quartus II TimeQuest Timing Analyzer. The Quartus II TimeQuest Timing Analyzer is not supported for the HardCopy Stratix design process. The Quartus II TimeQuest Timing Analyzer supports HardCopy II series devices.

Unsupported SDC Features

Some SDC commands and features are not supported by the current version of the Quartus II TimeQuest Timing Analyzer. The following commands are included:

- The **get_designs** command, because the Quartus II software supports a single design, so this command is not necessary
- True latch analysis with time-borrowing feature; it can, however, convert latches to negative-edge-triggered registers
- The case analysis feature
- Loads, clock transitions, input transitions, and other features

Constraint Passing

The Quartus II software can read constraints generated by other EDA software, and write constraints to be read by other EDA software.

Other synthesis software can generate constraints that target the **.qsf** file. If you change timing constraints in synthesis software after creating an **.sdc** file for the Quartus II TimeQuest Timing Analyzer, you must update the SDC constraints. You can use the conversion utility, or update the **.sdc** file manually.

Optimization

Gate-level re-timing is not supported if you turn on the Quartus II TimeQuest Timing Analyzer as your default timing analyzer.

If you use physical synthesis with the Quartus II TimeQuest Timing Analyzer, the design may have lower performance.

Clock Network Delay Reporting

In the Quartus II software version 6.0, the Quartus II TimeQuest Timing Analyzer reports delay on the clock network as a single number, rather than node-to-node segments, as the Quartus II Classic Timing Analyzer does. Beginning with version 6.0 SP1, the TimeQuest Timing Analyzer reports delay on the clock network by node-to-node segments.

PowerPlay Power Analysis

You must perform the following steps to generate an **Early Power Estimator** output file when you use the Quartus II TimeQuest Timing Analyzer and your design targets one of the following device families:

- Cyclone
- Stratix
- HardCopy Stratix

To generate an **Early Power Estimator** output file for designs targeting those families, you must perform the following steps.

1. Turn off the Quartus II TimeQuest Timing Analyzer. Refer to "Set the Default Timing Analyzer" on page 9–4 to learn how to turn off the Quartus II TimeQuest Timing Analyzer.
2. Manually convert your Quartus II TimeQuest Timing Analyzer timing constraints in the **.sdc** file to Quartus II Classic Timing Analyzer timing assignments. You can use the Assignment Editor to enter your Quartus II Classic Timing Analyzer timing assignments in your **.qsf** file.
3. Perform Quartus II Classic timing analysis.
4. Generate an **Early Power Estimator** output file.
5. Turn on the Quartus II TimeQuest Timing Analyzer.

Project Management

If you use the **project_open** Tcl command in the Quartus II TimeQuest Timing Analyzer to open a project compiled with an earlier version of the Quartus II software, the Quartus II TimeQuest Timing Analyzer overwrites the compilation results (**\db** folder) without warning. Opening a project any other way results in a warning, and you can choose not to open the project.

Conversion Utility

This section describes the notes for the QSF assignment to SDC constraint conversion utility.

t_{PD} and Minimum t_{PD} Requirement Conversion

The conversion utility treats the targets of single-point t_{PD} and minimum t_{PD} assignments as device outputs. It does not correctly convert targets of single-point t_{PD} and minimum t_{PD} assignments that are device inputs. The following QSF assignment applies to an a device input named `d_in`:

```
set_intance_assignment -name TPD_REQUIREMENT -to d_in "3 ns"
```

The conversion utility creates the following SDC command, regardless of whether `d_in` is a device input or device output:

```
set_max_delay "3 ns" -from [get_ports *] -to [get_ports d_in]
```

You must update any incorrect SDC constraints manually.

Referenced Documents

This chapter references the following documents:

- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *SDC and TimeQuest Tcl API Reference Manual*

Document Revision History

Table 9–17. Document Revision History

Date and Version	Changes Made	Summary of Changes
March 2009 v9.0.0	This was chapter 8 in version 8.1.	Updated for the Quartus II 9.0 software release.
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	Updated for the Quartus II 8.1 software release.
May 2008 v8.0.0	■ Updated to Quartus II software version 8.0 and date. ■ Added hyperlinks to referenced Altera documentation throughout the chapter.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

10. Quartus II Classic Timing Analyzer

QII53004-9.0.0

Introduction

Static timing analysis is a method for analyzing, debugging, and validating the timing performance of a design. The classic timing analyzer analyzes the delay of every design path and analyzes all timing requirements to ensure correct circuit operation. Static timing analysis, used in conjunction with functional simulation, allows you to verify overall design operation.

For information about switching to the Quartus® II TimeQuest Timing Analyzer, refer to the *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

As part of the compilation flow, the Quartus II software automatically performs a static timing analysis so that you do not need to launch a separate timing analysis tool. The Quartus II Classic Timing Analyzer checks every path in the design against your timing constraints for timing violations and reports results in the Timing Analysis reports, giving you immediate access to the data.

This chapter assumes you have some Tcl expertise; Tcl commands are used throughout this chapter to describe alternative methods for making timing analysis assignments. Refer to "Timing Analysis Using the Quartus II GUI" on page 10–34 for GUI-equivalent timing constraints.

This chapter details the following aspects of timing analysis:

- "Timing Analysis Tool Setup" on page 10–2
- "Static Timing Analysis Overview" on page 10–2
- "Clock Settings" on page 10–7
- "Clock Types" on page 10–8
- "Clock Uncertainty" on page 10–10
- "Clock Latency" on page 10–11
- "Timing Exceptions" on page 10–13
- "I/O Analysis" on page 10–21
- "Asynchronous Paths" on page 10–24
- "Skew Management" on page 10–28
- "Generating Timing Analysis Reports with report_timing" on page 10–30
- "Other Timing Analyzer Features" on page 10–31
- "Timing Analysis Using the Quartus II GUI" on page 10–34
- "Scripting Support" on page 10–38
- "MAX+PLUS II Timing Analysis Methodology" on page 10–43

Timing Analysis Tool Setup

The Quartus II software version 6.0 and above supports two static timing analysis tools namely, the classic timing analyzer and the Quartus II TimeQuest Timing Analyzer. Use the **Timing Analysis** option under the Settings menu to set the Timing Analyzer that is used in the compilation process.

Arria® GX is not supported by the Quartus II Classic Timing Analyzer. To perform a static timing analysis for Arria GX, the Quartus II TimeQuest Timing Analyzer must be enabled.

The following steps set the classic timing analyzer as the default timing analysis tool in the Quartus II software.

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click the "+" icon to expand **Timing Analysis Settings** and select the **Use Classic Timing Analyzer during compilation** radio button.

Refer to the *Quartus II TimeQuest Timing Analyzer* chapter of the *Quartus II Handbook* for more information about the Quartus II TimeQuest Timing Analyzer.

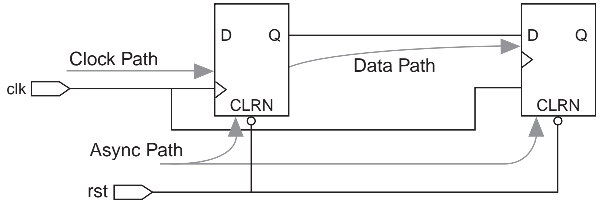
Static Timing Analysis Overview

This section provides information about static timing analysis concepts used throughout this chapter and used by the Quartus II Classic Timing Analyzer. A complete understanding of the concepts presented in this section allows you to take advantage of the powerful static timing analysis features available in the Quartus II software.

Various paths exist within any given design which connect design elements together, including the path from an output of a register to the input of another register. Timing paths play a significant role during a static timing analysis. Understanding the types of timing paths is important for timing closure and optimization. Some of the commonly analyzed paths are described in this section and are shown in Figure 10–1.

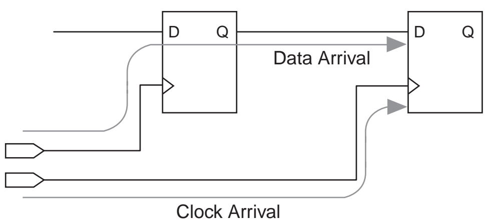
- *Clock paths*—Clock paths are the paths from device pins or internally generated clocks (nodes designated as a clock via a clock setting) to the clock ports of sequential elements such as registers.
- *Data paths*—Data paths are the paths from the data output port of a sequential element to the data input port of another sequential element.
- *Asynchronous paths*—Asynchronous paths are paths from a node to the asynchronous set or clear port of a sequential element.

Figure 10–1. Path Types



After the path types are identified, the classic timing analyzer computes data and clock arrival times for all valid register-to-register paths. Data arrival time is the delay from the source clock to the destination register. The Quartus II Classic Timing Analyzer calculates this delay by adding the clock path delay to the source register, the micro clock-to-out (t_{CO}) of the source register, and the data path delay from the source register to the destination register. Clock arrival time is the delay from the destination clock node to the destination register. Figure 10–2 shows a data arrival path and a clock arrival path.

Figure 10–2. Data Arrival and Clock Arrival

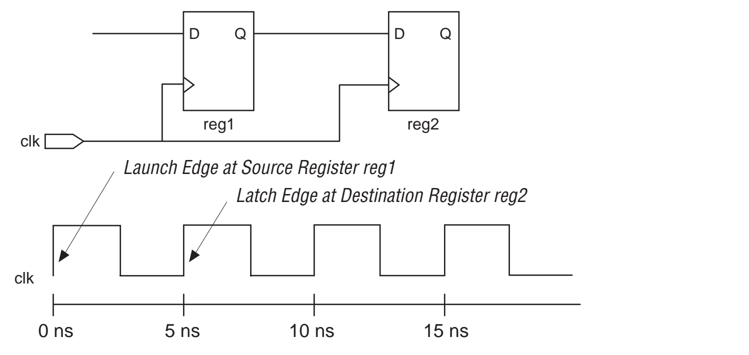


In addition to identifying various paths within a design, the Quartus II Classic Timing Analyzer analyzes clock characteristics to compute the worst-case requirement between any two registers in a single register-to-register path. You must use timing constraints to specify the characteristics of all clock signals in the design before this analysis occurs.

The active clock edge that sends data out of a sequential element, acting as a source for the data transfer, is the launch edge. The active clock edge that captures data at the data port of a sequential element, acting as a destination for the data transfer, is the latch edge.

Figure 10–3 shows a single-cycle system that uses consecutive clock edges to transmit and capture data, a register-to-register path, and the corresponding launch and latch edges timing diagram. In this example, the launch edge sends the data out of register `reg1` at 0 ns, and register `reg2` latch edge captures the data at 5 ns.

Figure 10–3. Launch Edge and Latch Edge



By analyzing specific paths relative to the launch and latch edges, the Quartus II Classic Timing Analyzer performs clock setup and clock hold checks, validating them against your timing assignments.

Clock Analysis

A comprehensive static timing analysis includes analysis of register-to-register, I/O, and asynchronous reset paths. Static Timing Analysis tools use data required times, data arrival times, and clock arrival times to verify circuit performance and detect possible timing violations. The Quartus II Classic Timing Analyzer determines the timing relationships that must be met for the design to correctly function, and checks arrival times against required times to verify timing.

Clock Setup Check

To determine if a design meets performance, the Quartus II Classic Timing Analyzer calculates clock timing, timing requirements, and timing exceptions to perform a clock setup check at each destination register based on the source and destination clocks and timing constraints, or exceptions that are applicable to those paths. A clock setup check ensures that data launched by a source register is latched correctly by the destination register. To perform a clock setup check, the Quartus II Classic Timing Analyzer determines the clock arrival time and data arrival time at the destination register by using the longest path for the data arrival time and the shortest path for the clock arrival time. The Quartus II Classic Timing Analyzer then checks that the difference is greater than or equal to the micro setup (t_{SU}) of the destination register as shown in Equation 10–1.

Equation 10–1.

$$\text{Clock Arrival Time} - \text{Data Arrival Time} \geq \text{micro } t_{SU}$$

By default, the Quartus II Classic Timing Analyzer assumes the launched and latched edges happen on consecutive active clock edges.

The results of clock setup checks are reported in terms of slack. Slack is the margin by which a timing requirement is met or not met. Positive slack indicates the margin by which a requirement is met, and negative slack indicates the margin by which a requirement is not met. The Quartus II Classic Timing Analyzer determines clock setup slack using Equation 10–2 through Equation 10–5.

Equation 10–2.

$$\text{Clock Setup Slack} = \text{Data Required Time} - \text{Data Arrival Time}$$

Equation 10–3.

$$\text{Data Required} = \text{Clock Arrival Time} - \text{micro } t_{SU} - \text{Setup Uncertainty}$$

Equation 10–4.

$$\text{Clock Arrival Time} = \text{Latch Edge} + \text{Shortest Clock Path to Destination Register}$$

Equation 10–5.

$$\begin{aligned}\text{Data Arrival Time} = {} & \text{Launch Edge} + \text{Longest Clock Path to Source Register} \\ & + \text{micro } t_{CO} + \text{Longest Data Delay}\end{aligned}$$

The Quartus II Classic Timing Analyzer reports clock setup slack using Equation 10–6 through Equation 10–9 (which are equivalent to Equation 10–2 through Equation 10–5).

Equation 10–6.

$$\text{Clock Setup Slack} = \text{Largest Register-to-Register Requirement} - \text{Longest Register-to-Register Delay}$$

Equation 10–7.

$$\begin{aligned}\text{Largest Register-to-Register Requirement} &= \text{Setup Relationship between Source and Destination} \\ &+ \text{largest clock skew} - \text{micro } t_{CO} \text{ of Source Register} - \text{micro } t_{SU} \text{ of Destination Register}\end{aligned}$$

Equation 10–8.

$$\begin{aligned}&\text{Setup Relationship between Source \& Destination Register} = \\ &\quad \text{Latch Edge} - \text{Launch Edge Setup Uncertainty}\end{aligned}$$

Equation 10–9.

$$\begin{aligned}\text{Largest Clock Skew} &= \text{Shortest Clock Path to Destination Register} \\ &- \text{Longest Clock Path to Source Register}\end{aligned}$$

Both sets of equations can be used to determine the slack value of any path.

Clock Hold Check

To prevent hold violations, the Quartus II Classic Timing Analyzer calculates clock timing, timing requirements, and timing exceptions to perform a clock hold check at each destination register. A clock hold check ensures data launched from the source register is not captured by an active clock edge earlier than the setup latch edge, and that the destination register does not capture data launched from the next active launch edge. To perform a clock hold check, the Quartus II Classic Timing Analyzer determines the clock arrival time and data arrival time at the destination register using the shortest path for the data arrival time and the longest path for the clock arrival time. The Quartus II Classic Timing Analyzer checks that the difference is greater than or equal to the micro hold time (t_H) of the destination register, as shown in Equation 10–10.

Equation 10–10.

$$\text{Data Arrival Time} - \text{Clock Arrival Time} \geq t_H$$

The Quartus II Classic Timing Analyzer determines clock hold slack using Equation 10–11 through Equation 10–14.

Equation 10–11.

$$\text{Clock Hold Slack} = \text{Data Arrival Time} - \text{Data Required Time}$$

Equation 10–12.

$$\text{Data Required Time} = \text{Clock Arrival Time} + \text{micro } t_H + \text{Hold Uncertainty}$$

Equation 10–13.

$$\text{Clock Arrival Time} = \text{Latch Edge} + \text{Longest Clock Path to Destination Register}$$

Equation 10–14.

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Shortest Clock Path to Source Register} + \text{micro } t_{CO} + \text{Shortest Data Delay}$$

The Quartus II Classic Timing Analyzer reports clock hold slack using Equation 10–15 through Equation 10–18.

Equation 10–15.

$$\text{Clock Hold Slack} = \text{Shortest Register-to-Register Delay} - \text{Smallest Register-to-Register Requirement}$$

Equation 10–16.

$$\text{Smallest Register-to-Register Requirement} = \text{Hold Relationship between Source \& Destination} + \text{Smallest Clock Skew} - \text{micro } t_{CO} \text{ of Source Register} + \text{micro } t_H \text{ of Destination Register}$$

Equation 10–17.

$$\text{Hold Relationship between Source and Destination Register} = \text{Latch} - \text{Launch} + \text{Hold Uncertainty}$$

Equation 10–18.

$$\text{Smallest Clock Skew} = \text{Longest Clock Path from Clock to Destination Register} - \text{Shortest Clock Path from Clock to Source Register}$$

These equations can be used to determine the slack value of any path.

Multicycle Paths

Multicycle paths are data paths that require more than one clock cycle to latch data at the destination register. For example, a register may be required to capture data on every second or third rising clock edge. Figure 10–4 shows an example of a multicycle path between a multiplier's input registers and output register where the destination latches data on every other clock edge.

Refer to "Multicycle" on page 10–13 for more information about multicycle exceptions.

 © March 2009 Altera Corporation

Figure 10–4. Example Diagram of a Multicycle Path

Figure 10–5 shows the default clock setup analysis launch and latch edges where multicycle assignment is equal to 1.

Figure 10–5. Default Clock Setup Analysis

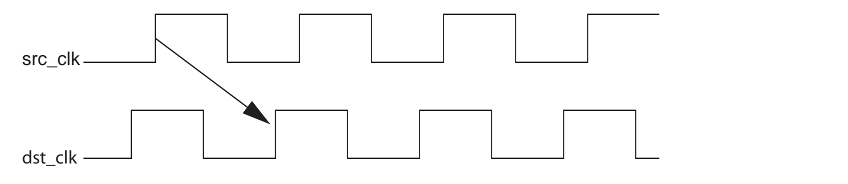
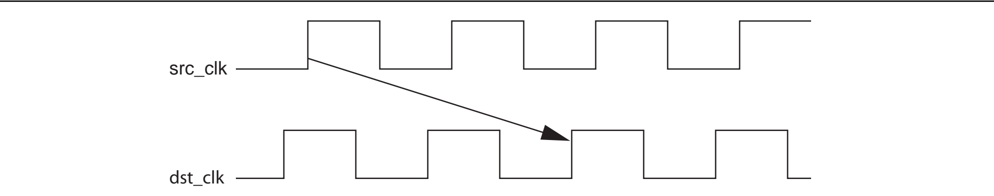


Figure 10–6 shows an analysis similar to Figure 10–5, but with a multicycle of 2.

Figure 10–6. Multicycle = 2 Clock Setup Analysis



Clock Settings

You can use individual and default clock settings to define the clocks in your design. You can base these clock settings on other clock settings already defined in your design.

To ensure the Quartus II Fitter achieves the desired performance requirements and the Quartus II Classic Timing Analyzer performs a thorough static timing analysis, you must specify all timing assignments prior to compiling the design.

Individual Clock Settings

Individual clock settings allow you to specify clock properties including performance requirements, offsets, duty cycles, and other properties for individual clock signals in your design.

You can define individual clock settings using the `create_base_clock` Tcl command. The following example defines an individual clock setting named `sys_clk` with a requirement of 100 MHz (10 ns), and assigns it to clock node `clk`.

```
create_base_clock -fmax 100MHz -target clk sys_clk
```

Default Clock Settings

You can assign a project-wide clock requirement to constrain all detected clocks in a design that do not have individual clock settings.

The `set_global_assignment -name FMAX_REQUIREMENT` Tcl command specifies a global default requirement assignment. The following example defines a 100 MHz default clock requirement:

```
set_global_assignment -name FMAX_REQUIREMENT "100.0 MHz"
```

For best placement and routing results, apply individual clock settings to all clocks in your design. All clocks adopting the default f_{MAX} are by default unrelated.

Clock Types

This section describes the types of clocks recognized by the Timing Analyzer:

- Base clocks
- Derived clocks
- Undefined clocks
- PLL clocks

Base Clocks

A base clock is independent of other clocks in a design. For example, a base clock is typically a clock signal driven directly by a device pin. A base clock is defined by individual clock settings, or automatically detected using the default clock setting.

You can use the `create_base_clock` Tcl command to define a base clock setting and assign the clock setting to a clock node. The following Tcl command creates a clock setting called `sys_clk` with a requirement of 5 ns (200 MHz) and applies the clock setting to clock node `main_clk`:

```
create_base_clock -fmax 5ns -target main_clk sys_clk
```

Derived Clocks

A derived clock is based on a previously defined base clock. For a derived clock, you can specify the phase shift, offset, multiplication and division factors, and duty cycle relative to the base clock.

You can use the `create_relative_clock` Tcl command to define and assign a derived clock setting. The following example creates a derived clock setting named `system_clockx2` that is twice as fast as the base clock `system_clock` applied to clock node `clkx2`.

```
create_relative_clock -base_clock system_clock -duty_cycle 50 \
-multiply 2 -target clkx2 system_clockx2
```

Undefined Clocks

The Quartus II Classic Timing Analyzer detects undeclared clocks in your design and displays a warning similar to the following:

```
Warning: Found pins functioning as undefined clocks and/or memory
enables
    Info: Assuming node "clk_src" is an undefined clock
    Info: Assuming node "clk_dst" is an undefined clock
```

The Quartus II Classic Timing Analyzer reports actual data delay for undefined clocks, but because no clock requirements exist for undefined clocks, the Quartus II Classic Timing Analyzer does not report slack for any register-to-register paths driven by an undefined clock.

PLL Clocks

Phase-locked loops (PLLs) are used for clock synthesis in Altera® devices. This device feature is configured and connected to your design using the altpll megafunction included with the Quartus II software. Using the MegaWizard™ Plug-In Manager, you can customize the input clock frequency, multiplication factors, division factors, and other parameters of the altpll megafunction.

For more information about using the PLL feature in your design, refer to the *ALTPLL Megafunction User Guide* or the handbook for the targeted device family.

For PLLs, the Quartus II Classic Timing Analyzer automatically creates derived clock settings based on the parameterization of the PLL and automatically creates a base clock setting for the input clock pin. For example, if the input clock frequency to a PLL is 100 MHz and the multiplication and division ratio is 5:2, the clock period of the PLL clock is set to 4.0 ns and is automatically calculated by the Quartus II Classic Timing Analyzer.

For the Stratix® and Cyclone® device families, you can override the PLL input clock frequency by applying a clock setting to the input clock pin of the PLL. For example, if the PLL input clock period is set to 10 ns (100 MHz) with a multiplication and division ratio of 5:2, but a clock setting of 20 ns (50 MHz) is applied to the input clock pin of the PLL, the setup relationship is 8.0 ns (125 MHz) and not 4.0 ns (250 MHz). The Quartus II Classic Timing Analyzer issues a message similar to the following:

```
Warning: ClockLock PLL "mypll_test:inst|altpll:altpll_component|_clk1"
input frequency requirement of 200.0 MHz overrides default required fmax
of 100.0 MHz -- Slack information will be reported
```

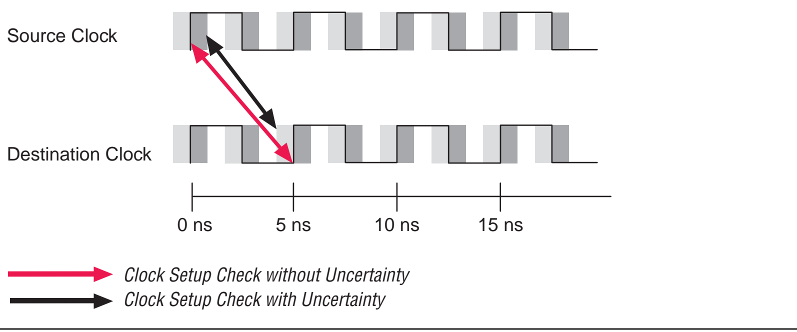
You cannot override the PLL output clock frequency with a clock setting in the Quartus II Classic Timing Analyzer.

Clock Uncertainty

You can use Clock Setup Uncertainty and Clock Hold Uncertainty assignments to model jitter, skew, or add a guard band associated with clock signals.

When a clock uncertainty assignment exists for a clock signal, the Timing Analyzer performs the most conservative setup and hold checks. For clock setup check, the setup uncertainty is subtracted from the data required time. Figure 10–7 shows an example of clock sources with a clock setup uncertainty applied.

Figure 10–7. Clock Setup Uncertainty

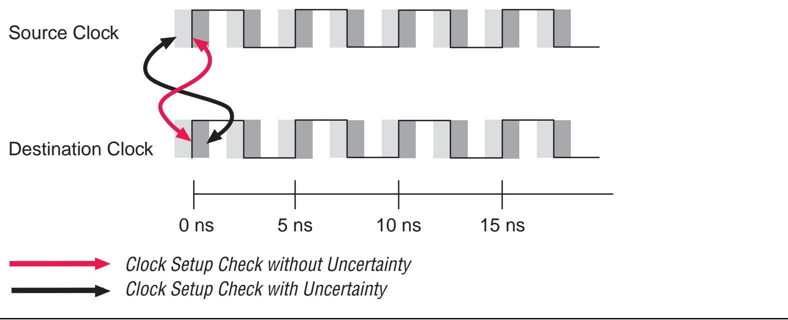


You can create clock uncertainty assignments using the Tcl command `set_clock_uncertainty`. The `set_clock_uncertainty` assignment used with the switch `-setup` specifies a clock setup uncertainty assignment. The following example creates a Clock Setup Uncertainty assignment with a value of 2 ns applied to clock signal `clk`:

```
set_clock_uncertainty -to clk -setup 2ns
```

For the clock hold check, the hold uncertainty is added to the data required time. Figure 10–8 shows an example of clock setup check with a clock setup uncertainty and clock hold uncertainty applied.

Figure 10–8. Clock Hold Uncertainty



You can use the `set_clock_uncertainty` Tcl command with the option `-hold` to specify a Clock Hold Uncertainty assignment. The following example creates a Clock Hold Uncertainty assignment with a value of 2 ns for clock signal `clk`.

```
set_clock_uncertainty -to clk -hold 2ns
```

You can also apply the clock uncertainty assignments between two clock sources. The following example creates a Clock Setup Uncertainty assignment for clock setup checks where `clk1` is the source clock and `clk2` is the destination clock:

```
set_clock_uncertainty -from clk1 -to clk2 -setup 2ns
```

Clock Latency

You can use clock latency assignments to model delays from the clock source. For example, you can use clock latency to model an external delay from an ideal clock source, such as an oscillator, to the clock pin or port of the device.

The Early Clock Latency assignment allows you to specify the shortest or earliest delay of the clock source. Conversely, the Late Clock Latency assignment allows you to specify the longest or latest delay of the clock source.

During setup analysis, the Quartus II Classic Timing Analyzer adds the Late Clock Latency assignment value to the source clock path delay and adds the Early Clock Latency assignment value to the destination clock path delay when determining clock skew for the path. During clock hold analysis, the Quartus II Classic Timing Analyzer adds the Early Clock Latency assignment value to the source clock path delay and adds the Late Clock Latency assignment value to the destination clock path delay when determining clock skew for the path.

The Early Clock Latency and Late Clock Latency assignments do not change the latch and launch edges defined by the clock setting and therefore does not change the setup or hold relationships between source and destination clocks. The clock latency assignments add only delay to the clock network and therefore only affects clock skew.

Figure 10–9 shows the clock edges used to calculate clock skew for a setup check when the Early Clock Latency and Late Clock Latency assignments are used.

Figure 10–9. Clock Setup Check Clock Skew

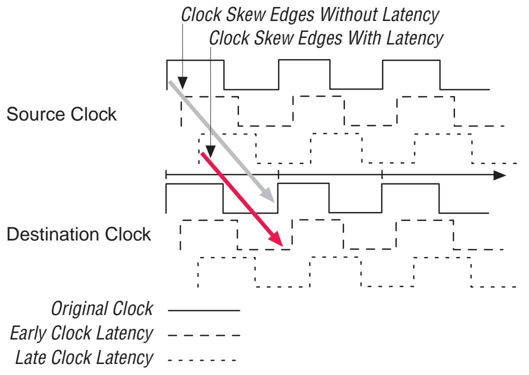
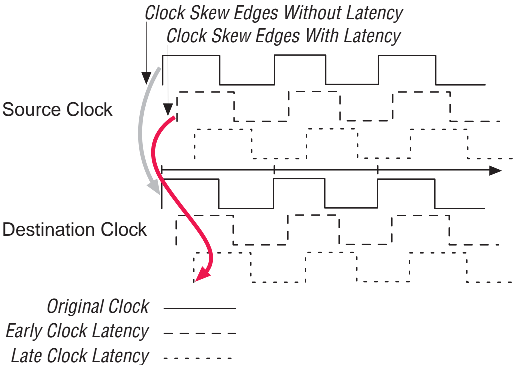


Figure 10–10 shows the clock edges used to calculate clock skew for a hold check when the Early Clock Latency and Late Clock Latency assignments are used.

Figure 10–10. Clock Hold Check Clock Skew



The Quartus II Classic Timing Analyzer ignores clock latency if the clock signal at the source and destination registers are the same.

You can use the `set_clock_latency` Tcl command with the switches `-early` or `-late` to specify an Early Clock Latency assignment or Late Clock Latency assignment, respectively. Example 10–1 specifies that the clock signal at `clk2` arrives as early as 1.8 ns and as late as 2.0 ns.

Example 10–1. Specifying Early or Late Clock Latency at clk2

```
set_clock_latency -early -to clk2 1.8ns
set_clock_latency -late -to clk2 2ns
```

The early clock latency default value is the same as the late clock latency delay, and the late clock latency default value is the same as the early clock latency delay, if only one is specified.

The **Enable Clock Latency** option must be set to **ON** for the Quartus II Classic Timing Analyzer to analyze clock latency. When this option is set to **ON**, the Quartus II Classic Timing Analyzer reports clock latency as part of the clock skew calculation for either the source or destination clock path depending upon the analysis performed. To set the **Enable Clock Latency** option to **ON**, you can use the following Tcl command:

```
set_global_assignment -name ENABLE_CLOCK_LATENCY ON
```

When the **Enable Clock Latency** option is enabled, the Quartus II Classic Timing Analyzer automatically calculates latencies for derived clocks instead of automatically calculating offsets; for example, PLL compensation delays. These clock path delays are accounted for as clock skew instead of part of the setup or hold relationship as done with offsets.

For more information about clock latency, refer to *AN 411: Understanding PLL Timing for Stratix II Devices*.

Timing Exceptions

Timing exceptions allow you to modify the default behavior of the Quartus II Classic Timing Analyzer. This section describes the following timing exceptions:

- Multicycle
- Setup relationship and hold relationship
- Maximum delay and minimum delay
- False paths

Not all timing exceptions presented in this chapter are applicable to the HardCopy® II devices. If you are designing for the HardCopy II device family, refer to the *Timing Constraints for HardCopy II Devices* chapter in the *HardCopy II Handbook*.

Multicycle

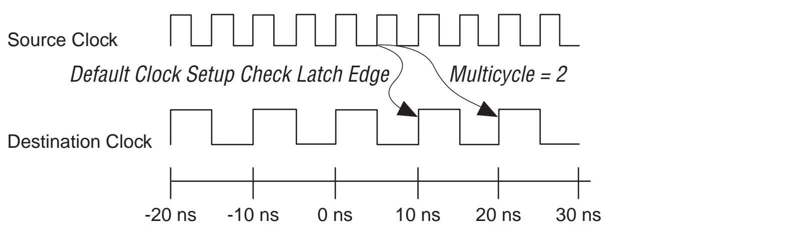
By default, the Quartus II Classic Timing Analyzer performs a single-cycle analysis for all valid register-to-register paths in the design. Multicycle assignments specify the number of clock periods before a source register launches the data or a destination register latches the data. Multicycle assignments adjust the latch or launch edges, which relaxes the required clock setup check or clock hold check between the source and destination register pairs. You can specify multicycles separately for setup and hold, and multicycles can be based on the source clock or destination clock. Apply Multicycle exception to time groups, clock nodes, or common clock enables.

Destination Multicycle Setup Exception

A destination multicycle setup, referred to as a Multicycle exception, specifies the minimum number of clock cycles required before a register should latch a value. A Multicycle exception changes the latch edge by relaxing the required setup relationship. Figure 10–11 shows a timing diagram for a multicycle path that exists in a design with related clocks, and with the latch edge label for a clock setup check.

By default, the Multicycle exception value is 1.

Figure 10–11. Multicycle Setup



You can apply Multicycle exception between any two registers or between any two clock domains. Use the Tcl command `set_multicycle_assignment`, and the switch `-setup` and `-end`. For example, to apply a Multicycle exception of 2 between all registers clocked by source clock `clk_src`, and all registers clocked by destination clock `clk_dst`, enter the following Tcl command:

```
set_multicycle_assignment -setup -end -from clk_src -to clk_dst 2
```

To apply a Multicycle exception of 2 between the source register `reg1` and the destination register `reg2`, enter the following Tcl command:

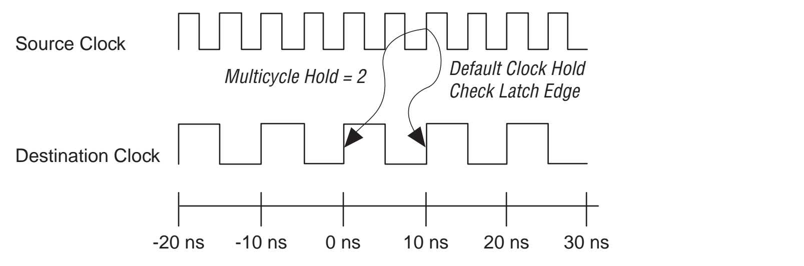
```
set_multicycle_assignment -setup -end -from reg1 -to reg2 2
```

Destination Multicycle Hold Exception

A destination multicycle hold, referred to as a Multicycle Hold exception, modifies the latch edge used for a clock hold check for the register-to-register path based on the destination clock. A Multicycle Hold exception changes the latch edge by relaxing the required hold relationship. Figure 10–12 shows a timing diagram labeling the latching edge for a clock setup check.

If no Multicycle Hold value is specified, the Multicycle Hold value defaults to the value of the multicycle exception.

Figure 10–12. Multicycle Hold



You can create Multicycle Hold exceptions with the Tcl command `set_multicycle_assingment` and the switch `-hold` and `-end`. The following example specifies a Multicycle Hold exception of 3 from register `reg1` to register `reg2`:

```
set_multicycle_assignment -hold -end -from reg1 -to reg2 3
```

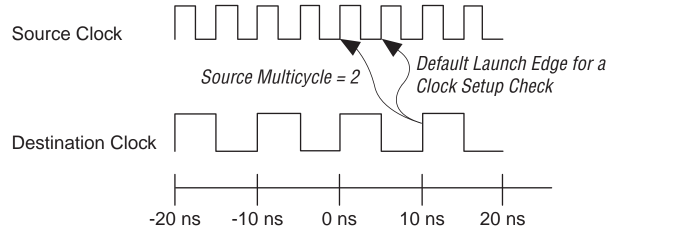
By default, the hold multicycle is set to equal that of the setup multicycle value along the same path. For example, if a setup multicycle of 2 has been applied to a register-to-register path without a separate hold multicycle, the hold multicycle value would be set to 2. The default hold multicycle value can also be changed to a value of 1. This forces all paths with a setup multicycle assignment to have a default hold multicycle of 1. To change the default hold multicycle value, in the **Settings** dialog box, click the **More Timing Settings** option.

If your design requires a hold multicycle value not equal to the setup multicycle or 1, you must explicitly apply a hold multicycle assignment to the path.

Source Multicycle Setup Exception

A source multicycle setup, referred to as Source Multicycle Setup exception, is used to extend the required delay by adjusting the source clock's launch edge rather than the destination clock's latch edge; for example, multicycle setup. Source Multicycle exceptions are useful when the source and destination registers are clocked by related clocks at different frequencies. Figure 10–13 shows an example of a Source Multicycle exception with the launch edge labeled for a clock setup check.

Figure 10–13. Source Multicycle



You can create Source Multicycle Setup exceptions with the Tcl command `set_multicycle_assignment` and the switches `-setup` and `-start`. The following example specifies a Source Multicycle exception of 3 from register `reg1` to register `reg2`:

```
set_multicycle_assignment -setup -start -from reg1 -to reg2 3
```

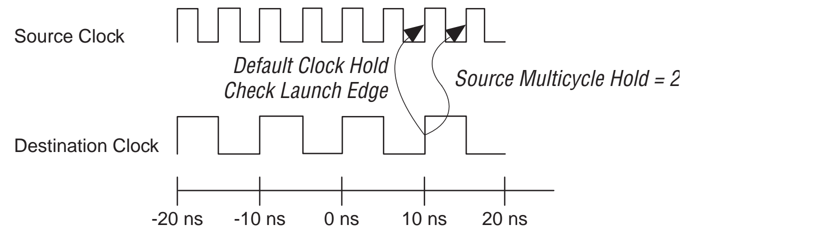
By default, the hold multicycle is set to equal that of the setup multicycle value along the same path. For example, if a setup multicycle of 2 has been applied to a register-to-register path without a separate hold multicycle, the hold multicycle value would be set to 2. The default hold multicycle value can also be changed to a value of 1. This forces all paths with a setup multicycle assignment to have a default hold multicycle of 1. To change the default hold multicycle value, in the **Settings** dialog box, click the **More Timing Settings** option.

If your design requires a hold multicycle value not equal to the setup multicycle or 1, you must explicitly apply a hold multicycle assignment to the path.

Source Multicycle Hold Exception

The Source Multicycle Hold exception modifies the latch edge used for a clock hold check for the register-to-register path based on the source clock. Source Multicycle Hold exceptions increase the required hold delay by adding source clock cycles. Figure 10–14 shows an example of a source multicycle hold with launch edge labeled for a clock hold check.

Figure 10–14. Source Multicycle Hold



You can create Source Multicycle Hold exceptions with the Tcl command `set_multicycle_assingment` and the switch `-setup` and `-start`. The following example specifies a Source Multicycle Hold exception of 3 from register `reg1` to register `reg2`:

```
set_multicycle_assignment -hold -start -from reg1 -to reg2 3
```

Default Hold Multicycle

The Quartus II Classic Timing Analyzer sets the hold multicycle value to equal the multicycle value when a multicycle exception has been entered without a corresponding hold multicycle. You can change the behavior with the `DEFAULT_HOLD_MULTICYCLE` assignment. The value of the assignment can either be "`ONE`" or "`SAME AS MULTICYCLE`".

The assignment has the following syntax:

```
set_global_assignment -name DEFAULT_HOLD_MULTICYCLE "<value>"
```

Clock Enable Multicycle

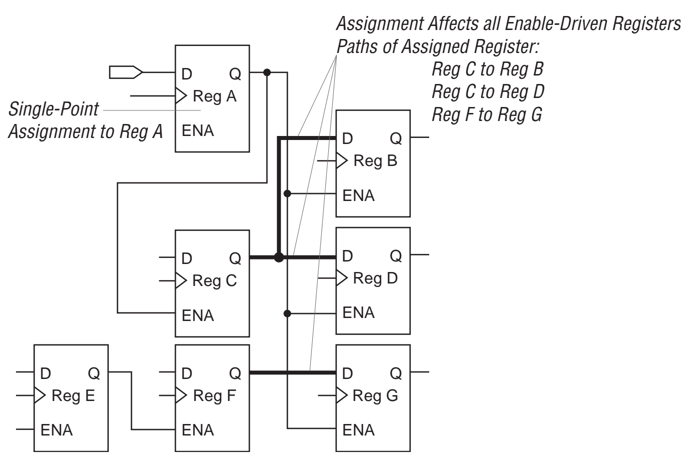
For all enable-driven registers, the setup relationship or hold relationship can be modified with the Clock Enable Multicycle, Clock Enable Multicycle Hold, Clock Enable Source Multicycle, or Clock Enable Multicycle Source Hold.

The **Clock Enable Multicycle** modifies the latching edge when a clock setup check is performed for all registers driven by the specified clock enables, and the **Clock Enable Multicycle Hold** modifies the latching edge when a clock hold check is performed for all registers driven by the specified clock enable. The **Clock Enable Source Multicycle** modifies the launching edge when a clock setup check is performed for all enabled driven registers, and the **Clock Enable Source Multicycle Hold** modifies the launching edge when a clock hold check is performed for all enabled driven registers.

Clock enable-based multicycle exceptions apply only to registers using dedicated clock enable circuitry. If the enable is synthesized into a logic cell; for example, due to signal prioritization, the multicycle does not apply.

The Clock Enable Multicycle, Clock Enable Multicycle Hold, Clock Enable Source Multicycle, and Clock Enable Multicycle Source Hold can be either a single-point or a point-to-point assignment. Figure 10–15 shows an example of a single-point assignment. In this example, register `Reg A` has the single-point assignment applied. This has the affect of modifying a register-to-register latching edge whose enable port is driven by register `Reg A`. All register-to-register paths with enables driven by the single-point assignment are affected, even those driven by different clock sources.

Figure 10–15. Single-Point Clock Enable Multicycle



Point-to-point assignments apply to all paths where the source registers' enable ports are driven by the source (`from`) node and the destination registers' enable ports are driven by the destination (`to`) node. Figure 10–16 shows an example of a point-to-point assignment made to different source and destination registers. In this example, register `Reg A` is specified as the source, and register `Reg B` is specified as the destination for the assignment. Only register-to-register paths that have their enables driven by the assigned point-to-point registers have their latching edges modified.

Figure 10–16. Different Source and Destination Point-to-Point Assignment Clock Enable Multicycle

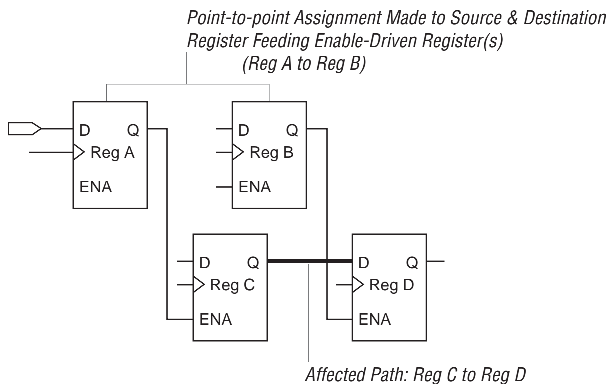
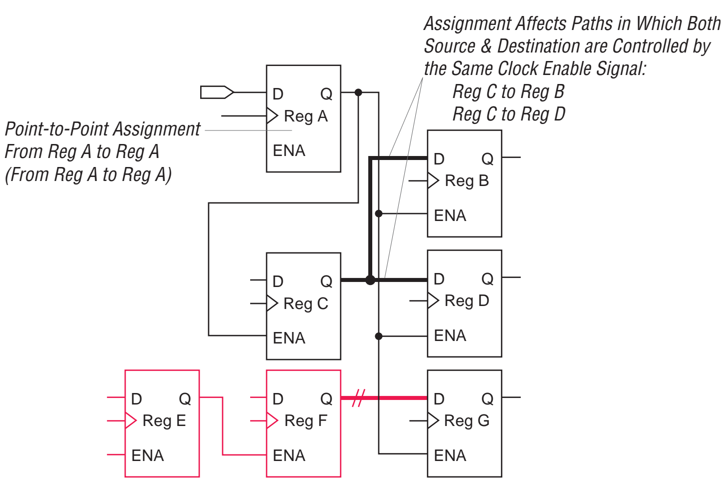


Figure 10–17 shows an example of a point-to-point assignment made to the same source and destination register. In this example, register `Reg A` has been specified as both the source and register for the assignment. Only register-to-register paths that have both the `source-enable` port and `destination-enable` port has the latching edge modified by the assigned point-to-point assignment.

Figure 10–17. Same Source and Destination Point-to-Point Assignment Clock Enable Multicycle



You can use the `set_instance_assignment -name CLOCK_ENABLE_MULTICYCLE` and `set_instance_assignment -name CLOCK_ENABLE_MULTICYCLE_HOLD` Tcl commands to specify either a Clock Enable Multicycle or a Clock Enable Multicycle Hold assignment, respectively. The following example specifies a single-point Clock Enable Multicycle assignment of 2 ns to `reg1`:

```
set_instance_assignment -name CLOCK_ENABLE_MULTICYCLE 2 -to reg1
```

The following example specifies a point-to-point Clock Enable Multicycle Hold assignment of 2 from register `reg1` to register `reg2`:

```
set_instance_assignment -name CLOCK_ENABLE_MULTICYCLE_HOLD 2 \
-from reg1 -to reg2
```

You can use the `set_instance_assignment -name CLOCK_ENABLE_SOURCE_MULTICYCLE` and `set_instance_assignment -name CLOCK_ENABLE_MULTICYCLE_SOURCE_HOLD` Tcl commands to specify either a Clock Enable Multicycle or Clock Enable Multicycle Hold assignment, respectively. The following example specifies a single-point Clock Enable Multicycle assignment of 2 ns to `reg1`:

```
set_instance_assignment -name CLOCK_ENABLE_SOURCE_MULTICYCLE 2 \
-to reg1
```

The following example specifies a point-to-point Clock Enable Multicycle Hold assignment of 2 from register `reg1` to register `reg2`:

```
set_instance_assignment -name \
CLOCK_ENABLE_SOURCE_MULTICYCLE_HOLD 2 -from reg1 -to reg2
```

Setup Relationship and Hold Relationship

By default, the Quartus II Classic Timing Analyzer determines all setup and hold relationships based on clock settings. The Setup Relationship and Hold Relationship exceptions allow you to override any default setup or hold relationships. Example 10–2 shows the path details of a register-to-register path that has a 10 ns clock setting applied to the clock signal driving the 2 registers.

Example 10–2. Default Setup Relationship with 10 ns Clock Setting

```
Info: Slack time is 9.405 ns for clock "data_clk" between source register "reg9" and
destination register "reg10"
    Info: Fmax is restricted to 500.0 MHz due to tcl and tch limits
    Info: + Largest register to register requirement is 9.816 ns
        Info: + Setup relationship between source and destination is 10.000 ns
            Info: + Latch edge is 10.000 ns
            Info: - Launch edge is 0.000 ns
        Info: + Largest clock skew is 0.000 ns
        Info: - Micro clock to output delay of source is 0.094 ns
        Info: - Micro setup delay of destination is 0.090 ns
    Info: - Longest register to register delay is 0.411 ns
```

In Example 10–3, a 15 ns Setup Relationship exception is applied to the register-to-register path, overriding the default 10 ns setup relationship.

Example 10–3. Setup Relationship Assignment of 15 ns

```
Info: Slack time is 14.405 ns for clock "data_clk" between source register "reg9" and
destination register "reg10"
   Info: Fmax is restricted to 500.0 MHz due to tcl and tch limits
   Info: + Largest register to register requirement is 14.816 ns
      Info: + Setup relationship between source and destination is 15.000 ns
      Info: Setup Relationship assignment value is 15.000 ns between source "reg9" and
      destination "reg10"
      Info: + Largest clock skew is 0.000 ns
      Info: Total interconnect delay = 1.583 ns ( 51.31 % )
   Info: - Micro clock to output delay of source is 0.094 ns
   Info: - Micro setup delay of destination is 0.090 ns
Info: - Longest register to register delay is 0.411 ns
```

You can create a Setup Relationship exception with the Tcl command `set_instance_assignment -name SETUP_RELATIONSHIP`. The following example specifies a Setup Relationship exception of 5 ns from register `reg1` to register `reg2`:

```
set_instance_assignment -name SETUP_RELATIONSHIP 5ns -from reg1 \
-to reg2
```

You can use Hold Relationship exception to override the default hold relationship of any register-to-register paths.

You can use the `set_instance_assignment -name HOLD_RELATIONSHIP` Tcl command to specify a hold relationship assignment. The following example specifies a Hold Relationship exception of 1 ns from register `reg1` to register `reg2`:

```
set_instance_assignment -name HOLD_RELATIONSHIP 1ns -from reg1 \
-to reg2
```

Maximum Delay and Minimum Delay

You can use Maximum Delay and Minimum Delay assignments to specify delay requirements for pin-to-register, register-to-register, and register-to-pin paths. The Maximum Delay assignment overrides any setup relationship for any path. The Minimum Delay assignment overrides any hold relationship for any path.

The Quartus II Classic Timing Analyzer ignores the effects of clock skew when checking a design against Maximum Delay and Minimum Delay assignments.

You can use the `set_instance_assignment -name MAX_DELAY` and `set_instance_assignment -name –MIN_DELAY` Tcl commands to specify a Maximum Delay assignment or a Minimum Delay assignment, respectively. The following example specifies a maximum delay of 2 ns between source register `reg1` and destination register `reg2`:

```
set_instance_assignment -name MAX_DELAY 2ns -from reg1 -to reg2
```

The following example specifies a minimum delay of 1 ns between input pin `data_in` to destination register `dst_reg`:

```
set_instance_assignment -name MIN_DELAY 1ns -from data_in -to dst_reg
```

False Paths

A false path is any path that is not relevant to a circuit's operation, such as test logic. There are several global assignments to cut different classes of paths, such as unrelated clock domains and paths through bidirectional pins, but you can also cut an individual timing path to a specific false path.

The Timing Analyzer provides the following three global options that allow you to remove false paths from your design:

- Cut off feedback from I/O pins
- Cut off `read-during-write` signal paths
- Cut paths between unrelated clock domains

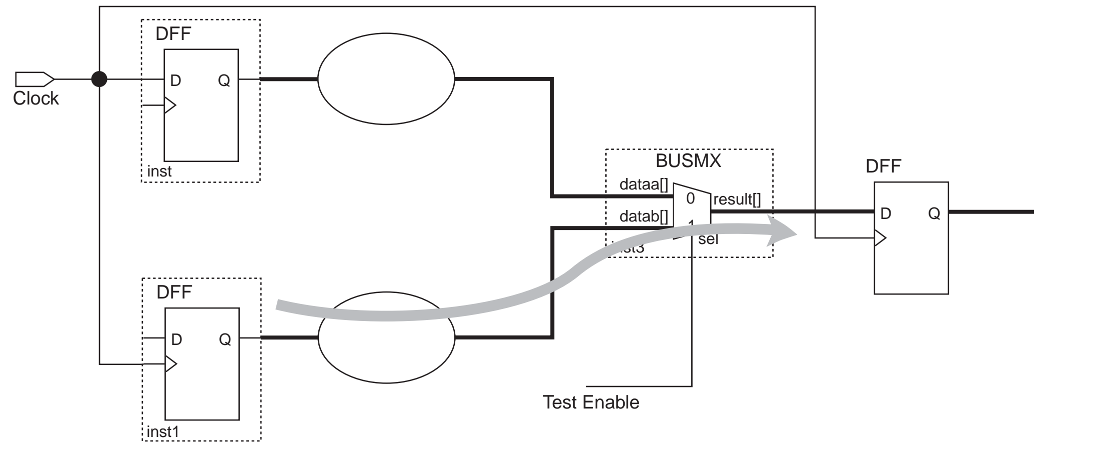
You can use the `set_global_assignment -name CUT_OFF_IO_PIN_FEEDBACK ON` Tcl command to cut the feedback path when a bidirectional I/O pin is connected directly or indirectly to both the input and output of a latch.

You can use the `set_global_assignment -name CUT_OFF_READ_DURING_WRITE_PATHS ON` Tcl command to cut the path from the `write-enable` register through memory element to a destination register.

You can use the `set_global_assignment -name CUT_OFF_PATHS_BETWEEN_CLOCK_DOMAINS ON` Tcl command to cut paths between register-to-register where the source and destination clocks are different.

You can use the `set_timing_cut_assignment` Tcl command to cut specific timing paths. In Figure 10–18, the path from `inst1` through the multiplexer to `inst2` is used only for design testing. This false path is not required under normal operation and does not need to be analyzed during static timing analysis. Figure 10–18 shows an example of a false path.

Figure 10–18. False Path Signal



To cut the timing path from source register `inst1` to destination register `inst2`, enter the following Tcl command:

```
set_timing_cut_assignment -from inst1 -to inst2
```

You can also use the `set_timing_cut_assignment` Tcl command as a single point assignment. When you use the single point assignment, all fanout of the node is cut. For example, the following Tcl command cuts all timing paths originating for node `src_reg`:

```
set_timing_cut_assignment -to src_reg
```

I/O Analysis

The I/O analysis performed by the Quartus II Classic Timing Analyzer ensures your Altera FPGA design meets all timing specifications for interfacing with external devices. This section describes assignments relevant to I/O analysis and other I/O analysis features and options available with the Quartus II Classic Timing Analyzer.

External Input Delay and Output Delay Assignments

External input and output delays represent delays from or to external devices or boards traces. You can make **Input Delay** and **Output Delay** assignments to ensure the Quartus II Classic Timing Analyzer can perform a full system analysis. By providing **Input Delays** and **Output Delays**, the Quartus II Classic Timing Analyzer is able to perform clock setup and clock hold checks for these paths. This also allows other timing assignments, such as multicycle or clock uncertainty, to be applied to input and output paths.

Do not combine individual or global t_{SU}, t_H, t_{PD}, t_{CO}, minimum t_{CO}, or minimum t_{PD} assignments with **Input Delay** or **Output Delay** assignments.

Input Delay Assignment

External input delays are specified with either Input Maximum Delay or Input Minimum Delay assignments. Make Input Maximum Delay assignments to specify the maximum delay of a signal from an external register to a specified input or bidirectional pin on the FPGA relative to a specified clock source. Make Input Minimum Delay assignments to specify the minimum delay of a signal from an external register to a specified input or bidirectional pin on the FPGA relative to a specified clock source.

When performing a clock setup check, the Quartus II Classic Timing Analyzer adds the Input Maximum Delay assignment value to the data arrival time (or subtracts the assignment value from the point-to-point requirement).

When performing a clock hold check, the Quartus II Classic Timing Analyzer adds the Input Minimum Delay assignment value to the data arrival time (or subtracts the assignment value from the point-to-point requirement).

The value of the input delay assignment usually represents the sum of the t_{CO} of the external device, the actual board delay to the input pin of the Altera device, and the board clock skew.

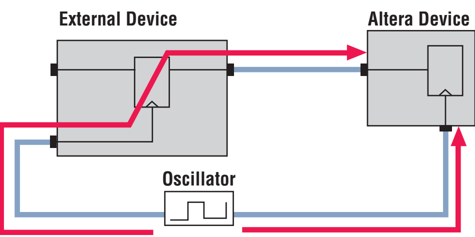
The **Input Minimum Delay** defaults to the **Input Maximum Delay** and the **Input Maximum Delay** defaults to the Input **Minimum Delay** if only one is specified.

For example, the Input Maximum Delay and Input Minimum Delay can be used to model the delay associated with an external device driving into an Altera FPGA. Figure 10–19 shows an example of the input delay path. For Figure 10–19, the Input Maximum Delay can be calculated as shown in Equation 10–19.

Equation 10–19.

$$\text{Input Maximum Delay} = \text{External Device Board Clock Path} + \text{External Device } t_{CO} + \text{External Device to Altera Device Board Delay} - \text{External Clock Path to Altera Device}$$

Figure 10–19. Input Delay



Use the Tcl command `set_input_delay` to specify an input delay. The following example specifies an **Input Maximum** Delay assignment of 1.5 ns from clock node `clk` to input pin `data_in`:

```
set_input_delay -clk_ref clk -to "data_in" -max 1.5ns
```

The following example specifies an Input Minimum Delay assignment of 1 ns from clock node `clk` to input pin `data_in`:

```
set_input_delay -clk_ref clk -to "data_in" -min 1ns
```

When using **Input Delay** assignments, specify a particular clock reference. The clock reference is the clock that feeds the external register's clock port that feeds the Altera device. This allows the Quartus II Classic Timing Analyzer to perform the proper analysis for the input path.

The t_{SU}, t_H, t_{PD}, and min t_{PD} timing paths reported for input pins, where input delay internal to the Altera FPGA assignments has been applied, include only the data delay from these pins and do not account for any clock setup relationships, clock hold relationships, or slack.

Output Delay Assignment

You can specify external output delays with either Output Maximum Delay or Output Minimum Delay assignments. Make Output Maximum Delay assignments to specify the maximum delay of a signal from the specified FPGA output pin to an external register, relative to a specified clock source. Make Output Minimum Delay assignments to specify the minimum delay of a signal from the specified FPGA output pin to an external register relative to a specified clock source.

When performing a clock setup check, the Quartus II Classic Timing Analyzer subtracts the Output Maximum Delay assignment value from the data required time (or subtracts the assignment value from the point-to-point requirement).

When performing a clock hold check, the Quartus II Classic Timing Analyzer subtracts the Output Minimum Delay assignment value from the data required time (or subtracts the assignment value from the point-to-point requirement).

The value of this assignment usually represents the sum of the t_{SU} of the external device, the actual board delay from the output pin of the Altera device, and the board clock skew.

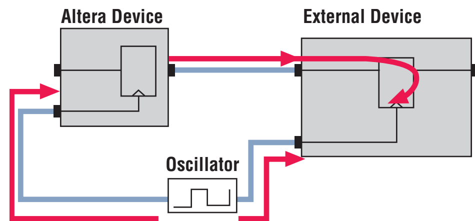
The Output Minimum Delay default value is the same as the Output Maximum Delay, and the Output Maximum Delay default value is the same as the Output Minimum Delay if only one is specified.

For example, use the Output Maximum Delay and Output Minimum Delay to model the delay associated with outputs for an Altera FPGA driving into an external device. Figure 10–20 shows an example of an output delay path. For Figure 10–20 the Output Maximum Delay can be calculated, as shown in Equation 10–20.

Equation 10–20.

$$\text{Output Maximum Delay} = \text{Altera Device to External Device Board Delay} + \text{External Device } t_{SU} + \text{External Clock Path to Altera Device} - \text{External Device Board Clock Path}$$

Figure 10–20. Output Delay



The Tcl command `set_output_delay` specifies an Output Delay assignment. The following example specifies an Output Maximum Delay assignment of 2 ns from clock `clk` to output pin `data_out`:

```
set_output_delay -clk_ref clk -to data_out -max 2ns
```

The following example specifies an Output Minimum Delay assignment of 1 ns from clock `clk` to output pin `data_out`:

```
set_output_delay -clk_ref clk -to data_out -min 1ns
```

When using output delay assignments, specify a specific clock reference. The clock reference is the clock that feeds the external register's clock port that is fed by the Altera device. This allows the Quartus II Classic Timing Analyzer to perform the correct static timing analysis on the output path.

The t_{CO}, minimum t_{CO}, t_{PD}, and minimum t_{PD} timing paths reported for output pins, where output delay assignments have been applied include only the data delay internal to the Altera FPGA to those pins, and do not account for any clock setup relationships, clock hold relationships, or slack.

Virtual Clocks

You can use virtual clocks to model clock signals outside of the Altera FPGA, that is, clocks that do not directly drive anything within the Altera FPGA. For example, you can use a virtual clock to model a clock signal feeding an external output register that feeds the Altera FPGA.

Using the `-virtual` option of the `create_base_clock` Tcl command specifies a virtual clock assignment.

Before a you can use virtual clock for either an input or output delay assignment, the virtual clock must have the Virtual Clock Reference assignment enabled for the virtual clock setting.

The code in Example 10–4 creates a virtual clock named `virt_clk`, with a 200 MHz requirement, and uses the virtual clock setting as the clock reference for the input delay assignment.

Example 10–4. Creating a Virtual Clock Named virt_clk

```
#create the virtual clock setting
create_base_clock -fmax 200MHz -virtual virt_clk

#enable the virtual clock reference for the virtual clock setting
set_instance_assignment -name VIRTUAL_CLOCK_REFERENCE On -to virt_clk

#use the virtual clock setting as the clock reference for the input delay assignment
set_input_delay -clk_ref virt_clk -to data_in -max 2ns
```

Asynchronous Paths

The Quartus II Classic Timing Analyzer can analyze asynchronous signals that connect to the clear, preset, or load ports of a register. This section explains how the Quartus II Classic Timing Analyzer analyzes asynchronous paths.

Recovery and Removal

Recovery time is the minimum length of time an asynchronous control signal; for example, clear and preset, must be stable before the active clock edge. Removal time is the minimum length of time an asynchronous control signal must be stable after the active clock edge. The Enable Recovery/Removal analysis option reports the results of recovery and removal checks for paths that end at an asynchronous clear, preset, or load signal of a register.

Enable the recovery and removal analysis with the following Tcl command:

```
set_global_assignment -name ENABLE_RECOVERY_REMOVAL_ANALYSIS ON
```

With this option enabled, the Quartus II Classic Timing Analyzer reports the result of the recovery analysis and removal analysis.

By default, the recovery and removal analysis is disabled. You should enable his option for all designs that contain asynchronous controls signals.

Recovery Report

When you set `ENABLE_RECOVERY_REMOVAL_ANALYSIS` to **ON**, the Quartus II Classic Timing Analyzer determines the recovery time as the minimum amount of time required between an asynchronous control signal becoming inactive and the next active clock edge, compares this to your design, and reports the results as slack. The Recovery report alerts you to conditions where an active clock edge occurs too soon after the asynchronous input becomes inactive, rendering the register's data uncertain.

The recovery slack time calculation is similar to the calculation for clock setup slack, which is based on data arrival time and data required time except for asynchronous control signals. If the asynchronous control is registered, the Quartus II Classic Timing Analyzer calculates the recovery slack time using Equation 10–21 through Equation 10–23.

Equation 10–21.

$$\text{Recovery Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}$$

Equation 10–22.

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Longest Clock Path to Source Register} + \text{micro } t_{CO} \text{ of Source Register} + \text{Longest Register-to-Register Delay}$$

Equation 10–23.

$$\text{Data Required Time} = \text{Latch Edge} + \text{Longest Clock Path to Source Register} + \text{micro } t_{SU} \text{ of Destination Register}$$

Example 10–5 shows recovery time as reported by the Timing Analyzer.

Example 10–5. Recovery Time Reporting for a Registered Asynchronous Reset Signal

```
Info: Slack time is 8.947 ns for clock "a_clk" between source register "async_reg1" and
destination register "reg_1"
    Info: Requirement is of type recovery
    Info: - Data arrival time is 4.028 ns
        Info: + Launch edge is 0.000 ns
            Info: + Longest clock path from clock "a_clk" to source register is 3.067 ns
        Info: + Micro clock to output delay of source is 0.094 ns
        Info: + Longest register to register delay is 0.867 ns
    Info: + Data required time is 12.975 ns
        Info: + Latch edge is 10.000 ns
            Info: + Shortest clock path from clock "a_clk" to destination register is 3.065 ns
        Info: - Micro setup delay of destination is 0.090 ns
```

If the asynchronous control is not registered, the Quartus II Classic Timing Analyzer uses Equation 10–24 through Equation 10–26 to calculate the recovery slack time.

Equation 10–24.

$$\text{Recovery Slack Time} = \text{Data Required Time} - \text{Data Arrival Time}$$

Equation 10–25.

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Maximum Input Delay} + \text{Maximum Pin-to-Register Delay}$$

Equation 10–26.

$$\text{Data Required Time} = \text{Latch Edge} + \text{Shortest Clock Path to Destination Register Delay} - \text{micro } t_{SU} \text{ of Destination Register}$$

Example 10–6 shows recovery time as reported by the Timing Analyzer.

Example 10–6. Recovery Time Reporting for a Non-Registered Asynchronous Reset Signal

```
Info: Slack time is 8.744 ns for clock "a_clk15" between source pin "a_arst2" and
destination register "inst5"
       Info: Requirement is of type recovery
       Info: - Data arrival time is 4.787 ns
              Info: + Launch edge is 0.000 ns
              Info: + Max Input delay of pin is 1.500 ns
              Info: + Max pin to register delay is 3.287 ns
       Info: + Data required time is 13.531 ns
    Info: + Latch edge is 10.000 ns
    Info: + Shortest clock path from clock "a_clk15" to destination register is 3.542 ns
              Info: - Micro setup delay of destination is 0.011 ns
```

If the asynchronous reset signal is from a device pin, an Input Maximum Delay assignment must be made to the asynchronous reset pin for the Quartus II Classic Timing Analyzer to perform recovery analysis on that path.

Removal Report

When you set ENABLE_RECOVERY_REMOVAL_ANALYSIS to **ON**, the Quartus II Classic Timing Analyzer determines the removal time as the minimum amount of time required between an active clock edge that occurs while an asynchronous input is active, and the deassertion of the asynchronous control signal. The Quartus II Classic Timing Analyzer then compares this to your design and reports the results as slack. The Removal report alerts you to a condition in which an asynchronous input signal goes inactive too soon after a clock edge, thus rendering the register's data uncertain.

The removal time slack calculation is similar to the one used to calculate clock hold slack, which is based on data arrival time and data required time except for asynchronous control signals. If the asynchronous control is registered, the Quartus II Classic Timing Analyzer uses Equation 10–27 through Equation 10–29 to calculate the removal slack time.

Equation 10–27.

$$\text{Removal Slack Time} = \text{Data Arrival Time} - \text{Data Required Time}$$

Equation 10–28.

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Shortest Clock Path from Source Register Delay} + \text{micro } t_{CO} \text{ of Source Register} + \text{Shortest Register-to-Register Delay}$$

Equation 10–29.

$$\text{Data Required Time} = \text{Latch Edge} + \text{Longest Clock Path to Destination Register Delay} + \text{micro } t_H \text{ of Destination Register}$$

Example 10–7 shows removal time as reported by the Quartus II Classic Timing Analyzer.

Example 10–7. Removal Time Reporting for a Registered Asynchronous Reset Signal

```
Info: Minimum slack time is 814 ps for clock "a_clk" between source register "async_reg1"
and destination register "reg_1"
    Info: Requirement is of type removal
    Info: + Data arrival time is 4.028 ns
        Info: + Launch edge is 0.000 ns
        Info: + Shortest clock path from clock "a_clk" to source register is 3.067 ns
        Info: + Micro clock to output delay of source is 0.094 ns
        Info: + Shortest register to register delay is 0.867 ns
    Info: - Data required time is 3.214 ns
        Info: + Latch edge is 0.000 ns
        Info: + Longest clock path from clock "a_clk" to destination register is 3.065 ns
        Info: + Micro hold delay of destination is 0.149 ns
```

If the asynchronous control is not registered, the Quartus II Classic Timing Analyzer uses Equation 10–30 through Equation 10–32 to calculate the removal slack time.

Equation 10–30.

$$\text{Removal Slack Time} = \text{Data Arrival Time} - \text{Data Required Time}$$

Equation 10–31.

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Input Minimum Delay of Pin} + \text{Minimum Pin-to-Register Delay}$$

Equation 10–32.

$$\text{Data Required Time} = \text{Latch Edge} + \text{Longest Clock Path to Destination Register Delay} + \text{micro } t_H \text{ of Destination Register}$$

Example 10–8 shows removal time as reported by the Quartus II Classic Timing Analyzer.

Example 10–8. Removal Time Reporting for a Non-Registered Asynchronous Reset Signal

```
Info: Minimum slack time is 1.131 ns for clock "a_clk15" between source pin "a_arst2"
and destination register "inst5"
          Info: Requirement is of type removal
          Info: + Data arrival time is 4.787 ns
    Info: + Launch edge is 0.000 ns
    Info: + Min Input delay of pin is 1.500 ns
    Info: + Min pin to register delay is 3.287 ns
          Info: - Data required time is 3.656 ns
    Info: + Latch edge is 0.000 ns
    Info: + Longest clock path from clock "a_clk15" to destination register is 3.542 ns
       Info: + Micro hold delay of destination is 0.114 ns
```

If the asynchronous reset signal is from a device pin, an Input Minimum Delay assignment must be made to the asynchronous reset pin for the Quartus II Classic Timing Analyzer to perform a removal analysis on this path.

Skew Management

Clock skew is the difference in the arrival times of a clock signal at two different registers, which can be caused by path length differences between two clock paths, or by using gated or rippled clocks. As clock periods become shorter and shorter, the skew between data arrival times and clock arrival times becomes more significant. The Quartus II Classic Timing Analyzer provides two assignments for analyzing and constraining skew for data and clock signals.

Maximum Clock Arrival Skew

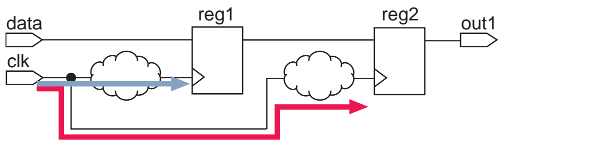
Make Maximum Clock Arrival Skew assignments to specify the maximum allowable clock arrival skew between a clock signal and various destination registers. The Quartus II Classic Timing Analyzer compares the longest clock path to the registers' clock port and the shortest clock path to the registers' clock port to determine if your maximum clock arrival skew is achieved. Maximum clock arrival skew is calculated using Equation 10–33.

Equation 10–33.

$$\text{Maximum Clock Arrival Skew} = \text{Longest Clock Path} - \text{Shortest Clock Path}$$

For example, if the delay from clock pin `clk` to the clock port of register `reg1` is 1.0 ns, and the delay from clock pin `clk` to the clock port of register `reg2` is 3.0 ns, as shown in Figure 10–21, the Quartus II Classic Timing Analyzer provides a clock skew slack time of 2.0 ns.

Figure 10–21. Clock Arrival Paths



You should apply the Maximum Clock Arrival Skew assignment to a clock node and a group of registers. When you make a Maximum Clock Arrival Skew assignment, the Fitter attempts to satisfy the skew requirement.

You can use the `set_instance_assignment -name max_clock_arrival_skew` Tcl command to specify a Maximum Clock Arrival Skew assignment. The following example specifies a maximum clock arrival skew of 1 ns from clock signal `clk` to the bank of registers matching `reg*`:

```
set_instance_assignment -name max_clock_arrival_skew 1ns -from clk \
-to reg*
```

Maximum Data Arrival Skew

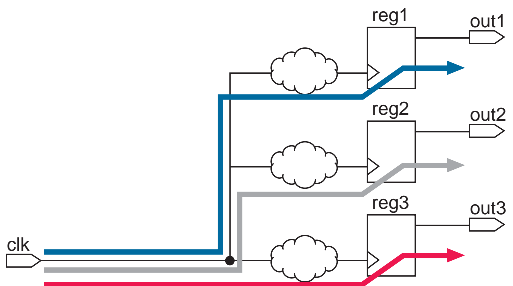
Make Maximum Data Arrival Skew assignments to specify the maximum allowable data arrival skew to various destination registers or pins. The Quartus II Classic Timing Analyzer compares the longest data arrival path to the shortest data arrival path to determine if your specified maximum data arrival skew is achieved. Maximum data arrival skew is calculated using Equation 10–34.

Equation 10–34.

$$\text{Maximum Data Arrival Skew} = \text{Longest Data Arrival Path} - \text{Shortest Data Arrival Path}$$

For example, if the data arrival time to output pin `out1` is 2.0 ns, the data arrival time to output pin `out2` is 1.5 ns, and the data arrival time to output pin `out3` is 1.0 ns, as shown in Figure 10–22, the Quartus II Classic Timing Analyzer provides a maximum data arrival skew slack time of 1.0 ns.

Figure 10–22. Data Arrival Paths



When you make a Maximum Data Arrival Skew assignment, the Fitter attempts to satisfy the skew requirement.

You can use the `set_instance_assignment -name max_data_arrival_skew` Tcl command to specify a maximum data arrival skew value. The following example specifies a maximum data arrival skew of 1 ns from clock signal `clk` to the bank of output pins `dout`:

```
set_instance_assignment -name max_data_arrival_skew 1ns -from clk \
-to dout[*]
```

Generating Timing Analysis Reports with report_timing

The Quartus II Classic Timing Analyzer includes the `report_timing` Tcl command for generating text-based timing analysis reports. You can customize the output of `report_timing` using multiple switches that allow the generation of both detailed and general timing reports on any path in the design.

The `report_timing` Tcl command is available in the `quartus_tan` executable.

Prior to using the `report_timing` Tcl command, you must open a Quartus II project and create a timing netlist. For example, the following two Tcl commands accomplish this:

```
project_open my_project
create_timing_netlist
```

The `report_timing` Tcl command provides `-from` and `-to` switches for filtering specific source and destination nodes. For example, the following `report_timing` Tcl command reports all clock setup paths, with the switch `-clock_setup`, between registers `src_reg*` and `dst_reg*`. The `-npaths` 20 switch limits the report to 20 paths.

```
report_timing -clock_setup -from src_reg* -to dst_reg* -npaths 20
```

The switches `-clock_filter` and `-src_clock_filter` are also available for filtering based on specific clock sources. For example, the following `report_timing` Tcl command reports all clock setup paths where the destination registers are clocked by `clk`:

```
report_timing -clock_setup -clock_filter clk
```

The following example reports clock setup paths where the destination registers are clocked by `clk`, and the source registers are clocked by `src_clock`.

```
report_timing -clock_setup -clock_filter clk -src_clock_filter src_clk
```

Example 10–9 is an example script that can be sourced by the `quartus_tan` executable:

Example 10–9. Source for the quartus_tan Executable

```
# Open a project
project_open my_project
# Always create the netlist first
create_timing_netlist
# List clock setup paths for clock clk
# from registers abc* to registers xyz*
report_timing -clock_setup -clock_filter clk -from abc* -to xyz*
# List the top 5 pin-to-pin combinational paths
report_timing -tpd -npaths 5
# List the top 5 pin-to-pin combinational paths and
# write output to an out.tao file
report_timing -tpd -npaths 5 -file out.tao
# Compute min tpd and append results to existing out.tao
report_timing -min_tpd -npaths 5 -file out.tao -append
# Show longest path (register to register data path) between a* and b*
report_timing -longest_paths -npaths 1
delete_timing_netlist
project close
```

Other Timing Analyzer Features

The Quartus II Classic Timing Analyzer provides many features for customizing and increasing the efficiency of static timing analysis, including:

- Wildcard assignments
- Assignment groups
- Fast corner analysis
- Early timing estimation
- Timing constraint checker
- Latch analysis

Wildcard Assignments

To simplify the tasks of making assignments to many node assignments, the Quartus II software accepts the * and ? wildcard characters. Use these wildcard characters to reduce the number of individual assignments you need to make for your design.

The "*" wildcard character matches any string. For example, given an assignment made to a node specified as `reg*`, the Quartus II Classic Timing Analyzer searches and applies the assignment to all design nodes that match the prefix `reg` with none, one, or several characters following, such as `reg1`, `reg[2]`, `regbank`, and `reg12bank`.

The "?" wildcard character matches any single character. For example, given an assignment made to a node specified as `reg?`, the Quartus II Classic Timing Analyzer searches and applies the assignment to all design nodes that match the prefix `reg` and any single character following, such as `reg1`, `rega`, and `reg4`.

Assignment Groups

Assignment groups, also known as time groups, allow you to define a custom group of nodes to which you can assign timing assignments. You can also exclude specific nodes, wildcards, and time groups from a time group.

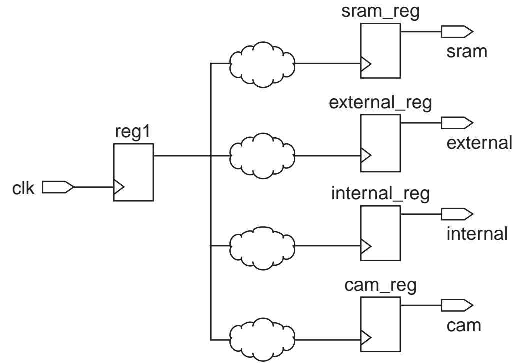
Use the `timegroup` Tcl command to create an assignment group. The following example creates an assignment group `srcgrp` and adds nodes with names that match `src1*` to the group:

```
timegroup srcgrp -add_member src1*
```

For example, Figure 10–23 has false paths between source register `reg1` and destination register bank `sram_reg`, `external_reg`, `internal_reg`, and `cam_reg` that need to be cut. Without the use of assignment groups, the assignments required are:

```
set_timing_cut_assignment -from reg1 to sram_reg
set_timing_cut_assignment -from reg1 to external_reg
set_timing_cut_assignment -from reg1 to internal_reg
set_timing_cut_assignment -from reg1 to cam_reg
```

Figure 10–23. False Path



With an assignment group called `dst_reg_bank`, the assignments required are:

```
#create a time group called dst_reg
timegroup dst_reg_bank -add_member sram_reg
timegroup dst_reg_bank -add_member external_reg
timegroup dst_reg_bank -add_member internal_reg
timegroup dst_reg_bank -add_member cam_reg
#cut timing paths
set_timing_cut_assignment -from reg1 to dst_reg_bank
```

Once an assignment group has been defined, applicable timing assignment can be made to the time group without redefining the assignment group.

Assigning individual nodes to time groups and applying timing assignments to these time groups can improve the performance of the Quartus II Classic Timing Analyzer.

Fast Corner Analysis

Fast Corner Analysis uses timing models generated under best-case conditions (voltage, process, and temperature) for the fastest speed-grade device.

Both Fast Corner and Slow Corner static timing analysis reports are saved to the <*project name*>**.tan.rpt** file, potentially overwriting previous timing analysis reports. To preserve a copy of your reports, save the file with a new name before the next compilation or static timing analysis, or use the Combined Fast/Slow Analysis report feature.

The Quartus II software also reports minimum delay checks after a slow corner (default) analysis. These results are generated by reporting minimum delay checks using worst-case timing models.

To perform fast corner static timing analysis with the best-case timing models, you can use the `switch --fast_model=on` with the `quartus_tan` executable. The following Tcl command enables the fast timing models:

```
quartus_tan <project_name> --fast_model=on
```


© March 2009 Altera Corporation

Early Timing Estimation

The majority of Quartus II software compilation time is consumed by the place-and-route process used to obtain optimal design results. To accelerate the design process for large designs, the Quartus II software provides **Early Timing Estimation**. This feature provides a quick static timing analysis in a fraction of the time required for a full compilation by performing a preliminary place-and-route on the design without full optimizations, which reduces total compile time by up to five times compared to a fully fitted design.

An Early Timing Estimate fit is not fully optimized or legally routed. The timing delay report is only an estimate. Typically, the estimated delays are within 10% of those obtained with a full fit when the realistic setting is used.

The Early Timing Estimate has three settings for generating timing estimates: Realistic, Optimistic, and Pessimistic. Table 10–1 describes these settings.

Table 10–1. Early Timing Estimate Setting Options

Setting	Description
Realistic (default setting: estimates final timing using standard fitting)	Generates timing estimates that are likely to be closest to full compilation results.
Optimistic (estimates best-case final timing)	Generates timing estimates that are unlikely to be exceeded by full compilation.
Pessimistic (estimates worst-case final timing)	Generates timing estimates that are likely to be exceeded by full compilation.

To use the **Early Timing Estimate** feature, enter the following Tcl command when performing a fit:

```
quartus_fit \
--early_timing_estimate[=<realistic|optimistic|pessimistic>]
```

After **Early Timing Estimate** is complete, a full timing report is generated based on the early placement and routing delays. In addition, you can view the preliminary logic placement in the Timing Closure floorplan. The early timing placement allows you to perform initial placement and view the timing interaction of various placement topology.

Timing Constraint Checker

Altera recommends that you enter all timing constraints into the Quartus II software prior to performing a full compilation. This ensures that the Fitter targets the correct timing requirements and ensures that the Quartus II Classic Timing Analyzer reports the correct violations for all timing paths in the design. To ensure that all constraints have been applied to design nodes, the **Timing Constraint Check** feature reports all unconstraint paths in your design. Example 10–10 shows the timing constraint check summary generated after a full compilation.

Example 10–10. Timing Constraint Check Summary

```
+-------------------------------------------------------------------------------------+
; Timing Constraint Check Summary                                                     ;
+---------------------------------------+---------------------------------------------+
; Timing Constraint Check Status        ; Analyzed - Tue Feb 28 11:42:31 2006      ;
; Quartus II Version               ; 6.1 Internal Build 143 02/20/2006 SJ Full Version ;
; Revision Name                         ; test                                        ;
; Top-level Entity Name                 ; Block1                                      ;
; Unconstrained Clocks                  ; 0                                           ;
; Unconstrained Paths (Setup)           ; 22                                          ;
; Unconstrained Reg-to-Reg Paths (Setup) ; 0                                          ;
; Unconstrained I/O Paths (Setup)       ; 22                                          ;
; Unconstrained Paths (Hold)            ; 12                                          ;
; Unconstrained Reg-to-Reg Paths (Hold)  ; 0                                          ;
; Unconstrained I/O Paths (Hold)        ; 12                                          ;
+---------------------------------------+---------------------------------------------+
```

To perform a timing constraint check, use the `switch --check_constraints` with the `quartus_tan` executable. The following Tcl command performs a timing constraint check on both setup and hold on the design system:

```
quartus_tan block1 --check_constraints=both
```

Latch Analysis

Latches are implemented in the Quartus II software as look-up-tables (LUTs) feeding back onto themselves. The Quartus II Classic Timing Analyzer can analyze these latches as synchronous elements rather than as combinational elements. The clock enables are analyzed as inverted clocks. The Quartus II Classic Timing Analyzer reports the results of setup and hold analysis on these latches.

You can turn on the Analyze Latches As Synchronous Elements option with the following Tcl command:

```
set_global_assignment -name ANALYZE_LATCHES_AS_SYNCHRONOUS_ELEMENTS ON
```

Timing Analysis Using the Quartus II GUI

In addition to the extensive scripting support available in the Quartus II Classic Timing Analyzer, the Quartus II software provides the Assignment Editor and other user interface tools, giving you access to the Quartus II Classic Timing Analyzer features and assignments.

Assignment Editor

The Assignment Editor is a spreadsheet-style interface used for adding, modifying, and deleting timing assignments.

To make timing assignments in the Assignment Editor, choose **Timing** from the category list to cause the Assignment Name column to display only timing assignments. Double-click **<<new>>** in the Assignment Name field, the Assignment Name list displays. Figure 10–24 shows the Assignment Editor with the Assignment Name list displaying timing assignment types.

Figure 10–24. Assignment Editor

For more information about the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

Timing Settings

You can specify delay requirements and clock settings with the **Timing Analysis Settings** page of the **Settings** dialog box.

To access this page, on the Assignments menu, click **Settings**. In the **Category** list, click the "+" icon next to **Timing Analysis Settings** to expand the folder. (Be sure that the **Use Classic Timing Analyzer during compilation** radio button is turned on.) Select **Classic Timing Analyzer Settings**. The **Classic Timing Analysis Settings** page appears.

Clock Settings Dialog Box

You can create or modify base clock settings or derived clock settings using the **Clock Settings** dialog box. To access this page, on the Assignments menu, click **Settings**. In the **Category** list, click the "+" icon next to **Timing Analysis Settings** to expand the folder. (Be sure that the **Use Classic Timing Analyzer during compilation** radio button is turned on.) Click on **Classic Timing Analyzer Settings**. The Timing **Analysis Settings** page displays. Under **Clock Settings**, click **Individual Clocks**. The **Individual Clock** dialog box appears.

Click the **New** button in the **Individual Clocks** dialog box to access the **New Clock Settings** dialog box and create a base or derived clock setting.

More Timing Settings Dialog Box

On the **Timing Analysis Settings** page of the **Settings** dialog box, click **More Settings** to display the **More Timing Settings** dialog box. The **More Timing Settings** dialog box provides access to many global timing analysis options.

Timing Reports

The Quartus II Classic Timing Analyzer report is a section of the Compilation Report containing the static timing analysis results. The Quartus II Classic Timing Analyzer report includes clock setup and clock hold measurements for all clock sources. The report also shows t_{CO} for all output pins, t_{SU} and t_H for all input pins, and t_{PD} for any pin-to-pin combinational paths in the design. Other reports are created for different analyses and device features.

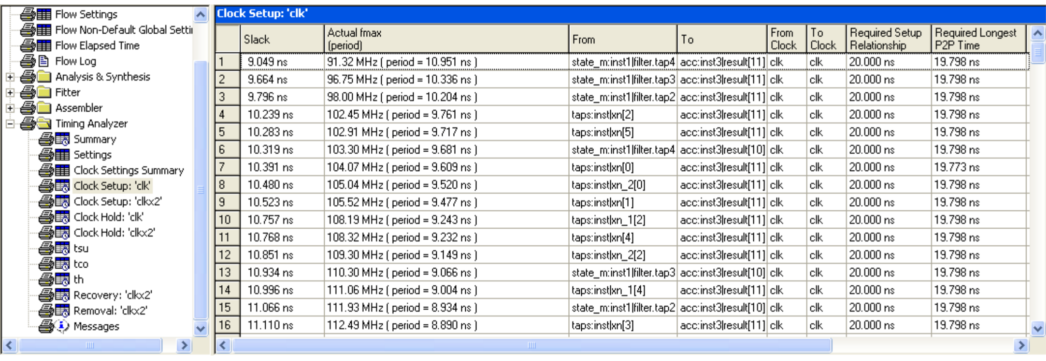
In the **Settings** dialog box, you can specify the range of information to be reported in the timing analysis of the Compilation Report. To access this page, on the Assignments menu, click **Settings**. In the **Category** list, click the ⊞ icon next to **Timing Analysis Settings** to expand the folder. (Be sure that the **Use Classic Timing Analyzer during compilation** radio button is turned on.) Click the ⊞ icon next to **Classic Timing Analyzer Settings** to expand the folder. Click **Classic Timing Analyzer Reporting**. The **Classic Timing Analyzer Reporting** dialog box appears.

If there are no timing assignments for the design, the Quartus II Classic Timing Analyzer does not generate slack reports for any detected clock nodes. The Quartus II Classic Timing Analyzer only reports slack measurements for pins with individual or global t_{SU}, t_H, or t_{CO} assignments. A positive slack indicates the margin by which the path surpasses the clock timing requirements. A negative slack indicates the margin by which the path fails the clock timing requirements.

This Timing Analysis report is also available in text format located in the design directory with the file name <*revision name*>**.tan.rpt**.

In the Compilation Report, select an analysis type under the Timing Analyzer folder to display the analysis report; for example, Clock Setup or Clock Hold. Figure 10–25 shows an example of a Clock Setup report for clock signal `clk`.

Figure 10–25. Timing Analysis Report

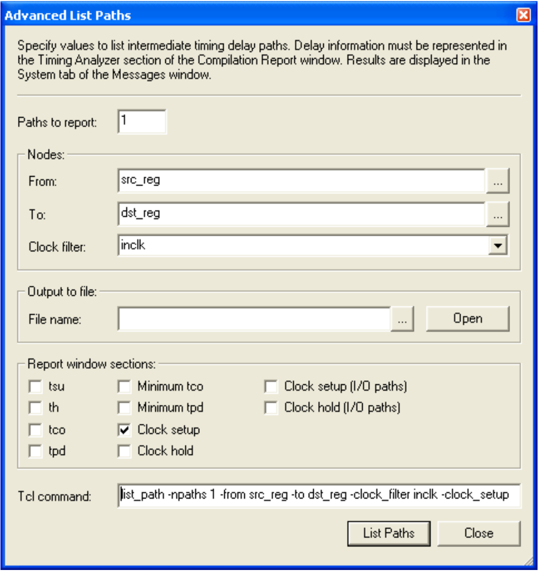


Advanced List Path

The **Advanced List Paths** dialog box provides detailed information about a specific path, such as interconnect and cell delays between any two valid register-to-register paths (Figure 10–26).

The **Advanced List Paths** dialog box allows you to select the type of paths you want listed. For example, you can obtain detailed information for Clock Setup and Clock Hold for a specific clock. In addition, the Tcl command field in the window matches the equivalent Tcl command you can use in either a custom Tcl script or in the Tcl console.

Figure 10–26. Advanced List Paths Dialog Box



You can perform a list path command directly from the Timing Analysis report. To do this, right click a path and click **List Path** (Figure 10–27). To launch the **Advanced List Paths** dialog box, right-click a path and in the menu that appears, and select **Advanced List Paths**.

The **Advanced List Paths** dialog box displays only paths that are visible in the Timing Analysis report. To increase the amount of paths reported by the Quartus II Classic Timing Analyzer, on the Assignments menu, click **Timing Analysis Settings**. In the **Category** list, expand **Timing Analysis Settings** and select **Timing Analyzer Reporting**. In the **Timing Analyzer Reporting page**, specify the range of information to be reported by the Quartus II Classic Timing Analyzer.

Both the **Advanced List Paths** and the **List Path** commands display the path information in the **System** message window.

Figure 10–27. List Path in the Message Window

```
Info: Slack time is 8.064 ns for clock "clk3" between source register "reg1" and destination register "reg2"
  Info: Fmax is restricted to 275.03 MHz due to tcl and tch limits
  Info: + Largest register to register requirement is 8.891 ns
    Info: + Setup relationship between source and destination is 10.000 ns
    Info: + Largest clock skew is -0.848 ns
    Info: - Micro clock to output delay of source is 0.224 ns
    Info: - Micro setup delay of destnation is 0.037 ns
  Info: - Longest register to register delay is 0.827 ns
    Info: 1: + IC(0.000 ns) + CELL(0.000 ns) = 0.000 ns; Loc. = LC_X33_Y20_N6; Fanout = 1; REG Node = 'reg1'
    Info: 2: + IC(0.518 ns) + CELL(0.309 ns) = 0.827 ns; Loc. = LC_X33_Y20_N2; Fanout = 2; REG Node = 'reg2'
    Info: Total cell delay = 0.309 ns ( 37.36 % )
    Info: Total interconnect delay = 0.518 ns ( 62.64 % )
```

If the **Combined Fast/Slow Timing** option is enabled, the **List Path** Tcl command displays only path delays reported in the Slow Model section.

Early Timing Estimate

To start an Early Timing Estimate, on the Processing menu, point to **Start** and click **Start Early Timing Estimate**. To specify the Early Timing Estimate mode, on the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Processes Settings**, select **Early Timing Estimate** and click the desired timing estimate mode. For more information about the Early Timing Estimate feature, refer to "Early Timing Estimation" on page 10–33.

Assignment Groups

To define, modify, and delete assignment groups, also known as time groups, from a single dialog box, on the Assignments menu, click **Assignment** (**Time) Groups**. The **Assignment Groups** dialog box appears.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp
```

For more information in PDF form, refer to the *Quartus II Scripting Reference Manual*.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For information about all settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Reference Manual*.

For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Creating Clocks

There are two Tcl commands that allow you to define clocks in a design, `create_base clock` and `create_relative_clock`.

Base Clocks

Use the `create_base_clock` Tcl command to define a base clock:

```
create_base_clock [-h | -help] [-long_help] -fmax <fmax> \
[-duty_cycle <integer>] [-virtual] [-target <name>] [-no_target] \
[-entity <entity>] [-disable] [-comment <comment>] <clock_name>
```

To define a base clock setting named `sys_clk` with a 100 MHz requirement applied to node `clk_src`, enter the following Tcl command:

```
create_base_clock -fmax 100MHz -target clk_src sys_clk
```

Derived Clocks

Use the `create_relative_clock` Tcl command to define a relative clock:

```
create_relative_clock [-h | -help] [-long_help] \
-base_clock <Base clock> [-duty_cycle <integer>] \
[-multiply <integer>] [-divide <integer>] [-offset <offset>] \
[-phase_shift <integer>] [-invert] [-virtual] [-target <name>] \
[-no_target] [-entity <entity>] [-disable] \
[-comment <comment>] <clock_name>
```

To define a relative clock named `aux_clk` based upon base clock setting `sys_clk` with a multiplication factor of 2 applied to node `rel_clk`, enter the following Tcl command:

```
create_relative_clock -base_clock sys_clk -multiply 2 \
-target rel_clk aux_clk
```

Clock Latency

You can use the `set_clock_latency` Tcl command to create either an early or late clock latency assignment:

```
set_clock_latency [-h | -help] [-long_help] [-early] [-late] \
-to <to> [<value>]
```

To apply an early clock latency of 1 ns and a late clock latency of 2 ns to clock node `clk`, enter the following Tcl commands:

```
set_clock_latency -early -to clk 2ns
```

Clock Uncertainty

You can use the `set_clock_uncertainty` Tcl command to create clock uncertainty assignments as shown in the following example:

```
set_clock_uncertainty [-h] [-help] [-long_help [-from \
<source clock name> ] -to <destination clock name> [-setup] [-hold] \
[-remove] [-disable] [-comment <comment>] <value>
```

To apply a clock setup uncertainty of 50 ps between source clock node `clk_src` and destination clock node `clk_dst`, enter the following Tcl command:

```
set_clock_uncertainty -from clk_src -to clk_dst -setup 50ps
```

To apply a clock hold uncertainty of 25 ps between to clock node `clk_sys`, enter the following Tcl command:

```
set_clock_uncertainty -to clk_sys -setup 25ps
```

Cut Timing Paths

You can use the `set_timing_cut_assignment` Tcl command to create cut timing assignments:

```
set_timing_cut_assignment [-h | -help] [-long_help] \
[-from <from_node_list>] [-to <to_node_list>] [-remove] [-disable] \
[-comment <comment>]
```

To cut the timing path from source register `reg1` to destination register `reg2`, enter the following Tcl command:

```
set_timing_cut_assignment -from reg1 -to reg2
```

Input Delay Assignment

You can use the Tcl command `set_input_delay` to create input delay assignments:

```
set_input_delay [-h | -help] [-long_help] [-clk_ref <clock>] \
-to <input_pin> [-min] [-max] [-clock_fall] [-remove] [-disable] \
[-comment <comment>] [<value>]
```

To apply an input maximum delay of 2 ns to an input pin named `data_in` that feeds a register clocked by clock source `clk`, enter the following Tcl command:

```
set_input_delay -clk_ref clk -to data_in –max 2ns
```

Maximum and Minimum Delay

The following Tcl commands create the Maximum Delay and Minimum Relationship assignments, respectively:

```
set_instance_assignment -name MAX_delay <value> -from <node> -to <node>
set_instance_assignment -name MIN_delay <value> -from <node> -to <node>
```

To apply a Maximum Delay of 8 ns and a minimum of 5 ns between source register `reg1` and destination register `reg2`, enter the following Tcl command:

```
set_instance_assignment -name MAX_DELAY 8ns -from reg1 -to reg2
set_instance_assignment -name MIN_DELAY 5ns -from reg1 -to reg2
```

To apply a Maximum Delay of 10 ns for all paths from source clock `clk_src` to destination clock `clk_dst`, enter the following Tcl command:

```
set_instance_assignment -name MAX_DELAY 10ns -from clk_src -to clk_dst
```

Maximum Clock Arrival Skew

The following Tcl command defines the Maximum Clock Arrival Skew assignment:

```
set_instance_assignment -name max_clock_arrival_skew <value> \
-from <clock> -to <node>
```

To apply a Maximum Clock Arrival Skew of 1 ns for clock source `clk` to a predefined timegroup called `reg_group`, enter the following Tcl command:

```
set_instance_assignment -name max_clock_arrival_skew 1ns -from clk \
-to reg_group
```

Maximum Data Arrival Skew

To create Maximum Data Arrival Skew assignments, use the Tcl command `set_instance_assignment -name max_data_arrival`:

```
set_instance_assignment -name max_data_arrival_skew <value> \
-from <clock> -to <node>
```

To apply a Maximum Data Arrival Skew of 1 ns for clock source `clk` to a predefined timegroup of pins called `pin_group`, enter the following Tcl command:

```
set_instance_assignment -name max_data_arrival_skew 1ns -from clk \
-to pin_group
```

Multicycle

Use the `set_multicycle_assignment` Tcl command to create Multicycle assignments:

```
set_multicycle_assignment [-h | -help] [-long_help] [-setup] [-hold] \
[-start] [-end] [-from <from_list>] [-to <to_list>] [-remove] \
[-disable] [-comment <comment>] <path_multiplier>
```

To apply a Multicycle Setup of 2 and a Hold Multicycle of 1 between source register `reg1` and destination register `reg2`, enter the following Tcl commands:

```
set_multicycle_assignment -setup -end -from reg1 -to reg2 2
set_multicycle_assignment -hold -end -from reg1 -to reg2 1
```

To apply a Source Multicycle Setup of 2 between source register `reg1` and destination register `reg2`, enter the following Tcl command:

```
set_multicycle_assignment -setup -start -from reg1 -to reg2 1
```

To apply a multicycle setup of 2 for all paths from source clock `clk_src` to destination clock `clk_dst`, enter the following Tcl command:

```
set_multicycle_assignment -setup -end -from clk_src -to clk_dst 2
```

Output Delay Assignment

Use the Tcl command `set_output_delay` to create Output Delay assignments:

```
set_output_delay [-h | -help] [-long_help] [-clk_ref <clock>] \
-to <output_pin> [-min] [-max] [-clock_fall] [-remove] [-disable] \
[-comment <comment>] [<value>]
```

To apply an Output Maximum Delay of 3 ns to an output pin named `data_out` that is fed to a register clocked by clock source `clk`, enter the following Tcl command:

```
set_output_delay -clk_ref clk -to data_out -max 3ns
```

Report Timing

Use the `report_timing` Tcl command to generate timing reports:

```
report_timing [-h | -help] [-long_help] [-npaths <number>] [-tsu] \
[-th] [-tco] [-tpd] [-min_tco] [-min_tpd] [-clock_setup] \
[-clock_hold] [-clock_setup_io] [-clock_hold_io] [-clock_setup_core] \
[-clock_hold_core] [-recovery] [-removal] [-dqs_read_capture] \
[-stdout] [-file <name>] [-append] [-table <name>] [-from <names>] \
[-to <names>] [-clock_filter <names>] [-src_clock_filter <names>] \
[-longest_paths] [-shortest_paths] [-all_failures]
```

The following example generates a list of all clock setup paths for clock source `clk` from registers `src_reg*` to registers `dst_reg*`:

```
report_timing -clock_setup -clock_filter clk -from src_reg* \
-to dst_reg*
```

Setup and Hold Relationships

The following Tcl commands create Setup Relationship and Hold Relationship assignments, respectively:

```
set_instance_assignment -name SETUP_RELATIONSHIP <value> -from <node> \
-to <node>
set_instance_assignment -name HOLD_RELATIONSHIP <value> -from <node> \
-to <node>
```

To apply a Setup Relationship of 12 ns and a Hold Relationship of 2 ns between source register `reg1` and destination registers `reg2`, enter the following Tcl command:

```
set_instance_assignment -name SETUP_RELATIONSHIP 12ns -from reg1 \
-to reg2
set_instance_assignment -name HOLD_RELATIONSHIP 2ns -from reg1 -to reg2
```

To apply a setup relationship of 10 ns for all paths from source clock `clk_src` to destination clock `clk_dst`, enter the following Tcl command:

```
set_instance_assignment -name SETUP_RELATIONSHIP 10ns -from clk_src \
-to clk_dst
```

Assignment Group

Use the `timegroup` Tcl command to create assignment groups:

```
timegroup [-h | -help] [-long_help] [-add_member <name>] \
[-add_exception <name>] [-remove_member <name>] [-remove_exception \
<name>] [-get_members] [-get_exceptions] [-overwrite] [-remove] \
[-disable] [-comment <comment>] <group_name>
```

The following example creates an assignment group called `reg_bank` with members `dst_reg*`, and excludes register `dst_reg5`.

```
timegroup reg_bank -add_member dst_reg* -add_exception dst_reg5
```

Virtual Clock

Use the `create_relative_clock` with the `-virtual` switch to create Virtual Clock assignments:

```
create_relative_clock [-h | -help] [-long_help] -base_clock \
<Base clock> [-duty_cycle <integer>] [-multiply <integer>] \
[-divide <integer>] [-offset <offset>] [-phase_shift <integer>] \
[-invert] [-virtual] [-target <name>] [-no_target] [-entity <entity>] \
[-disable] [-comment <comment>] <clock_name>
```

To define a virtual clock derived from the base clock setting `clk_aux` named `brd_sys`, enter the following Tcl command:

```
create_relative_clock -base_clock clk_aux -virtual brd_sys
```

MAX+PLUS II Timing Analysis Methodology

This section describes the basic static timing analysis and assignments available in the Quartus II software that originated in the MAX+PLUS® II design software.

f_{MAX} Relationships

Maximum clock frequency is the fastest speed at which the design clock can run without violating internal setup and hold time requirements. The Quartus II software performs static timing analysis on both single- and multiple-clock designs.

Apply clock settings to all clock nodes in a design to ensure that you meet all performance requirements. Refer to “Clock Settings” on page 10–7 for more information.

Slack

Slack is the margin by which a timing requirement such as f_{MAX} is met or not met. Positive slack indicates the margin by which a requirement is met. Negative slack indicates the margin by which a requirement is not met. The Quartus II software determines slack using Equation 10–35 through Equation 10–38.

Equation 10–35.

$$\text{Clock Setup Slack} = \text{Longest Register-to-Register Requirement} - \text{Longest Register-to-Register Delay}$$

Equation 10–36.

$$\text{Register-to-Register Requirement} = \text{Setup Relationship} + \text{Largest Clock Skew} - \text{micro } t_{CO} \text{ of Source Register} - \text{micro } t_{SU} \text{ of Destination Register}$$

Equation 10–37.

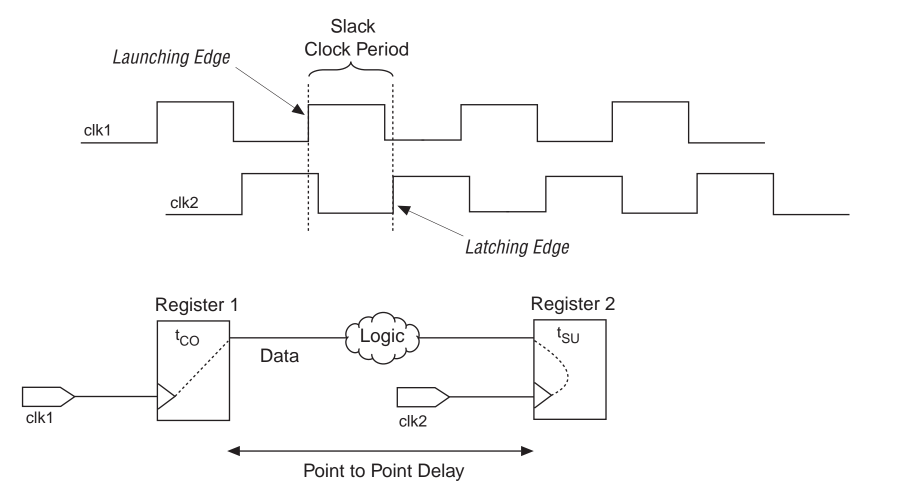
$$\text{Clock Hold Slack} = \text{Shortest Register-to-Register Delay} - \text{Smallest Register-to-Register Requirement}$$

Equation 10–38.

$$\text{Shortest Register-to-Register Requirement} = \text{Hold Relationship} + \text{Smallest Clock Skew} - \text{micro } t_{CO} \text{ of Source Register} - \text{micro } t_{H} \text{ of Destination Register}$$

Figure 10–28 shows a slack calculation diagram.

Figure 10–28. Slack Calculation Diagram



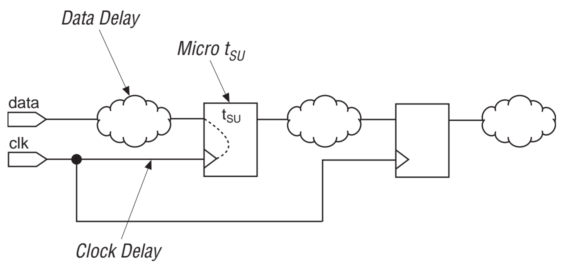
I/O Timing

This section describes the basic measurements made for I/O timing in the Quartus II software.

t_{SU} Timing

t_{SU} specifies the length of time data needs to arrive and be stable at an external input pin prior to a clock transition on an associated clock I/O pin. A t_{SU} requirement describes this relationship for an input register relative to the I/O pins of the FPGA. Figure 10–29 shows a diagram of clock setup time.

Figure 10–29. Clock Setup Time (t_{SU})



Micro t_{SU} is the internal setup time of the register. It is a characteristic of the register and is unaffected by the signals feeding the register. Equation 10–39 calculates the t_{SU} of data with respect to `clk` for the circuit shown in Figure 10–29.

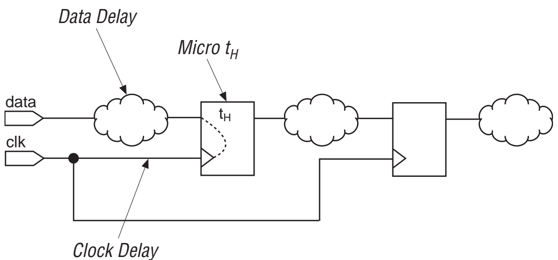
Equation 10–39.

$$t_{SU} = \text{Longest Data Delay} - \text{Shortest Clock Delay} + \text{micro } t_{SU} \text{ of Input Register}$$

t_H Timing

t_H specifies the length of time data needs to be held stable on an external input pin after a clock transition on an associated clock I/O pin. A t_H requirement describes this relationship for an input register relative to the I/O pins of the FPGA. Figure 10–30 shows a diagram of clock hold time.

Figure 10–30. Clock Hold Time (t_H)



Micro t_H is the internal hold time of the register. Equation 10–40 calculates the t_H of data with respect to `clk` for the circuit shown in Figure 10–30.

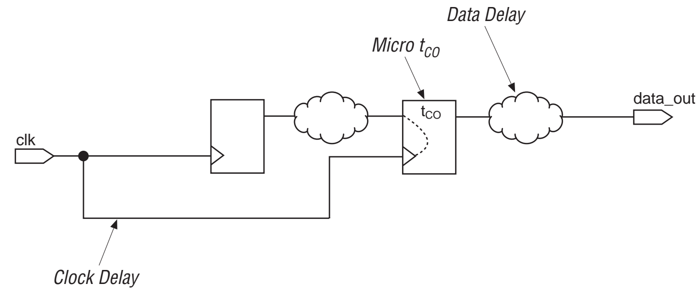
Equation 10–40.

$$t_H = \text{Longest Clock Delay} - \text{Shortest Data Delay} + \text{micro } t_H \text{ of Input Register}$$

t_{CO} Timing

Clock-to-output delay is the maximum time required to obtain a valid output at an output pin fed by a register, after a clock transition on the input pin that clocks the register. Micro t_{CO} is the internal clock-to-output delay of the register. Figure 10–31 shows a diagram of clock-to-output delay.

Figure 10–31. Clock-to-Output Delay (t_{CO})



Equation 10–41 calculates the t_{CO} for output pin `data_out` with respect to clock node `clk` for the circuit shown in Figure 10–31.

Equation 10–41.

$$t_{CO} = \text{Longest Clock Delay} + \text{micro } t_{CO} \text{ of Output Register}$$

Minimum t_{CO} (min t_{CO})

Minimum clock-to-output delay is the minimum time required to obtain a valid output at an output pin fed by a register, after a clock transition on the input pin that clocks the register. Micro t_{CO} is the internal clock-to-output delay of registers in Altera FPGAs. Unlike the t_{CO} assignment, the min t_{CO} assignment looks at the shortest delay paths (Equation 10–42).

Equation 10–42.

$$\text{min } t_{CO} = \text{Shortest Clock Delay} + \text{Shortest Data Delay} + \text{micro } t_{CO} \text{ of Output Register}$$

t_{PD} Timing

Pin-to-pin delay (t_{PD}) is the time required for a signal from an input pin to propagate through combinational logic and appear at an external output pin (Equation 10–43).

Equation 10–43.

$$t_{PD} = \text{Longest Pin-to-Pin Delay}$$

In the Quartus II software, you can make t_{PD} assignments between an input pin and an output pin.

Minimum t_{PD} (min t_{PD})

The minimum pin-to-pin delay (t_{PD}) is the time required for a signal from an input pin to propagate through combinational logic and appear at an external output pin. Unlike the t_{PD} assignment, the min t_{PD} assignment applies to the shortest pin-to-pin delay (Equation 10–44).

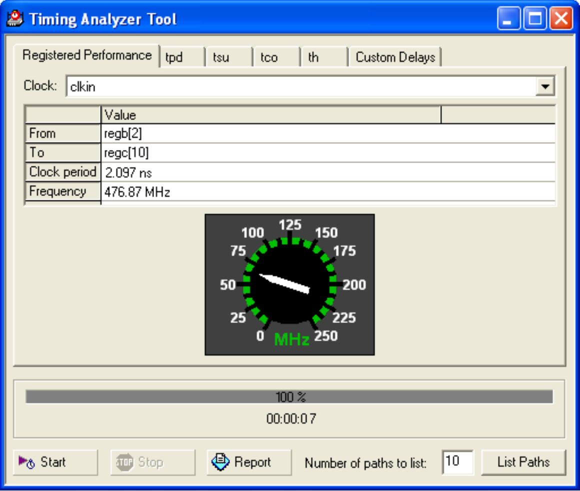
Equation 10–44.

$$\text{min } t_{PD} = \text{Shortest Pin-to-Pin Delay}$$

The Timing Analyzer Tool

To facilitate the classic static timing analysis flow and constraint, the Quartus II software provides a MAX+PLUS II-style Timing Analyzer Tool available on the Tools menu. The Timing Analyzer Tool provides a simple interface, similar to the Timing Analyzer tool in MAX+PLUS II, that reports register-to-register performance, I/O timing, and custom delay values (Figure 10–32).

Figure 10–32. Timing Analyzer Tool



Conclusion

Evolving design and aggressive process technologies require larger and higher-performance FPGA designs. Increasing design complexity demands enhanced static timing analysis tools that aid designers in verifying design timing requirements. Without advanced static timing analysis tools, you risk circuit failure in complex designs. The Quartus II Classic Timing Analyzer incorporates a set of powerful static timing analysis features critical in enabling system-on-a-programmable-chip (SOPC) designs.

Referenced Documents

This chapter references the following documents:

- *ALTPLL Megafunction User Guide*
- *AN 411: Understanding PLL timing for Stratix II Devices*
- *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Reference Manual*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 10–2 shows the revision history for this chapter.

Table 10–2. Document Revision History

Date and Version	Changes Made	Summary of Changes
March 2009 v9.0.0	This was chapter 9 in version 8.1.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	Added hyperlinks to referenced documents throughout the chapter. No other substantive changes were made.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

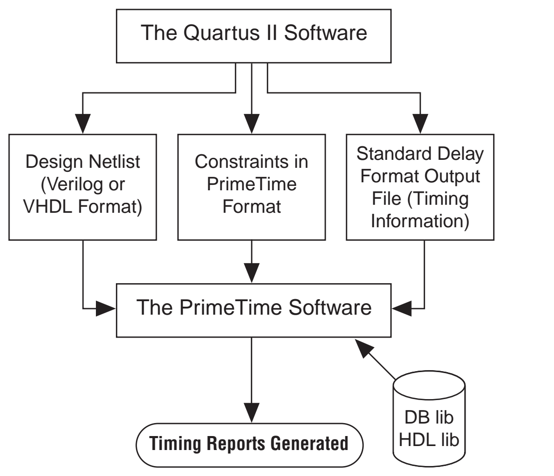
11. Synopsys PrimeTime Support

QII53005-9.0.0

Introduction

PrimeTime is an industry standard sign-off tool that performs static timing analysis on ASIC designs. The Quartus® II software makes it easy for designers to analyze their Quartus II projects using the PrimeTime software. The Quartus II software exports a netlist, design constraints (in the PrimeTime format), and libraries to the PrimeTime software environment. Figure 11–1 shows the PrimeTime flow diagram.

Figure 11–1. PrimeTime Software Flow Diagram



This chapter contains the following sections:

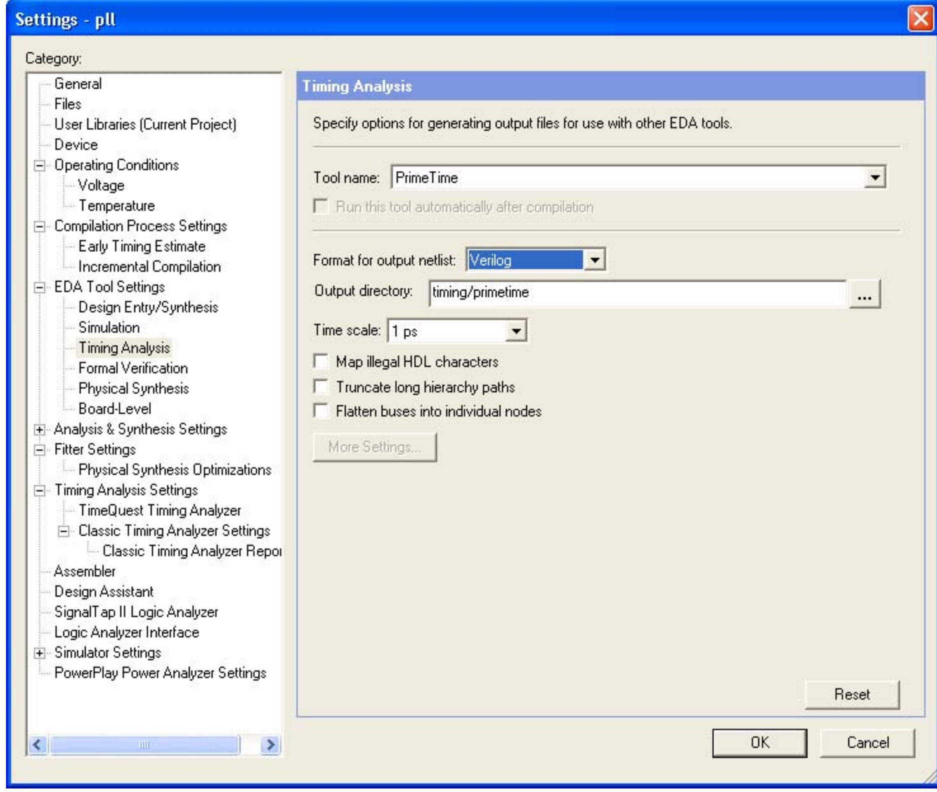
- "Quartus II Settings for Generating the PrimeTime Software Files"
- "Files Generated for the PrimeTime Software Environment" on page 11–2
- "Running the PrimeTime Software" on page 11–7
- "PrimeTime Timing Reports" on page 11–8
- "Static Timing Analyzer Differences" on page 11–18

Quartus II Settings for Generating the PrimeTime Software Files

To set the Quartus II software to generate files for the PrimeTime software, perform the following steps:

1. In the Quartus II software, on the Assignments menu, click **EDA Tool Settings**.
2. In the **Category** list, under **EDA Tool Settings,** select **Timing Analysis**.
3. In the **Tool name** list, select **PrimeTime**, and in the **Format for output netlist** list, select either **Verilog** or **VHDL**, depending on the HDL language you chose for use with the PrimeTime software (Figure 11–2).

Figure 11–2. Setting the Quartus II Software to Generate the PrimeTime Software Files



When you compile your project after making these settings, the Quartus II software runs the EDA Netlist Writer to create three files for the PrimeTime software. These files are saved in the <*revision_name*>**/timing/primetime** directory by default, where <*revision_name*> is the name of your Quartus II software revision. If it is not, you have used the wrong variable name.

Files Generated for the PrimeTime Software Environment

The Quartus II software generates a flattened netlist, a Standard Delay Output File (**.sdo**), and a Tcl script that prepares the PrimeTime software for timing analysis of the Quartus II project. These files are saved in the <*project directory*>**/timing/primetime** directory.

The Quartus II software uses the EDA Netlist Writer to generate PrimeTime files based on either the Quartus II Classic Timing Analyzer or the Quartus II TimeQuest Timing Analyzer static timing analysis results. When you run the EDA Netlist Writer, the PrimeTime SDO files are based on delays generated by the currently selected timing analysis tool in the Quartus II software.

To specify the timing analyzer, on the Assignments menu, click **Settings**. The **Settings** dialog box appears. Under **Category**, click **Timing Analysis Settings**. Select the timing analyzer of your choice.

For more information about specifying the Quartus II timing analyzers, refer to either the *Quartus II Classic Timing Analyzer* or the *Quartus II TimeQuest Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*. Also, refer to the *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* to help you decide which timing analyzer is most appropriate for your design.

The Netlist

Depending on whether **Verilog** or **VHDL** is selected as the **Format for output netlist** option, in the **Tool name** list on the **Timing Analysis** page of the **Settings** dialog box, the netlist is written and saved as either <*project name*>**.vo** or <*project name*>**.vho**, respectively. This file contains the flattened netlist representing the entire design.

When the Quartus II TimeQuest Timing Analyzer is selected, only a Verilog PrimeTime netlist is generated.

The SDO File

The Quartus II software saves the Standard Delay Format Output (**.sdo**) File as either <*revision_name*>**_v.sdo** or <*revision_name*>**_vhd.sdo**, depending on whether you selected **Verilog** or **VHDL** in the **Tool name** list on the **Timing Analysis** page of the **Settings** dialog box.

This file contains the timing information for each timing path between any two nodes in the design.

When the Quartus II Classic Timing Analyzer is enabled, the slow-corner (worst case) timing models are used by default when generating the SDO file. To generate the SDO file using the fast-corner (best case) timing models, perform the following steps:

1. In the Quartus II software, on the Processing menu, point to **Start** and click **Start Classic Timing Analyzer (Fast Timing Model)**.
2. After the fast-corner timing analysis is complete, on the Processing menu, point to **Start** and click **Start EDA Netlist Writer** to create a <*revision_name*>**_v_fast.sdo** or <*revision_name*>**_vhd_fast.sdo** file, which contains the best-case delay values for each timing path.

If you are running a best-case timing analysis, the Quartus II software generates a Tcl script similar to the following: <*revision_name*>**_pt_v_fast.tcl**.

When TimeQuest is run with the fast-corner netlist or when the **Optimize fast-corner timing** check box is selected in the **Fitter Settings** dialog box, the fast-corner SDC file is generated.

After the EDA Netlist Writer has finished, two SDO files are created: <*revision_name*>**_v.sdo** (slow-corner) or <*revision_name*>**_v_fast.sdo** (fast-corner).

Generating Multiple Operating Conditions with TimeQuest

Different operating conditions can be specified to the EDA Netlist Writer for PrimeTime analysis. The different operating conditions are reflected in the **.sdo** file generated by the EDA Netlist Writer.

Table 11–1 shows the available operating conditions that can be set for a few of Altera's device families.

Table 11–1. Available Operating Condition Combinations

Device Family	Available Conditions (Model, Voltage, Temperature)
Stratix® III	(slow, 1100 mV, 85° C), (slow, 1100 mV, 0° C), (fast, 1100 mV, 0° C)
Cyclone® III	(slow, 1200 mV, 85° C), (slow, 1200 mV, 0° C), (fast, 1200 mV, 0° C)
Stratix II	(slow, N/A, N/A), (fast, N/A, N/A)
Cyclone II	(slow, N/A, N/A), (fast, N/A, N/A)

From the TimeQuest Console pane, use the command `get_available_operating_conditions` to obtain a list of available operating conditions for the target device.

The following steps show how to generate the **.sdo** files for the three different operating conditions for a Stratix III design. Each command must be entered at the command prompt.

The `--tq2pt` option for `quartus_sta` is required only if the project does not specify that the PrimeTime tool will be used as the timing analysis tool.

1. Generate the first slow corner model at the operating conditions: slow, 1100 mV, and 85° C.

```
quartus_sta --model=slow --voltage=1100 --temperature=85
<project name>
```

2. Generate the fast corner model at the operating conditions: fast, 1100 mV, and 0° C.

```
quartus_sta --model=fast --voltage=1100 --temperature=0
--tq2pt <project name>
```

3. Generate the PrimeTime output files for the corners specified above. The output files will be generated in the **primetime_two_corner_files** directory.

```
quartus_eda --timing_analysis --tool=primetime
--format=verilog
--output_directory=primetime_two_corner_files
--write_settings_files=off <project name>
```

4. Generate the second slow corner model at the operating conditions: slow, 1100 mV, and 0° C.

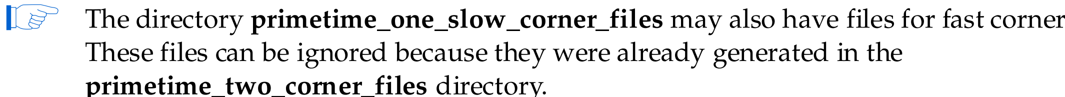
```
quartus_sta --model=slow --voltage=1100 --temperature=0
--tq2pt <project name>
```

5. Generate the PrimeTime output files for the second slow corner. The output files will be generated in the **primetime_one_slow_corner_files** directory.

```
quartus_eda --timing_analysis --tool=primetime
--format=verilog
--output_directory=primetime_one_slow_corner_files
--write_settings_files=off $revision
```

To summarize, the previous steps generate the following files for the three operating conditions:

- First slow corner (slow, 1100 mV, 85° C):
  VO File—primetime_two_corner_files/<*project name*>**.vo**
  SDO File—primetime_two_corner_files/<*project name*>**_v.sdo**
- Fast corner (fast, 1100 mV, 0° C):
  VO File—primetime_two_corner_files/<*project name*>**.vo**
  SDO File—primetime_two_corner_files/<*project name*>**_v_fast.sdo**
- Second slow corner (slow, 1100 mV, 0° C):
  VO File—primetime_one_slow_corner_files/<*project name*>**.vo**
  SDO File—primetime_one_slow_corner_files/<*project name*>**_v.sdo**



The directory **primetime_one_slow_corner_files** may also have files for fast corner. These files can be ignored because they were already generated in the **primetime_two_corner_files** directory.

The Tcl Script

The Tcl script generated by the Quartus II software contains information required by the PrimeTime software to analyze the timing and set up your post-fit design. This script specifies the search path and the names of the PrimeTime database library files provided with the Quartus II software. The `search_path` and `link_path` variables are defined at the beginning of the Tcl file. The `link_path` variable is a space-delimited list that contains the names of all database files used by the PrimeTime software.

Depending on whether you selected **Verilog** or **VHDL** in the **Format for output netlist** list on the **Timing Analysis** page of the **Settings** dialog box, when the Quartus II Classic Timing Analyzer is enabled, the EDA Netlist Writer generates and saves the script as either <*revision_name*>**_pt_v.tcl** or <*revision_name*>**_pt_vhd.tcl**.

To access the **EDA Settings** dialog box, on the Assignments menu, click **EDA Tool Settings**, then expand **EDA Tool Settings** under the **Category** list. In the dialog box, you can specify VHDL or Verilog for the format for the output netlist.

The script also directs the PrimeTime software to use the <*device family*>**_all_pt.v** or <*device family*>**_all_pt.vhd** file, which contains the Verilog or VHDL description of library cells for the targeted device family.

Example 11–1 shows the `search_path` and `link_path` variables defined in the Tcl script:

Example 11–1. Sample PrimeTime Setup Script

```
set quartus_root "altera/quartus/"
set search_path [list . [format "%s%s" $quartus_root "eda/synopsys/primetime/lib"]  ]

set link_path [list *  stratixii_lcell_comb_lib.db  stratixii_lcell_ff_lib.db
stratixii_asynch_io_lib.db stratixii_io_register_lib.db stratixii_termination_lib.db
bb2_lib.db  stratixii_ram_internal_lib.db  stratixii_memory_register_lib.db
stratixii_memory_addr_register_lib.db  stratixii_mac_out_internal_lib.db
stratixii_mac_mult_internal_lib.db  stratixii_mac_register_lib.db
stratixii_lvds_receiver_lib.db stratixii_lvds_transmitter_lib.db
stratixii_asmiblock_lib.db stratixii_crcblock_lib.db  stratixii_jtag_lib.db
stratixii_rublock_lib.db  stratixii_pll_lib.db  stratixii_dll_lib.db alt_vtl.db]

read_vhdl  -vhdl_compiler  stratixii_all_pt.vhd
```

The EDA Netlist Writer converts any Quartus II Classic Timing Analyzer timing assignments to the PrimeTime software constraints and exceptions when it generates the PrimeTime files. The converted constraints are saved to the Tcl script. The Tcl script also includes a PrimeTime software command that reads the Standard Delay Format Output (**.sdo**) file generated by the Quartus II software. You can place additional commands in the Tcl script to analyze or report on timing paths.

Table 11–2 shows some examples of timing assignments converted by the Quartus II software for the PrimeTime software. For example, the `set_input_delay -max` command sets the input delay on an input pin.

Table 11–2. Equivalent Quartus II and PrimeTime Software Constraints

Quartus II Equivalent	PrimeTime Constraint
Clock defined on input pin, clock of 10 ns period and 50% duty cycle	`create_clock -period 10.000 -waveform {0 5.000} \` [get_ports clk] -name clk
Input maximum delay of 1 ns on input pin, din	`set_input_delay -max -add_delay 1.000 -clock \ [get_clocks clk] [get_ports din]`
Input minimum delay of 1 ns on input pin, din	`set_input_delay -min -add_delay 1.000 -clock \ [get_clocks clk] [get_ports din]`
Output maximum delay of 3 ns on output pin, out	`set_output_delay -max -add_delay 3.000 -clock \ [get_clocks clk] [get_ports out]`

When the Quartus II TimeQuest Timing Analyzer is turned on, the EDA Netlist Writer generates and saves the script as <*revision_name*>**.pt.tcl**.

The EDA Netlist Writer converts all Quartus II TimeQuest Timing Analyzer SDC constraints and exceptions into compatible PrimeTime software constraints and exceptions when it generates the PrimeTime files. The constraints and exceptions are saved to the <*revision_name*>**.constraints.sdc** file.

Generated File Summary

The files that are generated by the EDA Netlist Writer for the PrimeTime software depend on the Quartus II timing analysis tool you selected.

Table 11–3 shows the files that are generated for the PrimeTime software when the Quartus II Classic Timing Analyzer is selected.

Table 11–3. Quartus II Classic Timing Analyzer-Generated PrimeTime Files

File	Description
<*revision_name*>**.vho** | <*revision_name*>**.vo**	The PrimeTime software output netlist. Either a VHDL Output file or a Verilog Output file is generated, depending on the output netlist language set.
<*revision_name*>**_vhd.sdo** | <*revision_name*>**_v.sdo**	The PrimeTime software standard delay file. Either a VHDL Standard Delay Output file or a Verilog Standard Delay Output file is generated, depending on the output netlist language set.
<*revision_name*>**_pt_vhd.tcl** | <*revision_name*>**_pt_v.tcl**	PrimeTime setup and constraint script. Either a VHDL Tcl script or a Verilog Tcl script is generated, depending on the output netlist language set.

Table 11–4 shows the files that are generated for the PrimeTime software when the Quartus II TimeQuest Timing Analyzer is selected. The EDA Netlist Writer supports the output netlist format only when the TimeQuest Timing Analyzer is enabled.

Table 11–4. Quartus II TimeQuest Timing Analyzer-Generated PrimeTime Files

File	Description
<*revision_name*>**.vo**	The PrimeTime software output netlist. When the Quartus II TimeQuest Timing Analyzer is enabled, only PrimeTime (Verilog) is supported.
<*revision_name*>**_v.sdo** | <*revision_name*>**_v_fast.sdo**	The PrimeTime software standard delay file. When the Quartus II TimeQuest Timing Analyzer is enabled, only PrimeTime (Verilog) is supported.
<*revision_name*>**.pt.tcl**	PrimeTime setup and constraint script. When the Quartus II TimeQuest Timing Analyzer is enabled, only PrimeTime (Verilog) is supported.
<*revision_name*>**.collections.sdc**	Contains the mapping from the Quartus II TimeQuest Timing Analyzer netlist to the PrimeTime netlist.
<*revision_name*>**.constraints.sdc**	Contains the converted Quartus II TimeQuest Timing Analyzer constraints for the PrimeTime software.

Running the PrimeTime Software

The PrimeTime software runs only on UNIX operating systems. If the Quartus II output files for the PrimeTime software were generated by running the Quartus II software on a PC/Windows-based system, follow these steps to run the PrimeTime software using Quartus II output files:

1. Install the PrimeTime libraries on a UNIX system by installing the Quartus II software on UNIX.

   The PrimeTime libraries are located in the **<*Quartus II installation directory*>/eda/synopsys/primetime/lib** directory.

2. Copy the Quartus II output files to the appropriate UNIX directory. You may need to run a PC to UNIX program, such as `dos2unix`, to remove any control characters.

3. Modify the Quartus II path in Tcl scripts to point to the PrimeTime libraries, as described in Step 1. In Example 11–1, the first line is:

```
set quartus_root "c:/altera/quartus51/" set search_path [list . [format
"%s%s" $quartus_root "eda/synopsys/primetime/lib"] ]
```

This is the Tcl script that should be modified.

Analyzing Quartus II Projects

The PrimeTime software is controlled with Tcl scripts and can be run through `pt_shell`. You can run the <*revision_name*>**_pt_v.tcl** script file. For example, type the following at a UNIX system command prompt:

```
pt_shell -f <revision_name>_pt_v.tcl ↵
```

When the Quartus II TimeQuest Timing Analyzer is selected, type the following at a UNIX system command prompt:

```
pt_shell -f <revision_name>.pt.tcl ↵
```

After all Tcl commands in the script are interpreted, the PrimeTime software returns control to the `pt_shell` prompt, which allows you to use other commands.

Other pt_shell Commands

You can run additional `pt_shell` commands at the `pt_shell` prompt, including the `man` program. For example, to read documentation about the `report_timing` command, type the following at the `pt_shell` prompt:

```
man report_timing ↵
```

You can list all commands available in `pt_shell` by typing the following at the `pt_shell` prompt:

```
help ↵
```

Type `quit` ↵ at the `pt_shell` prompt to close `pt_shell`.

You can also run `pt_shell` without a script file by typing `pt_shell`↵ at the UNIX command line prompt.

PrimeTime Timing Reports

This section describes PrimeTime timing reports.

Sample of the PrimeTime Software Timing Report

After running the script, the PrimeTime software generates a timing report. If the timing constraints are not met, `Violated` is displayed at the end of the timing report. The timing report also gives the negative slack.

The PrimeTime software report is similar to the sample shown in Example 11–2. The starting point in this report is a register clocked by clock signal, `clock`, the endpoint is another register, `inst3-I.lereg`.

Example 11–2. Hold Path Report in PrimeTime

```
Startpoint: inst2~I.lereg
   (rising edge-triggered flip-flop clocked by clock)
Endpoint: inst3~I.lereg
   (rising edge-triggered flip-flop clocked by clock)
Path Group: clock
Path Type: min
Point                                                Incr     Path
-----------------------------------------------------------------
clock clock (rise edge)                              0.000    0.000
clock network delay (propagated)                     3.166    3.166
inst2~I.lereg.clk (stratix_lcell_register)           0.000    3.166r
inst2~I.lereg.regout (stratix_lcell_register) <- 0.176*   3.342r
inst2~I.regout (stratix_lcell)                       0.000*   3.342r
inst3~I.datac (stratix_lcell)                        0.000*   3.342r
inst3~I.lereg.datac (stratix_lcell_register)         3.413*   6.755r
data arrival time                                             6.755
clock clock (rise edge)                              0.000    0.000
clock network delay (propagated)                     3.002    3.002
inst3~I.lereg.clk (stratix_lcell_register)                    3.002r
library hold time                                    0.100*   3.102
data required time                                            3.102
----------------------------------------------------------------
data required time                                            3.102
data arrival time                                            -6.755
----------------------------------------------------------------
slack (MET)                                                   3.653
```

Comparing Timing Reports from the Quartus II Classic Timing Analyzer and the PrimeTime Software

Both the Quartus II Classic Timing Analyzer and the Quartus II TimeQuest Timing Analyzer generate a static timing analysis report for every successful design compilation. The timing report lists all of the timing paths in your design that were analyzed, and indicates whether these paths have met or violated their timing requirements. Violations are reported only if timing constraints were specified.

The Quartus II TimeQuest Timing Analyzer uses an equivalent set of equations as PrimeTime when reporting the static timing analysis result for a design. However, the Quartus II Classic Timing Analyzer uses slightly different reporting equations when reporting the static timing analysis results for a design. This section describes these differences between the Quartus II Classic Timing Analyzer and the PrimeTime software.

The timing report generated by the Quartus II Classic Timing Analyzer differs from the report generated by the PrimeTime software. Both tools provide the same data but present in different formats. The following sections show how the PrimeTime software reports the following slack values differently from the Quartus II Classic Timing Analyzer report:

- "Clock Setup Relationship and Slack" on page 11–10
- "Clock Hold Relationship and Slack" on page 11–13
- "Input Delay and Output Delay Relationships and Slack" on page 11–16

Clock Setup Relationship and Slack

The Quartus II Classic Timing Analyzer performs a setup check that ensures that the data launched by source registers is latched correctly at the destination registers. The Quartus II Classic Timing Analyzer does this by determining the data arrival time and clock arrival time at the destination registers, and compares this data with the setup time delay of the destination register. Equation 11–1 expresses the inequality that is used for a setup check. The data arrival time includes the longest path from the clock to the source register, the clock-to-out micro delay of the source register, and the longest path from the source register to the destination register. The clock arrival time is the shortest delay from the clock to the destination register.

Equation 11–1.

$$\text{Clock Arrival} - \text{Data Arrival} \geq t_{su}$$

Slack is the margin by which a timing requirement is met or not met. Positive slack indicates the margin by which a requirement is met. Negative slack indicates the margin by which a requirement was not met. The Quartus II Classic Timing Analyzer determines the clock setup slack, as shown in Equation 11–2:

Equation 11–2.

$$\text{Clock Setup Slack} = \text{Largest Register-to-Register Requirement} - \text{Longest Register-to-Register Delay}$$

The longest register-to-register delay in the previous equation is equal to the register-to-register data delay.

Equation 11–3.

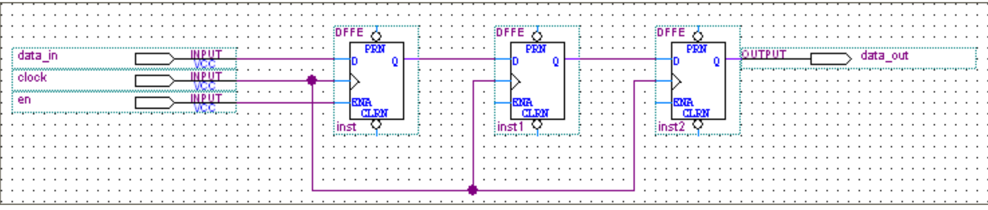
$$\text{Largest Register-to-Register Requirement} = \text{Setup Relationship between Source and Destination} + \text{Largest Clock Skew} - \text{Micro } t_{co} \text{ of Destination Register} - \text{Micro } t_{su} \text{ of Destination Register}$$

$$\text{Setup Relationship between Source and Destination} = \text{Latch Edge} - \text{Launch Edge}$$

$$\text{Clock Skew} = \text{Shortest Clock Path to Destination} - \text{Longest Clock Path to Source}$$

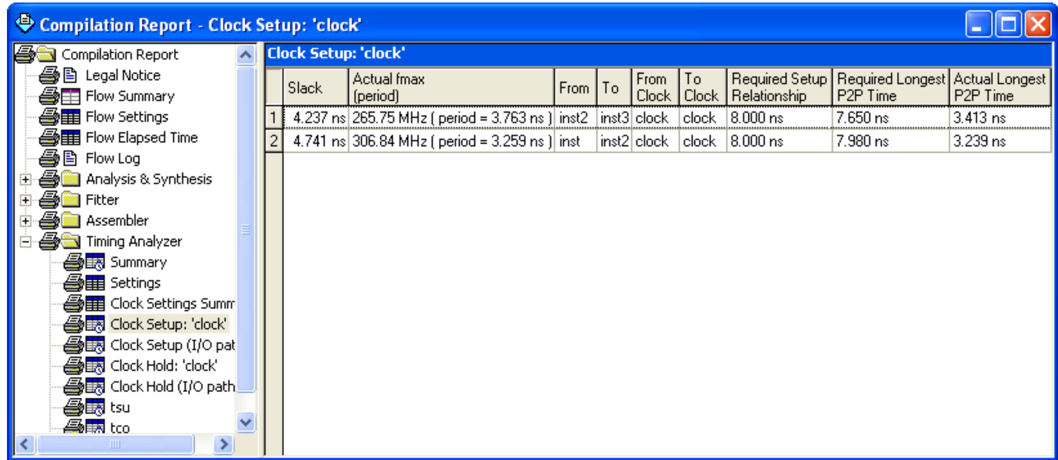
For a simple three-register design, refer to Figure 11–3.

Figure 11–3. Simple Three-Register Design



The Quartus II Classic Timing Analyzer generates a report for the design, as shown in Figure 11–4.

Figure 11–4. Timing Analyzer Report from Figure 11–3



	Slack	Actual fmax (period)	From	To	From Clock	To Clock	Required Setup Relationship	Required Longest P2P Time	Actual Longest P2P Time
1	4.237 ns	265.75 MHz (period = 3.763 ns)	inst2	inst3	clock	clock	8.000 ns	7.650 ns	3.413 ns
2	4.741 ns	306.84 MHz (period = 3.259 ns)	inst	inst2	clock	clock	8.000 ns	7.980 ns	3.239 ns

Equation 11–1, Equation 11–2, and Equation 11–3 are similar to those found in other static timing analysis tools, such as the PrimeTime software. Equation 11–4 to Equation 11–7, used by the PrimeTime software, are essentially the same as those used by the Quartus II Classic Timing Analyzer, but they are rearranged.

Equation 11–4.

$$\text{Slack} = \text{Data Required} - \text{Data Arrival}$$

Equation 11–5.

$$\text{Clock Arrival} = \text{Latch Edge} + \text{Shortest Clock Path to Destination}$$

Equation 11–6.

$$\text{Data Required} = \text{Clock Arrival} - \text{Micro } t_{su}$$

Equation 11–7.

$$\text{Data Arrival} = \text{Launch Edge} + \text{Longest Clock Path to Source} + \text{Micro } t_{co} + \text{Longest Data Delay}$$

The longest data delay in the previous equation is equal to register-to-register data delay.

Figure 11–5 shows a clock setup check in the Quartus II software.

Figure 11–5. Clock Setup Check Reporting with the Quartus II Classic Timing Analyzer

```
Info: Slack time is 4.237 ns for clock "clock" between source register "inst2" and destination register "inst3"
  Info: Fmax is 265.75 MHz (period= 3.763 ns)
  Info: + Largest register to register requirement is 7.650 ns
    Info: + Setup relationship between source and destination is 8.000 ns
      Info: + Latch edge is 8.000 ns
      Info: - Launch edge is 0.000 ns
      Info: - Clock setup uncertainty between source and destination is 0.000 ns
    Info: + Largest clock skew is -0.164 ns
      Info: + Shortest clock path from clock "clock" to destination register is 3.002 ns
      Info: - Longest clock path from clock "clock" to source register is 3.166 ns
    Info: - Micro clock to output delay of source is 0.176 ns
    Info: - Micro setup delay of destination is 0.010 ns
  Info: - Longest register to register delay is 3.413 ns
```

The results in Equation 11–8 are obtained by extracting the numbers from the Quartus II Classic Timing Analyzer report and applying them to the clock setup slack equations from the Quartus II Classic Timing Analyzer:

Equation 11–8.

$$\text{Setup Relationship between Source and Destination} = \text{Latch Edge} - \text{Launch Edge} - \text{Clock Setup Uncertainty}$$

$$8.0 - 0.0 - 0.0 = 8.0\text{ns}$$

$$\text{Clock Skew} = \text{Shortest Clock Path to Destination} - \text{Longest Clock Path to Source}$$

$$3.002 - 3.166 = -0.164\text{ns}$$

$$\text{Largest Register-to-Register Requirement} = \text{Setup Relationship between Source \& Destination} + \text{Largest Clock Skew} - \text{Micro } t_{co} \text{ of Source Register} - \text{Micro } t_{su} \text{ of Destination Register}$$

$$8 + (-0.164) - 0.176 - 0.010 = 7.650\text{ns}$$

$$\text{Clock Setup Slack} = \text{Largest Register-to-Register Requirement} - \text{Longest Register-to-Register Delay}$$

$$7.650 - 3.413 = 4.237\text{ns}$$

For the same register-to-register path, the PrimeTime software generates a clock setup report as shown in Example 11–3:

Example 11–3. Setup Path Report in PrimeTime

```
Startpoint: inst2~I.lereg
   (rising edge-triggered flip-flop clocked by clock)
Endpoint: inst3~I.lereg
   (rising edge-triggered flip-flop clocked by clock)
Path Group: clock
Path Type: max
Point                                                Incr     Path
-----------------------------------------------------------------
clock clock (rise edge)                              0.000    0.000
clock network delay (propagated)                     3.166    3.166
inst2~I.lereg.clk (stratix_lcell_register)           0.000    3.166r
inst2~I.lereg.regout (stratix_lcell_register) <- 0.176*   3.342r
inst2~I.regout (stratix_lcell) <-                    0.000*   3.342r
inst3~I.datac (stratix_lcell) <-                     0.000*   3.342r
inst3~I.lereg.datac (stratix_lcell_register)         3.413*   6.755r
data arrival time                                             6.755
clock clock (rise edge)                              8.000    8.000
clock network delay (propagated)                     3.002   11.002
inst3~I.lereg.clk (stratix_lcell_register                    11.002r
library setup time                                  -0.010*  10.992
data required time                                           10.992
-----------------------------------------------------------------
data required time                                           10.992
data arrival time                                            -6.755
-----------------------------------------------------------------
slack (MET)                                                   4.237
```

Clock Hold Relationship and Slack

The Quartus II Classic Timing Analyzer performs a hold time check along every register-to-register path in the design to ensure that no hold time violations have occurred. The hold time check verifies that data from the source register does not reach the destination until after the hold time of the destination register. The condition used for a hold check is shown in Equation 11–9:

Equation 11–9.

$$\text{Data Arrival} - \text{Clock Arrival} \geq t_H$$

The Quartus II Classic Timing Analyzer determines the clock hold slack with Equation 11–10, Equation 11–11, Equation 11–12, and Equation 11–13:

Equation 11–10.

$$\text{Clock Hold Slack} = \text{Shortest Register-to-Register Delay} - \text{Smallest Register-to-Register Requirement}$$

Equation 11–11.

$$\text{Smallest Register-to-Register Requirement} = \text{Hold Relationship between Source \& Destination} + \text{Smallest Clock Skew} - \text{Micro } t_{su} \text{ of Source} + \text{Micro } t_H \text{ of Destination}$$

Equation 11–12.

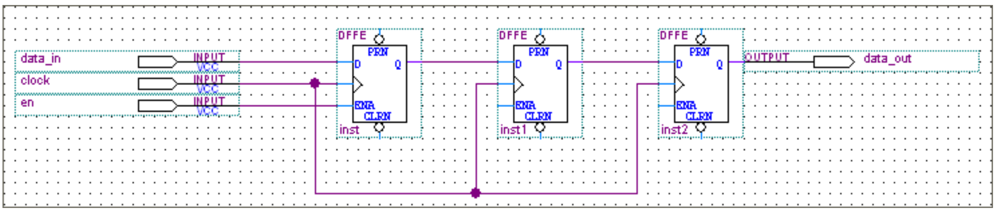
Hold Relationship between Source & Destination = Latch Edge – Launch Edge

Equation 11–13.

Smallest Clock Skew = Longest Clock Path from Clock to Destination Register – Shortest Clock Path from Clock to Source Register

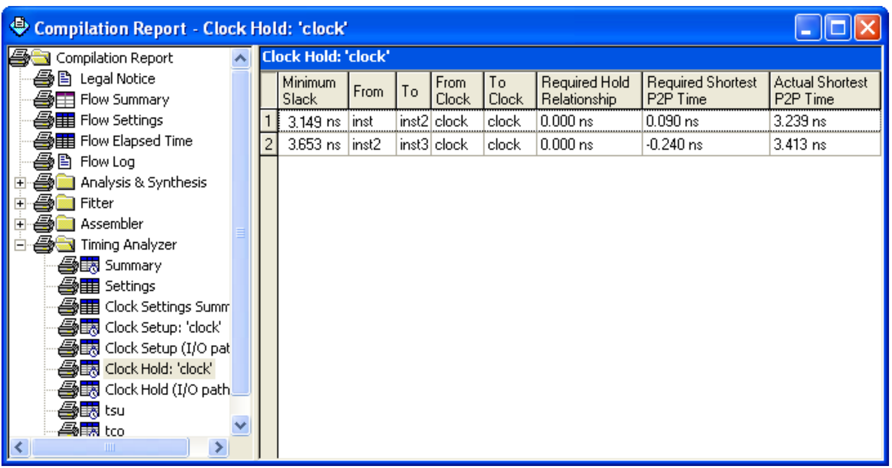
Figure 11–6 shows a simple three-register design.

Figure 11–6. A Simple Three-Register Design



The Quartus II Classic Timing Analyzer generates a report as shown in Figure 11–7.

Figure 11–7. Timing Analyzer Report Generated from the Three-Register Design



	Minimum Slack	From	To	From Clock	To Clock	Required Hold Relationship	Required Shortest P2P Time	Actual Shortest P2P Time
1	3.149 ns	inst	inst2	clock	clock	0.000 ns	0.090 ns	3.239 ns
2	3.653 ns	inst2	inst3	clock	clock	0.000 ns	-0.240 ns	3.413 ns

The previous equations are similar to those found in the Quartus II software. The following equations are the same equations that are used by the PrimeTime software, but they are rearranged.

Equation 11–14.

Slack = Data Required – Data Arrival

Equation 11–15.

Clock Arrival = Latch Edge + Longest Clock Path to Destination

© March 2009 Altera Corporation

Equation 11–16.

$$\text{Data Required} = \text{Clock Arrival} - \text{Micro } t_H$$

Equation 11–17.

$$\text{Data Arrival} = \text{Launch Edge} + \text{Longest Clock Path to Source} + \text{Micro } t_{co} + \text{Shortest Data Delay}$$

The shortest register-to-register delay in the previous equation is equal to register-to-register data delay.

Figure 11–8 shows a clock setup check with the Quartus II Classic Timing Analyzer.

Figure 11–8. Clock Hold Check Reporting with the Quartus II Classic Timing Analyzer

```
Info: Minimum slack time is 3.653 ns for clock "clock" between source register "inst2" and destination register "inst3"
  Info: + Shortest register to register delay is 3.413 ns
  Info: - Smallest register to register requirement is -0.240 ns
    Info: + Hold relationship between source and destination is 0.000 ns
      Info: + Latch edge is 0.000 ns
      Info: - Launch edge is 0.000 ns
      Info: - Clock hold uncertainty between source and destination is 0.000 ns
    Info: + Smallest clock skew is -0.164 ns
      Info: + Longest clock path from clock "clock" to destination register is 3.002 ns
      Info: - Shortest clock path from clock "clock" to source register is 3.166 ns
    Info: - Micro clock to output delay of source is 0.176 ns
    Info: + Micro hold delay of destination is 0.100 ns
```

The following results are obtained by extracting the numbers from the Timing Analysis report and applying the clock setup slack equations from the Quartus II Classic Timing Analyzer.

Equation 11–18.

$$\text{Clock Hold Slack} = \text{Shortest Register-to-Register Delay} - \text{Smallest Register-to-Register Requirement}$$

$$3.413 - (-0.240) = 3.653\text{ns}$$

$$\text{Smallest Register-to-Register Requirement} = \text{Hold Relationship between Source \& Destination} + \text{Smallest Clock Skew} - \text{Micro } t_{co} \text{ of Source} + \text{Micro } t_H \text{ of Destination}$$

$$0 + (-0.164) - 0.176 + 0.100 = -0.240\text{ns}$$

$$\text{Hold Relationship between Source \& Destination} = \text{Latch} - \text{Launch}$$

$$0.0 - 0.0\text{ns}$$

$$\text{Smallest Clock Skew} = \text{Longest Clock Path from Clock to Destination Register} - \text{Shortest Clock Path from Clock to Source Register}$$

$$3.002 - 3.166 = -0.164\text{ns}$$

For the same register-to-register path, the PrimeTime software generates the report shown in Example 11–4:

Example 11–4. Hold Path Report in PrimeTime

```
Startpoint: inst2~I.lereg
   (rising edge-triggered flip-flop clocked by clock)
Endpoint: inst3~I.lereg
   (rising edge-triggered flip-flop clocked by clock)
Path Group: clock
Path Type: min
Point                                                Incr     Path
-----------------------------------------------------------------
clock clock (rise edge)                              0.000    0.000
clock network delay (propagated)                     3.166    3.166
inst2~I.lereg.clk (stratix_lcell_register)           0.000    3.166r
inst2~I.lereg.regout (stratix_lcell_register)<-       0.176*   3.342r
inst2~I.regout (stratix_lcell)                       0.000*   3.342r
inst3~I.datac (stratix_lcell)                        0.000*   3.342r
inst3~I.lereg.datac (stratix_lcell_register)         3.413*   6.755r
data arrival time                                             6.755

clock clock (rise edge)                              0.000    0.000
clock network delay (propagated)                     3.002    3.002
inst3~I.lereg.clk (stratix_lcell_register)                    3.002r
library hold time                                    0.100*   3.102
data required time                                            3.102
---------------------------------------------------------------
data required time                                            3.102
data arrival time                                            -6.755
---------------------------------------------------------------
slack (MET)                                                   3.653
```

Both sets of hold slack equations can be used to determine the hold slack value of any path.

Input Delay and Output Delay Relationships and Slack

Input delay and output delay reports generated by the Quartus II Classic Timing Analyzer are similar to the clock setup and clock hold relationship reports. Figure 11–9 shows the input delay and output delay report for the design shown in Figure 11–6 on page 11–14.

Figure 11–9. Input and Output Delay Reporting with the Quartus II Classic Timing Analyzer

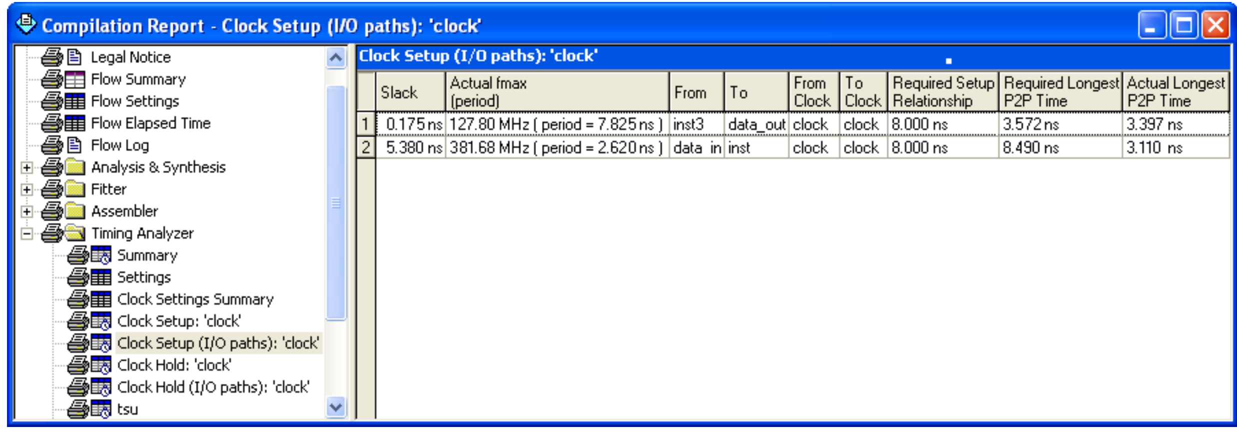


Figure 11–10 shows the fully expanded view for the output delay path.

Figure 11–10. Output Delay Path Reporting with the Quartus II Classic Timing Analyzer

Info: Slack time is 175 ps for clock "clock" between source register "inst3" and destination pin "data_out"
Info: Fmax is 127.8 MHz (period= 7.825 ns)
Info: + Largest register to pin requirement is 3.572 ns
Info: + Setup relationship between source and destination is 8.000 ns
Info: + Latch edge is 8.000 ns
Info: - Launch edge is 0.000 ns
Info: - Clock setup uncertainty between source and destination is 0.000 ns
Info: - Longest clock path from clock "clock" to source register is 3.002 ns
Info: - Micro clock to output delay of source is 0.176 ns
Info: - Max Output delay of pin is 1.25 ns
Info: - Longest register to pin delay is 3.397 ns

For the same output delay path, the PrimeTime software generates a report similar to Example 11–5:

Example 11–5. Setup Path Report in PrimeTime

```
Startpoint: inst3~I.lereg
  (rising edge-triggered flip-flop clocked by clock)
Endpoint: data_out
  (output port clocked by clock)
Path Group: clock
Path Type: max
Point                                                Incr     Path
-----------------------------------------------------------------
clock clock (rise edge)                              0.000    0.000
clock network delay (propagated)                     3.002    3.002
inst3~I.lereg.clk (stratix_lcell_register)           0.000    3.002r
inst3~I.lereg.regout (stratix_lcell_register)<-      0.176*   3.178r
inst3~I.regout (stratix_lcell)<-                     0.000    3.178r
data_out~I.datain (stratix_io)<-                     0.000    3.178r
data_out~I.out_mux3.A (mux21)<-                      0.000    3.178r
data_out~I.out_mux3.MO (mux21)<-                     0.000    3.178r
data_out~I.and2_22.IN1 (AND2)<-                      0.000    3.178r
data_out~I.and2_22.Y (AND2)<-                        0.000    3.178r
data_out~I.out_mux1.A (mux21)<-                      0.000    3.178r
data_out~I.out_mux1.MO (mux21)<-                     0.000    3.178r
data_out~I.inst1.datain (stratix_asynch_io)<-        0.902*   4.080r
data_out~I.inst1.padio (stratix_asynch_io)<-         2.495*   6.575r
data_out~I.padio (stratix_io)<-                      0.000    6.575r
data_out (out)                                       0.000    6.575r
data arrival time                                             6.575
clock clock (rise edge)                              8.000    8.000
clock network delay (propagated)                     0.000    8.000
output external delay                                1.250    6.750
data required time                                            6.750
----------------------------------------------------------------
data required time                                            6.750
data arrival time                                             6.575
----------------------------------------------------------------
slack (MET)                                                   0.175
```

To generate a list of the 100 worst paths and place this data into a file called **file.timing**, type the following command at the `pt_shell` prompt:

```
report_timing -nworst 100 > file.timing ↵
```

Timing paths in the PrimeTime software are listed in the order of most-negative-slack to most-positive-slack. The PrimeTime software does not categorize failing paths by default. Timing setup (tsu) and timing hold (th) times are not listed separately. In the PrimeTime software, each path is shown with a start and end point; for example, if it is a register-to-register or input-to-register type of path. If you only use the `report_timing` part of the command without adding a `-delay` option, only the `setup-time-related` timing paths are reported.

The following command is used to create a minimum timing report or a list of hold-time-related violations:

```
report_timing -delay_type min ↵
```

Ensure that the correct SDO file, either minimum or maximum delays, is loaded before running this command.

Static Timing Analyzer Differences

Under certain design conditions, several static timing analysis differences can exist between the Classic Timing Analyzer and the TimeQuest Timing Analyzer, and the PrimeTime software. The following sections explain the differences between the two static timing analysis engines and the PrimeTime software.

The Quartus II Classic Timing Analyzer and the PrimeTime Software

The following section describes the differences between the Quartus II Classic Timing Analyzer and the PrimeTime software.

Rise/Fall Support

The Quartus II Classic Timing Analyzer does not support rise/fall analysis. However, rise/fall support is available in PrimeTime.

Minimum and Maximum Delays

TimeQuest calculates minimum and maximum delays for all device components with the exception of clock routing. PrimeTime does not model these delays. This can result in different slacks for a given path on average by 2–3%.

Recovery/Removal Analysis

TimeQuest performs a more pessimistic recovery/removal analysis for asynchronous path than PrimeTime. This can result in different delays reported between the two tools.

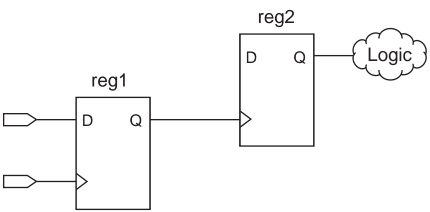
Encrypted Intellectual Property Blocks

The Quartus II software has the capability to decrypt all intellectual property (IP) blocks designed for Altera® devices that have been encrypted by their vendors. The decryption process allows the Quartus II software to perform a full compilation of the design that contains an encrypted IP block. This also allows the Quartus II Classic Timing Analyzer to perform a complete static timing analysis on the design. However, when the PrimeTime software is designated as the static timing analysis tool, the Quartus II EDA Netlist Writer does not generate either a VHDL Output File (**.vho**) or Verilog Output File (**.vo**) netlist file for designs that contain encrypted IP blocks for which the license does not permit generation of output netlists for third-party tools.

Registered Clock Signals

Registered clock signals are clock signals that pass through a register before reaching the clock port of a sequential element. Figure 11–11 shows an example of a registered clock signal.

Figure 11–11. Registered Clock Signal



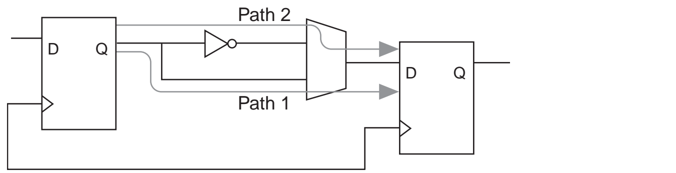
If no clock setting is applied to the register on the clock path (shown as register `reg_1` in Figure 11–11), the Quartus II Classic Timing Analyzer treats the register in the clock path as a buffer. The delay of the buffer is equal to the CELL delay of the register plus the t_{CO} of the register. The PrimeTime software does not treat the register as a buffer.

For more information about creating clock settings, refer to the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Multiple Source and Destination Register Pairs

In any design, multiple paths may exist from a source register to a destination register. Each path from the source register to the destination register may have a different delay value due to the different routes taken. For example, Figure 11–12 shows a sample design that contains multiple path pairs between the source register and destination register.

Figure 11–12. Multiple Source and Destination Pairs



The Quartus II Classic Timing Analyzer analyzes all source and destination pairs, but reports only the source and destination register pair with the worst slack. For example, if the Path 2 pair delay is greater than the Path 1 pair delay in Figure 11–12, the Quartus II Classic Timing Analyzer reports the slack value of the Path 2 pair and not the Path 1 pair. The PrimeTime software reports all possible source and destination register pairs.

Latches

By default, the Quartus II software implements all latches as combinational loops. The Quartus II Classic Timing Analyzer can analyze such latches by treating them as registers with inverted clocks or analyze latches as a combinational loop modeled as a combinational delay.

For more information about latch analysis, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

The PrimeTime software always analyzes these latches as combinational loops, as defined in the netlist file.

LVDS I/O

When it analyzes the dedicated LVDS transceivers in your design, the Quartus II Classic Timing Analyzer generates the Receiver Skew Margin (RSKM) report and a Channel-to-Channel Skew (TCCS) report. The PrimeTime software does not generate these reports.

Clock Latency

When a single clock signal feeds both the source and destination registers of a register-to-register path, and either an Early Clock Latency or a Late Clock Latency assignment has been applied to the clock signal, the Quartus II Classic Timing Analyzer does not factor in the clock latency values when it calculates the clock skew between the two registers. The Quartus II Classic Timing Analyzer factors in the clock latency values when the clock signal to the source and destination registers of a register-to-register path are different. The PrimeTime software applies the clock latency values when a single clock signal or different clock signals feeds the source and destination registers of a register-to-register path.

Input and Output Delay Assignments

When a purely combinational (non-registered) path exists between an input pin and output pin of the Altera FPGA and both pins have been constrained with an input delay and an output delay assignment applied, respectively, the Quartus II Classic Timing Analyzer does not perform a clock setup or clock hold analysis. The PrimeTime software analyzes these paths.

Generated Clocks Derived from Generated Clocks

The Quartus II Classic Timing Analyzer does not support a generated clock derived from a generated clock. This situation might occur if a generated clock feeds the input clock pin of a PLL. The output clock of the PLL is a generated clock.

The Quartus II TimeQuest Timing Analyzer and the PrimeTime Software

The following sections describe the static timing analysis differences between the Quartus II TimeQuest Timing Analyzer and the PrimeTime software.

Encrypted Intellectual Property Blocks

The Quartus II software has the capability to decrypt all IP blocks, designed for Altera devices that have been encrypted by their vendors. The decryption process allows the Quartus II software to perform a full compilation on the design containing an encrypted IP block. This also allows the Quartus II TimeQuest Timing Analyzer to perform a complete static timing analysis on the design. However, when the PrimeTime software is designated as the static timing analysis tool, the Quartus II EDA Netlist Writer does not generate **.vho** or **.vo** netlist files for designs that contain encrypted IP blocks whose license does not permit generation of output netlists for other tools.

Latches

By default, the Quartus II software implements all latches as combinational loops. The Quartus II TimeQuest Timing Analyzer can analyze such latches by treating them as registers with inverted clocks. The Quartus II TimeQuest Timing Analyzer analyzes latches as a combinational loop modeled as a combinational delay.

For more information about latch analysis, refer to the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

The PrimeTime software always analyzes these latches as combinational loops, as defined in the netlist file.

LVDS I/O

When it analyzes the dedicated LVDS transceivers in your design, the Quartus II TimeQuest Timing Analyzer generates a Receiver Skew Margin (RSKM) report and a Channel-to-Channel Skew (TCCS) report. The PrimeTime software does not generate these reports.

The Quartus II TimeQuest Timing Analyzer SDC File and PrimeTime Compatibility

Because of differences between node naming conventions with the netlist generated by the EDA Netlist Writer and the internal netlist used by the Quartus II software, SDC files generated for the Quartus II software or the Quartus II TimeQuest Timing Analyzer are not compatible with the PrimeTime software.

Run the EDA Netlist Writer to generate a compatible SDC file from the TimeQuest SDC file for the PrimeTime software. After the files <*revision_name*>**.collections.sdc** and <*revision_name*>**.constraints.sdc** have been generated, both files can be read in by the PrimeTime software for compatibility of constraints between the Quartus II TimeQuest Timing Analyzer and the PrimeTime software.

Clock and Data Paths

If a timing path acts both as a clock path (a path that connects to a clock pin with a clock associated to it), and a data path (a path that feeds into the data in port of a register), the Quartus II TimeQuest Timing Analyzer will report the data paths, whereas PrimeTime will not.

Inverting and Non-Inverting Propagation

TimeQuest always propagates non-inverting sense for clocks through non-unate paths in the clock network.

PrimeTime's default behavior is to propagate both inverting and non-inverting senses through a non-unate path in the clock network.

Multiple Rise/Fall Numbers For a Timing Arc

For a given timing path with a corresponding set of pins/ports that make up the path (including source and destination pair), if the individual components of that path have different rise/fall delays, there can potentially be many timing paths with different delays using the same set of pins. If this occurs, TimeQuest reports only one timing path for the set of pins that make up the path.

Virtual Generated Clocks

PrimeTime does not support generated clocks that are virtual. To maintain compatibility between TimeQuest and PrimeTime, all generated clocks should have an explicit target specified.

Generated Clocks Derived from Generated Clocks

The Quartus II Classic Timing Analyzer does not support the creation of a generated clock derived from a generated clock. This situation might occur if a generated clock feeds the input clock pin of another generated clock. The output clock of the PLL is a generated clock.

Conclusion

The Quartus II software can export a netlist, constraints, and timing information for use with the PrimeTime software. The PrimeTime software can use data from either best-case or worst-case Quartus II timing models to measure timing. The PrimeTime software is controlled using a Tcl script generated by the Quartus II software that you can customize to direct the PrimeTime software to produce violation and slack reports.

Referenced Documents

This chapter references the following documents:

- *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* in volume 3 of the *Quartus II Handbook*

- *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*

Document Revision History

Table 11–5 shows the revision history for this chapter.

Table 11–5. Document Revision History

Date and Version	Changes Made	Summary of Changes
March 2009 v9.0.0	This was chapter 10 in version 8.1.	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Updated to Quartus II software version 8.0 and date. ■ Added hyperlinks to referenced Altera documentation throughout the chapter.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

Section III. Power Estimation and Analysis

As FPGA designs grow larger and processes continue to shrink, power becomes an ever-increasing concern. When designing a PCB, the power consumed by a device must be accurately estimated to develop an appropriate power budget, and to design the power supplies, voltage regulators, heat sink, and cooling system.

The Quartus® II software allows you to estimate the power consumed by your current design during timing simulation. The power consumption of your design can be calculated using the Microsoft Excel-based power calculator, or the Simulation-Based Power Estimation features in the Quartus II software. This section explains how to use both.

This section includes the following chapter:

■ Chapter 12, PowerPlay Power Analysis

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

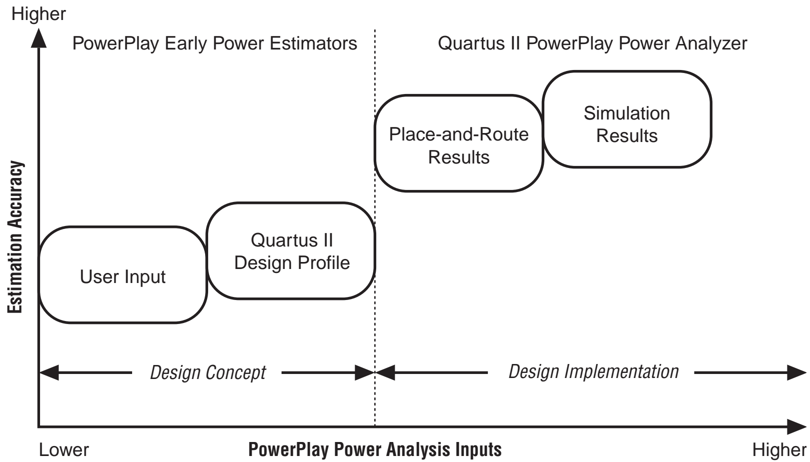
12. PowerPlay Power Analysis

QII53013-9.0.0

Introduction

As designs grow larger and process technology continues to shrink, power becomes an increasingly important design consideration. When designing a PCB, the power consumed by a device must be accurately estimated to develop an appropriate power budget and to design the power supplies, voltage regulators, heat sink, and cooling system. The PowerPlay Power Analysis tools, made available by Altera, provide improved power consumption accuracy and the ability to estimate power consumption from early design concept through design implementation, as shown in Figure 12–1.

Figure 12–1. PowerPlay Power Analysis



Depending where you are in your design cycle and the accuracy of the estimation required, you can either use the PowerPlay Early Power Estimator spreadsheet or the PowerPlay Power Analyzer Tool in the Quartus® II software. You can use the PowerPlay Early Power Estimator spreadsheet during the board design and layout phase to obtain a power estimate and then design for proper power management. The PowerPlay Power Analyzer Tool is used to obtain an accurate estimation of power after the design is complete, ensuring that thermal and supply budgets are not violated.

You can estimate power consumption for Arria® GX, Stratix® series devices, Cyclone® series devices, HardCopy® II, and MAX® II devices with the Microsoft Excel-based PowerPlay Early Power Estimator spreadsheet or the PowerPlay Power Analyzer Tool.

For more information about acquiring the PowerPlay Power Estimator spreadsheet for Arria GX, Stratix series devices, Cyclone series, HardCopy II, and MAX II devices and its use, refer to *PowerPlay Early Power Estimators (EPE) and Power Analyzer* on the Altera® website (www.altera.com).

This chapter discusses the following topics:

- "Quartus II Early Power Estimator File"
- "Types of Power Analyses" on page 12–5
- "Factors Affecting Power Consumption" on page 12–5
- "Using the PowerPlay Power Analyzer" on page 12–17

Quartus II Early Power Estimator File

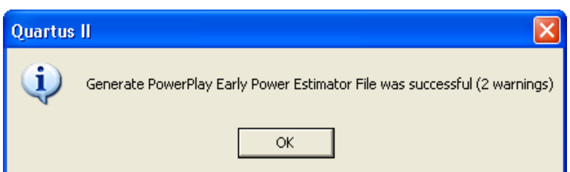
When entering data into the Early Power Estimator spreadsheet, you must enter the device resources, operating frequency, toggle rates, and other parameters. This requires familiarity with the design. If you do not have an existing design, you must estimate the number of device resources used in your design and enter it manually.

If you already have an existing design or a partially completed design, the power estimator file that is generated by the Quartus II software can aid in completing the PowerPlay Early Power Estimator spreadsheet.

To generate the power estimation file, you must first compile your design in the Quartus II software. After compilation is complete, on the Project menu, click **Generate PowerPlay Early Power Estimator File**. This command instructs the Quartus II software to write out a power estimator Comma-Separated Value (**.csv**) file (or a text [**.txt**] file for older device families).

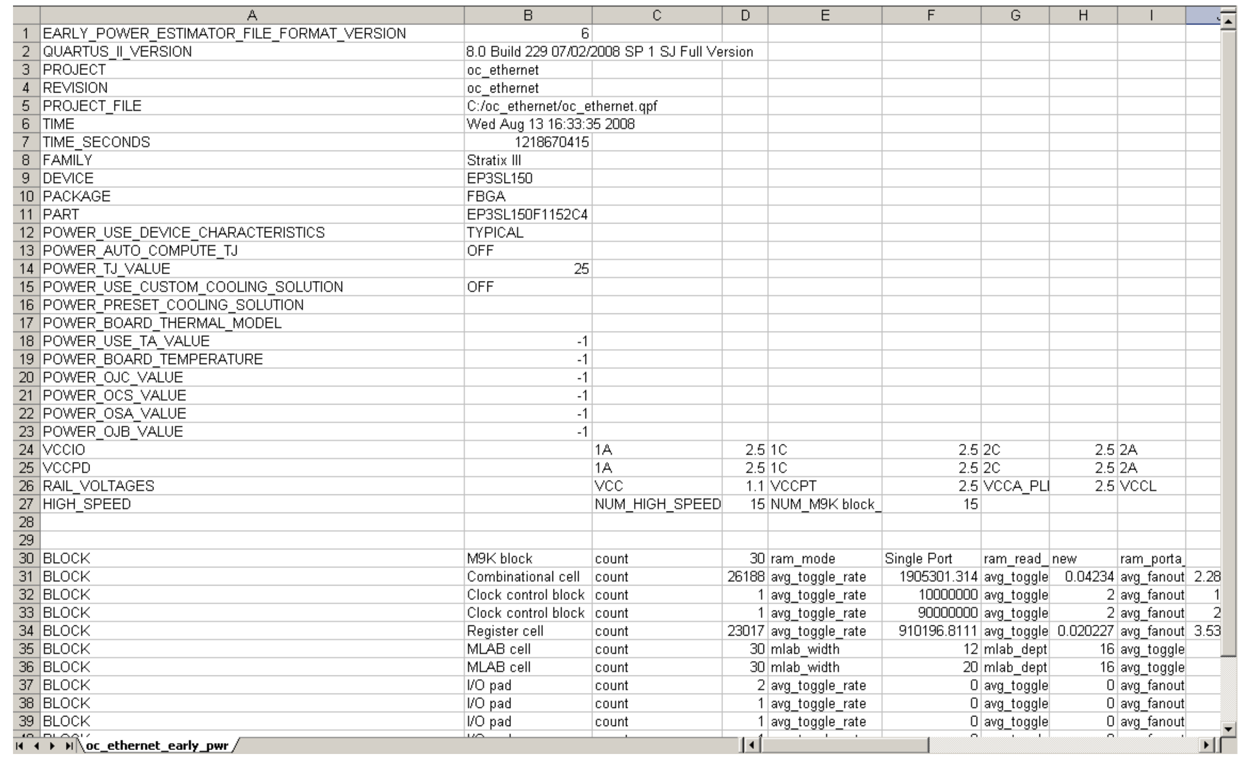
After the Quartus II software successfully generates the power estimator file, a message appears (Figure 12–2).

Figure 12–2. Generate PowerPlay Early Power Estimator File Message



The power estimator file is named <*name of Quartus II project*>**_early_pwr.csv**. Figure 12–3 is an example of the contents of a power estimation file generated by the latest version of Quartus II software using a Stratix II device.

Figure 12–3. Example of Power Estimation File



	A	B	C	D	E	F	G	H	I	J
1	EARLY_POWER_ESTIMATOR_FILE_FORMAT_VERSION	6								
2	QUARTUS_II_VERSION	8.0 Build 229 07/02/2008 SP 1 SJ Full Version								
3	PROJECT	oc_ethernet								
4	REVISION	oc_ethernet								
5	PROJECT_FILE	C:/oc_ethernet/oc_ethernet.qpf								
6	TIME	Wed Aug 13 16:33:35 2008								
7	TIME_SECONDS	1218670415								
8	FAMILY	Stratix III								
9	DEVICE	EP3SL150								
10	PACKAGE	FBGA								
11	PART	EP3SL150F1152C4								
12	POWER_USE_DEVICE_CHARACTERISTICS	TYPICAL								
13	POWER_AUTO_COMPUTE_TJ	OFF								
14	POWER_TJ_VALUE	25								
15	POWER_USE_CUSTOM_COOLING_SOLUTION	OFF								
16	POWER_PRESET_COOLING_SOLUTION									
17	POWER_BOARD_THERMAL_MODEL									
18	POWER_USE_TA_VALUE	-1								
19	POWER_BOARD_TEMPERATURE	-1								
20	POWER_OJC_VALUE	-1								
21	POWER_OCS_VALUE	-1								
22	POWER_OSA_VALUE	-1								
23	POWER_OJB_VALUE	-1								
24	VCCIO		1A	2.5	1C	2.5	2C	2.5	2A	
25	VCCPD		1A	2.5	1C	2.5	2C	2.5	2A	
26	RAIL_VOLTAGES		VCC	1.1	VCCPT	2.5	VCCA_PLI	2.5	VCCL	
27	HIGH_SPEED		NUM_HIGH_SPEED	15	NUM_M9K block_	15				
28										
29										
30	BLOCK	M9K block	count	30	ram_mode	Single Port	ram_read_	new	ram_porta_	
31	BLOCK	Combinational cell	count	26188	avg_toggle_rate	1905301.314	avg_toggle	0.04234	avg_fanout	2.28
32	BLOCK	Clock control block	count	1	avg_toggle_rate	10000000	avg_toggle	2	avg_fanout	1
33	BLOCK	Clock control block	count	1	avg_toggle_rate	90000000	avg_toggle	2	avg_fanout	2
34	BLOCK	Register cell	count	23017	avg_toggle_rate	910196.8111	avg_toggle	0.020227	avg_fanout	3.53
35	BLOCK	MLAB cell	count	30	mlab_width	12	mlab_dept	16	avg_toggle	
36	BLOCK	MLAB cell	count	30	mlab_width	20	mlab_dept	16	avg_toggle	
37	BLOCK	I/O pad	count	2	avg_toggle_rate	0	avg_toggle	0	avg_fanout	
38	BLOCK	I/O pad	count	1	avg_toggle_rate	0	avg_toggle	0	avg_fanout	
39	BLOCK	I/O pad	count	1	avg_toggle_rate	0	avg_toggle	0	avg_fanout	

oc_ethernet_early_pwr

The power estimator file is named *<name of Quartus II project>*_**early_pwr.txt** for older device families.

The PowerPlay Early Power Estimator spreadsheet includes the Import Data macro that parses the information in the power estimation file and transfers it into the spreadsheet. If you do not want to use the macro, you can transfer the data into the Early Power Estimator spreadsheet manually.

If the existing Quartus II project represents only a portion of your full design, you should enter the additional resources used in the final design manually. Therefore, you can edit the spreadsheet and add additional device resources after importing the power estimation file information.

PowerPlay Early Power Estimator File Generator Compilation Report

After successfully generating the power estimation file, a PowerPlay Early Power Estimator File Generator report is created under the **Compilation Report** section. This report is divided into the different sections, such as Summary, Settings, Generated Files, Confidence Metric Details, and Signal Activities.

For more information about the PowerPlay Early Power Estimator File Generator report, refer to “PowerPlay Power Analyzer Compilation Report” on page 12–27.

Table 12–1 lists the main differences between the PowerPlay Early Power Estimator and the PowerPlay Power Analyzer.

Table 12–1. Comparison of PowerPlay Early Power Estimator and PowerPlay Power Analyzer

Characteristic	PowerPlay Early Power Estimator	PowerPlay Power Analyzer
Phase in the design cycle	Any time	After fitting
Tool requirements	Spreadsheet program/Quartus II software	Quartus II software
Accuracy	Medium	Medium to very high
Data inputs	■ Resource usage estimates ■ Clock requirements ■ Environmental conditions ■ Toggle Rate	■ Design after fitting ■ Clock requirements ■ Register transfer level (RTL) simulation results (optional) ■ Post-fitting simulation results (optional) ■ Signal activities per node or entity (optional) ■ Signal activity defaults ■ Environmental conditions
Data outputs *(1)*	■ Total thermal power dissipation ■ Thermal static power ■ Thermal dynamic power ■ Off-chip power dissipation ■ Current drawn from voltage supplies *(2)*	■ Total thermal power ■ Thermal static power ■ Thermal dynamic power ■ Thermal I/O power ■ Thermal power by design hierarchy ■ Thermal power by block type ■ Thermal power dissipation by clock domain ■ Off-chip (non-thermal) power dissipation ■ Device supply currents *(2)*

Notes to Table 12–1:

(1) Early Power Estimator output varies by device family as some features might not be available.

(2) Available only for Arria GX, Stratix IV, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families.

The results of the Power Analyzer are only an estimation of power, not a specification. The purpose of the estimation is to help establish a guide for the design's power budget. Altera recommends measuring the actual power on the board. You must measure the device's total dynamic current during device operation because the estimate is design dependent and depends on many variable factors, including input vector quantity, quality, and exact loading conditions of a PCB design. Static power consumption must not be based on empirical observation; the values reported by the Power Analyzer or data sheet must be used because the devices tested might not exhibit worst-case behavior.

Types of Power Analyses

Understanding the uses of power analysis and the factors affecting power consumption help you use the Power Analyzer effectively. Power analysis meets two significant planning requirements:

- **Thermal planning**: The designer must ensure that the cooling solution is sufficient to dissipate the heat generated by the device. In particular, the computed junction temperature must fall within normal device specifications.
- **Power supply planning**: Power supplies must provide adequate current to support device operation.

The two types of analyses are closely related because much of the power supplied to the device is dissipated as heat from the device. However, in some situations, the two types of analyses are not identical. For example, when you use terminated I/O standards, some of the power drawn from the FPGA device power supply is dissipated in termination resistors, rather than in the FPGA.

Power analysis also addresses the activity of the design over time as a factor that impacts the power consumption of the device. Static power is defined as the power consumed regardless of design activity. Dynamic power is the additional power consumed due to signal activity or toggling.

For power supply planning, you can use the PowerPlay Early Power Estimator at the early stages of your design cycle, or use the Quartus II Power Analyzer reports when the design is completed to get an estimate of your design power requirement.

Factors Affecting Power Consumption

This section describes the factors affecting power consumption. Understanding these factors lets you use the Power Analyzer and interpret its results effectively.

Device Selection

Different device families have different power characteristics. Many parameters affect the device family power consumption, including choice of process technology, supply voltage, electrical design, and device architecture. For example, the Cyclone II device family architecture was designed to consume less static power than the high-performance, full-featured, Stratix II device family.

Power consumption also varies within a single device family. A larger device typically consumes more static power than a smaller device in the same family, due to its larger transistor count. Dynamic power can also increase with device size in devices that employ global routing architectures, such as the MAX device family. Stratix, Cyclone, and MAX II devices do not exhibit significantly increased dynamic power as device size increases.

The choice of device package also affects the device’s ability to dissipate heat. This can impact your cooling solution choice required to meet junction temperature constraints.

Finally, process variation can affect power consumption. Process variation primarily impacts static power, since sub-threshold leakage current varies exponentially with changes in transistor threshold voltage. As a result, it is critical to consult device specifications for static power and not rely on empirical observation. Process variation has a weak effect on dynamic power.

Environmental Conditions

Operating temperature primarily affects device static power consumption. Higher junction temperatures result in higher static power consumption. The device thermal power and cooling solution that you use must result in the device junction temperature remaining within the maximum operating range for that device.

The main environmental parameters affecting junction temperature are the cooling solution and ambient temperature.

Air Flow

Air flow is a measure of how quickly heated air is removed from the vicinity of the device and replaced by air at ambient temperature. This can either be specified as "still air" when no fan is used, or as the linear feet per minute rating of the fan used in the system. Higher air flow decreases thermal resistance.

Heat Sink and Thermal Compound

A heat sink allows more efficient heat transfer from the device to the surrounding area because of its large surface area exposed to the air. The thermal compound that interfaces the heat sink to the device also influences the rate of heat dissipation. The case-to-ambient thermal resistance (θ_{CA}) parameter describes the cooling capacity of the heat sink and thermal compound employed at a given airflow. Larger heat sinks and more effective thermal compounds reduce θ_{CA}.

Ambient Temperature

The junction temperature of a device is equal to:

$$T_{Junction} = T_{Ambient} + P_{Thermal} \cdot \theta_{JA}$$

where θ_{JA} is the total thermal resistance from the device transistors to the environment, having units of degrees Celsius per Watt. The value θ_{JA} is equal to the sum of the junction-to-case (package) thermal resistance (θ_{JC}) and the case-to-ambient thermal resistance (θ_{CA}) of your cooling solution.

Board Thermal Model

The thermal resistance of the path through the board is referred to as the junction-to-board thermal resistance (θ_{JB}) (the units are in degrees Celsius per Watt). This is used in conjunction with the board temperature, as well as the top-of-chip θ_{JA} and ambient temperatures, to compute junction temperature.

Design Resources

The design resource used greatly affects power consumption.

Number, Type, and Loading of I/O Pins

Output pins drive off-chip components, resulting in high-load capacitance that leads to a high-dynamic power per transition. Terminated I/O standards require external resistors that generally draw constant (static) power from the output pin.

Number and Type of Logic Elements, Multiplier Elements, and RAM Blocks

A design with more logic elements (LEs), multiplier elements, and memory blocks tends to consume more power than a design with fewer such circuit elements. Also, the operating mode of each circuit element affects its power consumption. For example, a digital signal processing (DSP) block performing 18×18 multiplications and a DSP block performing multiply-accumulate operations consume different amounts of dynamic power due to different amounts of internal capacitance being charged on each transition. Static power is also affected, to a small degree, by the operating mode of a circuit element.

Number and Type of Global Signals

Global signal networks span large portions of the device and have high capacitance, resulting in significant dynamic power consumption. The type of global signal is important as well. For example, Stratix II devices support several kinds of global clock networks that span either the entire device or a specific portion of the device (a regional clock network covers a quarter of the device). Clock networks that span smaller regions have lower capacitance and therefore, tend to consume less power. In addition, the location of the logic array blocks (LABs) that are driven by the clock network can have an impact, because the Quartus II software automatically disables unused branches of a clock.

Signal Activities

The final important factor in estimating power consumption is the behavior of each signal in the design. The two vital statistics are the toggle rate and the static probability.

The toggle rate of a signal is the average number of times that the signal changes value per unit of time. The units for toggle rate are transitions per second, and a transition is a change from `1` to `0`, or `0` to `1`.

The static probability of a signal is the fraction of time that the signal is logic `1` during the period of device operation that is being analyzed. Static probability ranges from `0` (always at ground) to `1` (always at logic high).

Dynamic power increases linearly with the toggle rate as the capacitive load is charged more frequently for logic and routing. The Quartus II models assume full rail-to-rail switching. For high toggle rates, especially on circuit output I/O pins, the circuit can transition before fully charging the downstream capacitance. The result is a slightly conservative prediction of power by the Quartus II PowerPlay Power Analyzer.

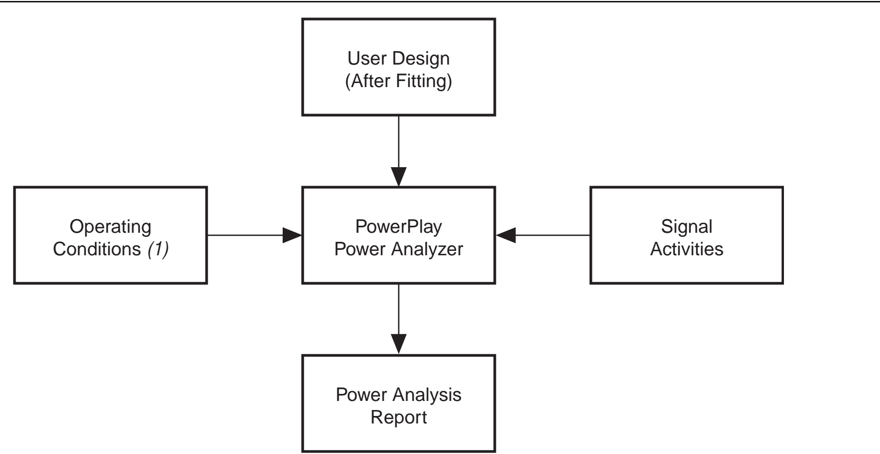
The static power consumed by both routing and logic can sometimes be affected by the static probabilities of their input signals. This effect is due to state-dependent leakage, and has a larger affect on smaller process geometries. The Quartus II software models this effect on devices at 90 nm (or smaller) if it is deemed important to the power estimate. The static power also varies with the static probability of a logic `1` or `0` on the I/O pin when output I/O standards drive termination resistors.

To get accurate results from the power analysis, the signal activities that are used for analysis must be representative of the actual operating behavior of the design. Inaccurate signal toggle rate data is the largest source of power estimation error.

PowerPlay Power Analyzer Flow

The PowerPlay Power Analyzer supports accurate and representative power estimation by allowing you to specify all the important design factors affecting power consumption. Figure 12–4 shows the high-level Power Analyzer flow.

Figure 12–4. PowerPlay Power Analyzer High-Level Flow



Note to Figure 12–4:

(1) Operating condition specifications are available only for the Arria GX devices, Stratix IV, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families.

The PowerPlay Power Analyzer requires that your design is synthesized and fit to the target device. Therefore, the Power Analyzer knows the target device, and how the design is placed and routed on the device. The electrical standard used by each I/O cell and the capacitive load on each I/O standard must be specified in the design to obtain accurate I/O power estimates.

Operating Conditions

For the Arria GX, Stratix IV, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families, you can specify the operating conditions for power analysis in the Quartus II software.

The following settings are available in the **Settings** dialog box:

- **Device power characteristics**—Should the Power Analyzer assume typical silicon or maximum power silicon? The typical setting is useful for comparing to empirical data measured on an average unit. The maximum setting uses worst-case data to provide a boundary to the worst-case device that you could receive. This setting impacts the static power estimate.

- **Selectable Core Voltage**—You can select a suitable core supply voltage for your design based on performance and power requirements using the **Core Supply Voltage** option, available for the Stratix III devices with variable voltage support. The power consumption of a device is heavily dependent on the voltage, so it is very important to choose the right core supply voltage for your design. The core supply voltage provides power to device logic resources such as LABs, Memory LABs (MLABs), DSP functions, memory, and interconnects.
- **Environmental conditions and junction temperature**—By default, the Power Analyzer automatically computes the junction temperature based on the specified ambient temperature and the cooling solution that you selected. For a more accurate analysis, enter the thermal resistance of your cooling solution. For some cooling solutions, such as a heat sink with no forced airflow, the thermal resistance varies with the amount of thermal power that is dissipated. Air convection increases as the difference between the device temperature and the ambient temperature increases, reducing thermal resistance. When entering a thermal resistance in such cases, it is important to use the thermal resistance that occurs when the heat flow (Q) is equal to the thermal power generated by the device.

You can also specify a junction temperature in the PowerPlay Power Analyzer. However, Altera does not recommend this because the PowerPlay Power Analyzer provides more accurate results by computing the junction temperature.

- **Board Thermal Modeling**—If you want the Power Analyzer thermal model to take the θ_{JB} into consideration, set the board thermal model to either **Typical** or **Custom**. This feature produces more accurate thermal power estimation.

A **Typical** board thermal model automatically sets θ_{JB} to a value based on the package and device selected. You only have to specify a board temperature. If you choose a **Custom** board thermal model, you must specify a value for θ_{JB} and a board temperature. If you do not want the PowerPlay Power Analyzer thermal model to take the θ_{JB} resistance into consideration, set the **Board thermal model** option to **None** (conservative). In this case, the path through the board and power dissipation is not considered, and a more conservative thermal power estimate is obtained.

The **Board thermal model** option is only available if you select the **Auto compute junction temperature** option with the pre-set cooling solution set to a heat sink solution option or custom solution. This option is disabled when a cooling solution with no heat sink is selected, as thermal conduction through the board is included in the θ_{JA} value used to compute a junction temperature in that case.

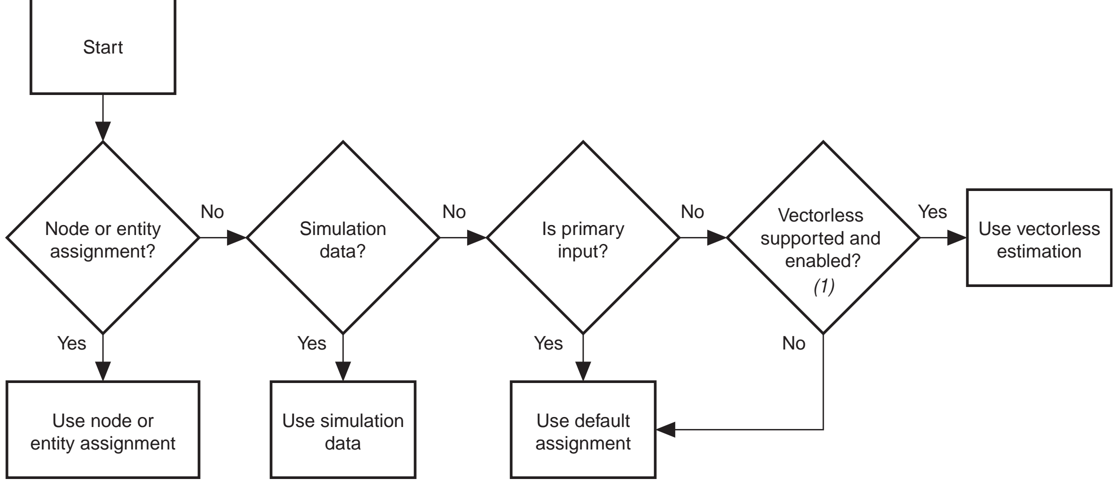
Signal Activities Data Sources

The Power Analyzer provides a flexible framework for specifying signal activities. This reflects the importance of using representative signal activity data during power analysis. You can use the following sources to provide information about signal activity:

- Simulation results
- User-entered node, entity, and clock assignments
- User-entered default toggle rate assignment
- Vectorless estimation

The PowerPlay Power Analyzer lets you mix and match the signal activity data sources on a signal-by-signal basis. Figure 12–5 shows the priority scheme. The data sources are described in the following sections.

Figure 12–5. Signal Activity Data Source Priority Scheme



Note to Figure 12–5:

(1) Vectorless estimation is available only for the Arria GX, Stratix IV, Stratix III, Stratix II, Stratix II GX, Cyclone II, HardCopy II, and MAX II device families.

Simulation Results

The Power Analyzer directly reads the waveforms generated by a design simulation. The static probability and toggle rate for each signal is calculated from the simulation waveform. Power analysis is most accurate when simulations are generated using representative input stimuli.

The Power Analyzer reads the results generated by the following simulators:

- Quartus II Simulator
- ModelSim® VHDL, Active HDL, ModelSim Verilog HDL, ModelSim-Altera VHDL, ModelSim-Altera Verilog
- NC-Verilog, NC-VHDL
- VCS

Signal activity and static probability information are stored in a Signal Activity File (**.saf**) or can be derived from a Value Change Dump File (**.vcd**), described in “Signal Activities” on page 12–7. The Quartus II simulator generates an **.saf** file or a **.vcd** file, which is then read by the Power Analyzer.

For third-party simulators, use the **Quartus II EDA Tool Settings for Simulation** to specify a **Generate Value Change Dump** file script. These scripts instruct the third-party simulators to generate a **.vcd** file that encodes the simulated waveforms. The Quartus II Power Analyzer reads this file directly to derive the toggle rate and static probability data for each signal.

Third-party EDA simulators, other than those listed above, can generate a **.vcd** file that can then be used with the Power Analyzer. For those simulators, it is necessary to manually create a simulation script to generate the appropriate **.vcd** file.

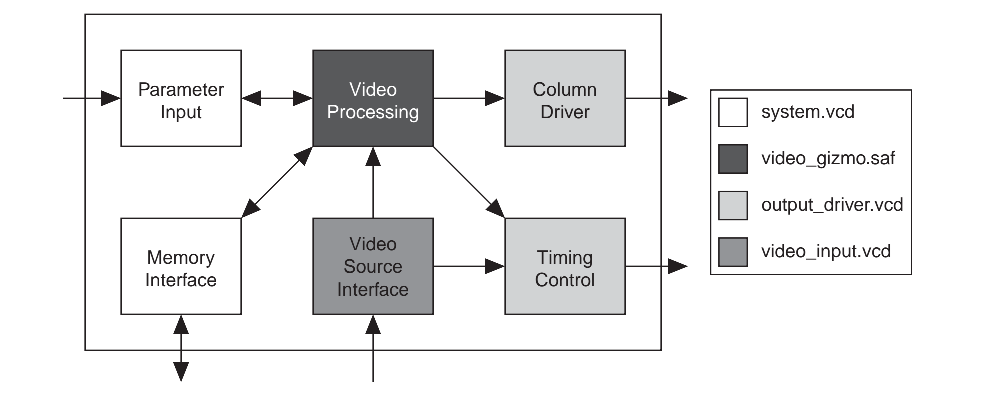
You can use a **.saf** or **.vcd** file created for power analysis to optimize the design for power during fitting by utilizing the appropriate settings in the **PowerPlay power optimization** list, available in **Fitter Settings** page of the **Settings** dialog box.

For more information about power optimization, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Using Simulation Files in Modular Design Flows

A common design practice is to create modular or hierarchical designs in which you develop each design entity separately and then instantiate it in a higher-level entity, forming a complete design. Simulation is performed on a complete design or on each modular design for verification. The Quartus II PowerPlay Power Analyzer Tool supports modular design flows when reading the signal activities generated from these simulation files, as shown in Figure 12–6.

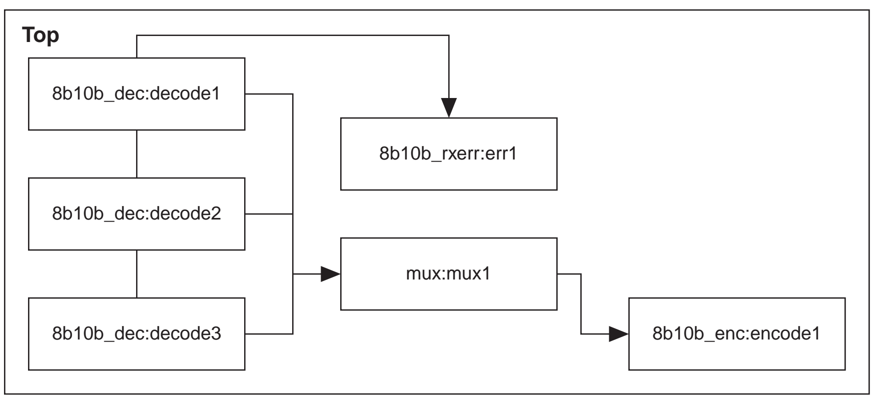
Figure 12–6. Modular Simulation Flow



When specifying a simulation file, an associated design entity name can be given, such that the signal activities derived from the simulation file (**.vcd** or **.saf** file) can be imported into the Power Analyzer for that particular design entity. The PowerPlay Power Analyzer Tool also supports the specification of multiple **.saf** files for power analysis with each having an associated design entity name to allow the integration of partial design simulations into a complete design power analysis. When specifying multiple **.saf** files for your design, it is possible that more than one simulation file will contain signal activity information for the same signal. In the case where multiple **.saf** files are applied to the same design entity, the signal activity used in the power analysis is the equal-weight arithmetic average of each **.saf** file. Also in the case where multiple simulation files are applied to design entities at different levels in the design hierarchy, the signal activity used in the power analysis is derived from the simulation file that is applied to the most specific design entity.

Figure 12–7 shows an example of a hierarchical design. The design Top consists of three 8b/10b Decoders, followed by a multiplexer whose output is then encoded again before being output from the design. There is also an error-handling module that handles any 8b/10b decoding errors. The top-level module, called Top, automatically contains the design's top-level entity and any logic not defined as part of another module. The design file for the top-level module might be just a wrapper for the hierarchical entities below it, or it might contain its own logic. The following usage scenarios show common ways that you can simulate your design and import **.saf** files into the PowerPlay Power Analyzer Tool.

Figure 12–7. Example Hierarchical Design



Complete Design Simulation

You can simulate the entire design Top, generating a **.vcd** file if you use a third-party simulator, or generating a **.saf** or **.vcd** file if you use the Quartus II Simulator. The **.vcd** or **.saf** file can then be imported (specifying Entity Top) into the Power Analyzer. The resulting power analysis uses all the signal activities information from the generated **.vcd** or **.saf** file, including those that apply to submodules, such as `decode [1-3]`, `err1`, `mux1`, and `encode1`.

Modular Design Simulation

You can simulate submodules of the design Top independently, and then import all of the resulting **.saf** files into the Power Analyzer. For example, you can simulate the `8b10b_dec` independent of the entire design, as well as multiplexer, `8b10b_rxerr`, and `8b10b_enc`. You can then import the **.vcd** or **.saf** file generated from each simulation by specifying the appropriate instance name. For example, if the files produced by the simulations are **8b10b_dec.vcd**, **8b10b_enc.vcd**, **8b10b_rxerr.vcd**, and **mux.saf**, the import specifications in Table 12–2 are used.

Table 12–2. Import Specifications (Part 1 of 2)

File Name	Entity
8b10b_dec.vcd	`Top|8b10b_dec:decode1`
8b10b_dec.vcd	`Top|8b10b_dec:decode2`
8b10b_dec.vcd	`Top|8b10b_dec:decode3`
8b10b_rxerr.vcd	`Top|8b10b_rxerr:err1`

Table 12–2. Import Specifications (Part 2 of 2)

File Name	Entity
8b10b_enc.vcd	`Top|8b10b_enc:encode1`
mux.saf	`Top|mux:mux1`

The resulting power analysis applies the simulation vectors found in each file to the assigned entity. Simulation provides signal activities for the pins and for the outputs of functional blocks. If the inputs to an entity instance are input pins for the entire design, the simulation file associated with that instance does not provide signal activities for the inputs of that instance. For example, an input to an entity such as `mux1` has its signal activity specified at the output of one of the decode entities.

Multiple Simulations on the Same Entity

You can perform multiple simulations of an entire design or specific modules of a design. For example, in the process of verifying the "Top" design, you can have three different simulation testbenches: one for normal operation, and two for corner cases. Each of these simulations produces a separate **.vcd** or **.saf** file. In this case, apply the different **.vcd** or **.saf** file names to the same top-level entity, shown in Table 12–3.

Table 12–3. Multiple Simulation File Names and Entities

File Name	Entity
normal.saf	`Top`
corner1.vcd	`Top`
corner2.vcd	`Top`

The resulting power analysis uses an arithmetic average of the signal activities calculated from each simulation file to obtain the final signal activities used. Thus, if a signal `err_out` has a toggle rate of 0 toggles per second in **normal.saf**, 50 toggles per second in **corner1.vcd**, and 70 toggles per second in **corner2.vcd**, the final toggle rate that is used in the power analysis is 40 toggles per second.

Overlapping Simulations

You can perform a simulation on the entire design Top and more exhaustive simulations on a sub-module, such as `8b10b_rxerr`. Table 12–4 shows the import specification for overlapping simulations.

Table 12–4. Overlapping Simulation Import Specifications

File Name	Entity
full_design.vcd	`Top`
error_cases.vcd	`Top|8b10b_rxerr:err1`

In this case, signal activities from **error_cases.vcd** are used for all of the nodes in the generated **.saf** file, and signal activities from **full_design.vcd** are used for only those nodes that do not overlap with nodes in **error_cases.vcd**. In general, the more specific hierarchy (the most bottom-level module) is used to derive signal activities for overlapping nodes.

Partial Simulations

You can perform a simulation where the entire simulation time is not applicable to signal activity calculation. For example, if you run a simulation for 10,000 clock cycles and reset the chip for the first 2,000 clock cycles. If the signal activity calculation is performed over all 10,000 cycles, the toggle rates are typically only 80% of their steady state value (since the chip is in reset for the first 20% of the simulation). In this case, you should specify the useful parts of the **.vcd** file for power analysis. The **Limit VCD Period** option enables you to specify a start and end time to be used when performing signal activity calculations.

Node Name Matching Considerations

Node name mismatches happen when you have **.saf** or **.vcd** files applied to entities other than the top-level entity. In a modular design flow, the gate-level simulation files created in different Quartus II software projects may not match their node names with the current Quartus II project.

For example, if you have a file named **8b10b_enc.vcd**, which was generated in a separate project called **8b10b_enc** and is simulating the `8b10b` encoder, and you import that **.vcd** file into another project called **Top**, you might encounter name mismatches when applying the **.vcd** file to the `8b10b_enc` module in the **Top** project. This is because all of the combinational nodes in the **8b10b_enc.vcd** file might be named differently in the **Top** project.

You can avoid name mismatching by using only register transfer level (RTL) simulation data, where register names usually do not change, or by using an incremental compile flow that preserves node names in conjunction with a gate-level simulation.

To ensure the best accuracy, Altera recommends using an incremental compilation flow to preserve your design’s node names.

For more information about the incremental compile flow, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Glitch Filtering

The Power Analyzer defines a glitch as two signal transitions that are so closely spaced in time that the pulse, or glitch, occurs faster than the logic and routing circuitry can respond. The output of a transport delay model simulator (the default mode of the Quartus II simulator) generally contains glitches for some signals. The device’s logic and routing structures form a low-pass filter that filters out glitches that are tens to hundreds of picoseconds long, depending on the device family.

Some third-party simulators use different models than the transport delay model as default. Different models cause differences in signal activity and power estimation. The inertial delay model, which is the ModelSim default model, filters out many more glitches than the transport delay model; therefore, it usually yields a lower power estimate.

Altera recommends using the transport simulation model when using the Quartus II glitch filtering support with third-party simulators. If the inertial simulation model is used, simulation glitch filtering has little effect.

For more information about how to set the simulation model type for your specific simulator, refer to the Quartus II Help.

Glitch filtering in a simulator can also filter a glitch on one LE (or other circuit element) output from propagating to downstream circuit elements so that the glitch will not affect simulated results. This prevents a glitch on one signal from producing non-physical glitches on all downstream logic, which would result in a signal toggle rate that is too high and a power estimate that is too high. Circuit elements in which every input transition produces an output transition, including multipliers and logic cells configured to implement XOR functions, are especially prone to glitches. Therefore, circuits with many such functions can have power estimates that are too high when glitch filtering is not used.

Altera recommends that the glitch filtering feature be used to obtain the most accurate power estimates. For **.vcd** files, the Power Analyzer flows support two types of glitch filtering, both of which are recommended for power estimation. In the first, glitches are filtered during simulation. To enable this level of glitch filtering in the Quartus II software for supported third-party simulators, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears.
3. Select the **Tool Name** to use for the simulation.
4. Turn on the **Enable glitch filtering** option.

To enable this level of glitch filtering in the Quartus II software using the Quartus II Simulator, refer to “Generating a .saf or .vcd File Using the Quartus II Simulator” on page 12–18.

The second level of glitch filtering occurs while the Power Analyzer is reading the **.vcd** file generated by the third-party simulator or Quartus II Simulator. Enable this level of glitch filtering by performing the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **PowerPlay Power Analyzer Settings**. The **PowerPlay Power Analyzer Settings** page appears.
3. Under **Input File(s)**, turn on the **Perform glitch filtering on VCD files** option.

Altera recommends that you use both forms of glitch filtering.

The **.vcd** file reader performs complementary filtering to the filtering performed during simulation and is often not as effective. While the **.vcd** file reader can remove glitches on logic blocks, it has no way of determining how downstream logic and routing are affected by a given glitch, and may might eliminate the impact of the glitch completely. Filtering the glitches during simulation avoids switching downstream routing and logic automatically.

When running simulation for design verification (rather than to produce input to the Quartus PowerPlay Power Analyzer), Altera recommends leaving glitch filtering turned off. This produces the most rigorous and conservative simulation from a functionality viewpoint. When performing simulation to produce input for the Quartus II PowerPlay Power Analyzer, Altera recommends turning on glitch filtering to produce the most accurate power estimates.

Node and Entity Assignments

You can assign specific toggle rates and static probabilities to individual nodes and entities in the design. These assignments have the highest priority, overriding data from all other signal activity sources.

Use the Assignment Editor or Tcl commands to make the **Power Toggle Rate** and **Power Static Probability** assignments. You can specify the power toggle rate as an absolute toggle rate in transitions using the **Power Toggle Rate** assignment or you can use the **Power Toggle Rate Percentage** assignment to specify a toggle rate relative to the clock domain of the assigned node for more specific assignment made in terms of hierarchy level.

If the **Power Toggle Rate Percentage** assignment is used, and the given node does not have a clock domain, a warning is issued and the assignment is ignored.

For more information about how to use the Assignment Editor in the Quartus II software, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

This method is appropriate for special-case signals where you have specific knowledge of the signal or entity being analyzed. For example, if you know that a 100-MHz data bus or memory output produces data that is essentially random (uncorrelated in time), you can directly enter a 0.5 static probability and a toggle rate of 50 million transitions per second.

Bidirectional I/O pins are treated specially. The combinational input port and the output pad for a given pin share the same name. However, those ports might not share the same signal activities. For the purpose of reading signal activity assignments, the Power Analyzer creates a distinct name `<node_name~output>` when the bidirectional signal is configured as an output and `<node_name~result>` when the signal is configured as an input. For example, if a design has a bidirectional pin named `MYPIN`, assignments for the combinational input use the name `MYPIN~result`, and the assignments for the output pad use the name `MYPIN~output`.

When making the logic assignment in the Assignment Editor, you will not find the `MYPIN~result` and `MYPIN~output` node names in the Node Finder. Therefore, to make the logic assignment, you must manually enter the two differentiating node names to make the specific assignment for the input and output port of the bidirectional pin.

Timing Assignments to Clock Nodes

For clock nodes, the Power Analyzer uses the timing requirements to derive the toggle rate when neither simulation data nor user entered signal activity data is available.

f_{MAX} requirements specify full cycles per second, but each cycle represents a rising transition and a falling transition. For example, a clock f_{MAX} requirement of 100 MHz corresponds to 200 million transitions per second.

Default Toggle Rate Assignment

You can specify a default toggle rate for primary inputs and all other nodes in the design. The default toggle rate is used when no other method has specified the signal activity data.

The toggle rate can be specified in absolute terms (transitions per second) or as a fraction of the clock rate in effect for each particular node. The toggle rate for a given clock is derived from the timing settings for the clock. For example, if a clock is specified with an f_{MAX} constraint of 100 MHz and a default relative toggle rate of 20%, nodes in this clock domain transition in 20% of the clock periods, or 20 million transitions occur per second. In some cases, the Power Analyzer cannot determine the clock domain for a given node because there is either no clock domain for the node or it is ambiguous. In these cases, the Power Analyzer substitutes and reports a toggle rate of zero.

Vectorless Estimation

For some device families, the Power Analyzer automatically derives estimates for signal activity on nodes with no simulation or user-entered signal-activity data. Vectorless estimation is available and enabled by default for Arria GX, Stratix IV, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families. Vectorless estimation statistically estimates the signal activity of a node based on the signal activities of all nodes feeding that node, and on the actual logic function implemented by the node. The **PowerPlay Power Analyzer Settings** dialog box lets you disable vectorless estimation. When enabled, vectorless estimation takes priority over default toggle rates. Vectorless estimation does not override clock assignments.

Vectorless estimation cannot derive signal activities for primary inputs. Vectorless estimation is generally accurate for combinational nodes, but not for registered nodes. Therefore, simulation data for at least the registered nodes and I/O nodes is required for accuracy.

Using the PowerPlay Power Analyzer

For all flows that use the PowerPlay Power Analyzer, synthesize your design first and then fit it to the target device. You must either provide timing assignments for all clocks in the design or use a simulation-based flow to generate activity data. The I/O standard used on each device input or output and the capacitive load on each output must be specified in the design.

Common Analysis Flows

You can use the analysis flows in this section with the PowerPlay Power Analyzer. However, vectorless activity estimation is only available for some device families.

Signal Activities from Full Post-Fit Netlist (Timing) Simulation

This flow provides the highest accuracy because all node activities reflect actual design behavior, provided that supplied input vectors are representative of typical design operation. Results are better if the simulation filters glitches. The disadvantage with this method is that simulation times can be long.

Signal Activities from RTL (Functional) Simulation, Supplemented by Vectorless Estimation

In this flow, simulation provides toggle rates and static probabilities for all pins and registers in the design. Vectorless estimation fills in the values for all the combinational nodes between pins and registers. This method yields good results, because vectorless estimation is accurate, given that the proper pin and register data is provided. This flow usually provides a compilation time benefit to the user in the third-party RTL simulator.

RTL simulation may not provide signal activities for all registers in the post-fitting netlist because some register names might be lost during synthesis. For example, synthesis might automatically transform state machines and counters, thus changing the names of registers in those structures.

Signal Activities from Vectorless Estimation, User-Supplied Input Pin Activities

This option provides a low level of accuracy, because vectorless estimation for registers is not entirely accurate.

Signal Activities from User Defaults Only

This option provides the lowest degree of accuracy.

Generating a .saf or .vcd File Using the Quartus II Simulator

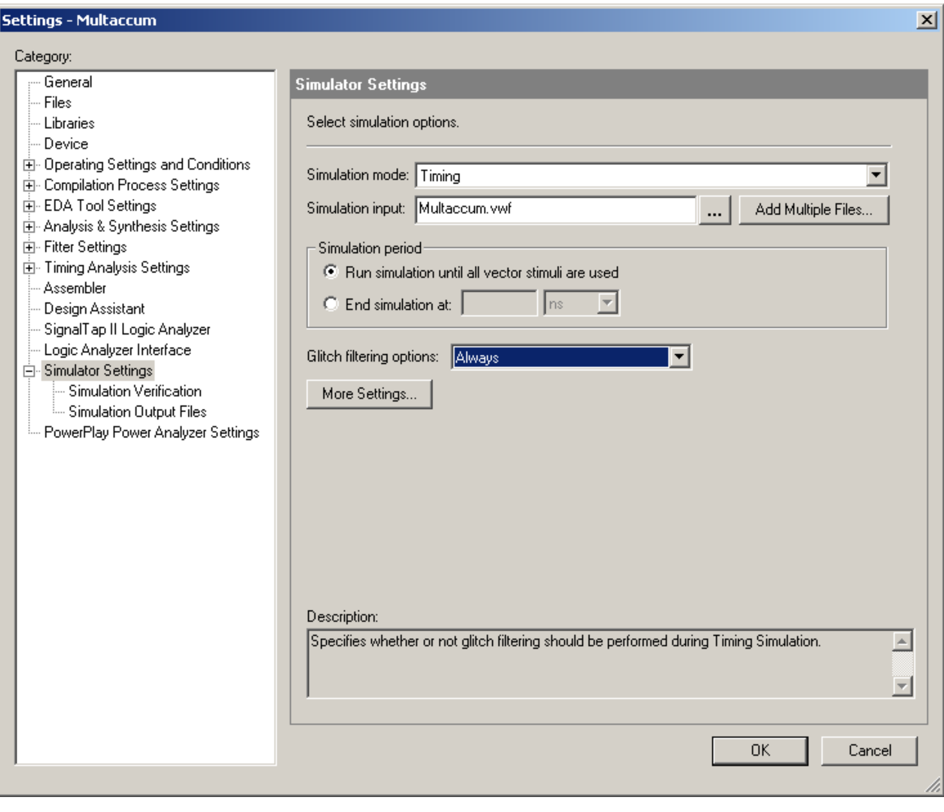
While performing a timing or functional simulation using the Quartus II Simulator, you can generate a **.saf** or **.vcd** file. These files store the toggle rate and static probability for each connected output signal based on the simulation vectors that are entered in the Vector Waveform File (**.vwf**) or the Vector File (**.vec**). You can use the **.saf** file(s) or **.vcd** file(s) as input to the PowerPlay Power Analyzer to estimate power for your design.

For more accurate results, Altera recommends that you use the **.saf** file created in the Quartus II Simulator as the input to the PowerPlay Power Analyzer.

To create a **.saf** or **.vcd** file for your design, perform the following steps:

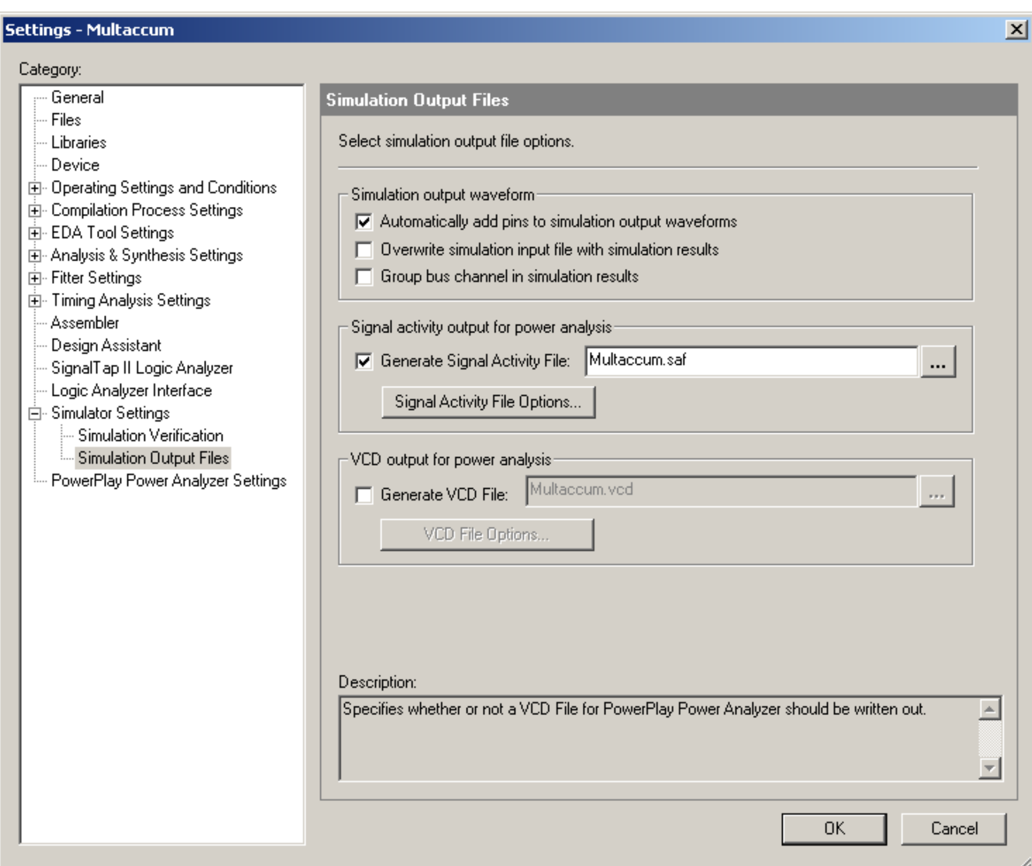
1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulator Settings**. The **Simulator Settings** page appears (Figure 12–8).

Figure 12–8. Simulator Settings Page



3. In the **Simulation mode** list, select either **Timing** or **Functional**. Refer to "Common Analysis Flows" on page 12–17 for a description of the difference in accuracy between the two types of simulation modes.
4. (Optional) Click **More Settings**. The **More Simulator Settings** dialog box appears.
5. (Optional) Turn on glitch filtering. To turn on glitch filtering, in the **Glitch filtering options** list, select **Always**.
6. In the **Category** list, click the "+" icon to expand **Simulator Settings** and select **Simulation Output Files** (Figure 12–9).

Figure 12–9. Simulator Output Files Page of the Settings Dialog Box



7. Turn on **Generate Signal Activity File** and enter the file name for the **.saf** file.

For more information about the Quartus II Simulator and how to create a **.saf** file, refer to the *Quartus II Simulator* chapter in volume 3 of the *Quartus II Handbook*.

When generating a **.vcd** file from the Quartus Simulator, you must make sure that you add **all nodes** to the input vector wave file. Only the nodes that have been added to your vector file are output to the Quartus-generated **.vcd** file. This is not the case when generating a **.saf** file. The Quartus II Simulator creates a **.saf** file, including all the internal nodes of your design, even if the stimuli file contains only the input vectors for your simulation.

8. (Optional) Click **Signal Activity File Options**. The **Signal Activity File Options** dialog box appears. Turn on the **Limit signal activity period** option to specify the simulation period to use when calculating the signal activities.

   Power estimation can be performed for the entire simulation time or for a portion of the simulation time. This allows you to look at the power consumption at different points in your overall simulation without having to rework your testbenches. This feature is also useful when multiple clock cycles are necessary to initialize the state of the design, but you want to measure the signal activity only during the normal operation of the design, not during its initialization phase. You

© March 2009 Altera Corporation

can specify the start time and end time in the **Signal Activity File Options** dialog box by turning on the **Limit signal activity period** option. Simulation information is used during this time interval only to calculate toggle rates and static probabilities. If no time interval is specified, the whole simulation is used to compute signal activity data.

9. After the simulation is complete, a **.saf** file is generated with the specified filename and stored in the main project directory.

For more information about how to perform simulations in the Quartus II software, refer to the Quartus II Help.

Generating a VCD File Using a Third-Party Simulator

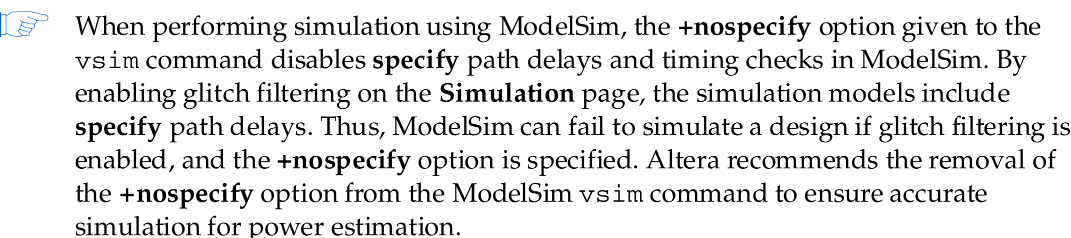
You can use other EDA simulation tools, such as the Model Technology™ ModelSim® software, to perform a simulation and create a **.vcd** file. You can use this file as input to the PowerPlay Power Analyzer to estimate power for your design. To do this, you must tell the Quartus II software to generate a script file that is used as input to the third-party simulator. This script tells the third-party simulator to generate a **.vcd** file that contains all the output signals. For more information about the supported third-party simulators, refer to "Simulation Results" on page 12–10.

To create a **.vcd** file for your design, perform the following steps:

1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation**. The **Simulation** page appears.
3. In the **Tool name** list, select the appropriate EDA simulation tool.
4. In the **Format for output netlist** list, select **VHDL** or **Verilog**.
5. Turn on **Generate Value Change Dump (VCD) file script**.

This turns on the **Map illegal HDL character** and **Enable glitch filtering** options.

6. (Optional) **Map illegal HDL characters** ensures that all signals have legal names and that signal toggle rates are available later in the PowerPlay Power Analyzer.
7. (Optional) By turning on **Enable glitch filtering**, glitch filtering logic is the output when you generate an EDA netlist for simulation. This option is always available, regardless of whether or not you want to generate the **.vcd** file scripts. For more information about glitch filtering, refer to "Glitch Filtering" on page 12–14.



When performing simulation using ModelSim, the **+nospecify** option given to the `vsim` command disables **specify** path delays and timing checks in ModelSim. By enabling glitch filtering on the **Simulation** page, the simulation models include **specify** path delays. Thus, ModelSim can fail to simulate a design if glitch filtering is enabled, and the **+nospecify** option is specified. Altera recommends the removal of the **+nospecify** option from the ModelSim `vsim` command to ensure accurate simulation for power estimation.

8. Click **Script Settings**. The **Script Settings** dialog box appears.

Select which signals should be output to the **.vcd** file. With **All signals** selected, the generated script instructs the third-party simulator to write all connected output signals to the **.vcd** file. With **All signals except combinational lcell outputs** selected, the generated script tells the third-party simulator to write all connected output signals to the **.vcd** file, except logic cell combinational outputs.

You might not want to write all output signals to the file because the file can become extremely large (since its size depends on the number of output signals being monitored and the number of transitions that occur).

9. Click **OK**.
10. Type a name for your testbench in the **Design instance name** box.
11. Compile your design with the Quartus II software and generate the necessary EDA netlist and script that tells the third-party simulator to generate a **.vcd** file.

For more information about NativeLink use, refer to *Section I. Simulation* in volume 3 of the *Quartus II Handbook*.

12. Perform a simulation with the third-party EDA simulation tool. Call the generated script in the simulation tool before running the simulation. The simulation tool generates the **.vcd** file and places it in the project directory.

Generating a VCD File from ModelSim Software

The following example provides step-by-step instructions to successfully produce a **.vcd** file with the ModelSim software:

1. In the Quartus II software, on the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulator Settings**. On the **Simulator Settings** page, choose the appropriate ModelSim selection in the **Tool Name** list, and turn on the **Generate Value Change Dump File Script** option.
3. To generate the **.vcd** file, perform a full compilation.
4. In the ModelSim software, compile the files necessary for simulation.
5. Load your design by clicking **Start Simulation** on the Tools menu, or use the `vsim` command.
6. Source the Quartus II **.vcd** script created in step 3 using the following command:
`source <design>_dump_all_vcd_nodes.tcl`
7. Run the simulation (for example, `run 2000ns` or `run -all`).
8. Quit the simulation using the `quit -sim` command, if required.
9. Exit the ModelSim software. If you do not exit the software, the ModelSim software might end the writing process of the **.vcd** files improperly, resulting in a corrupted **.vcd** file.

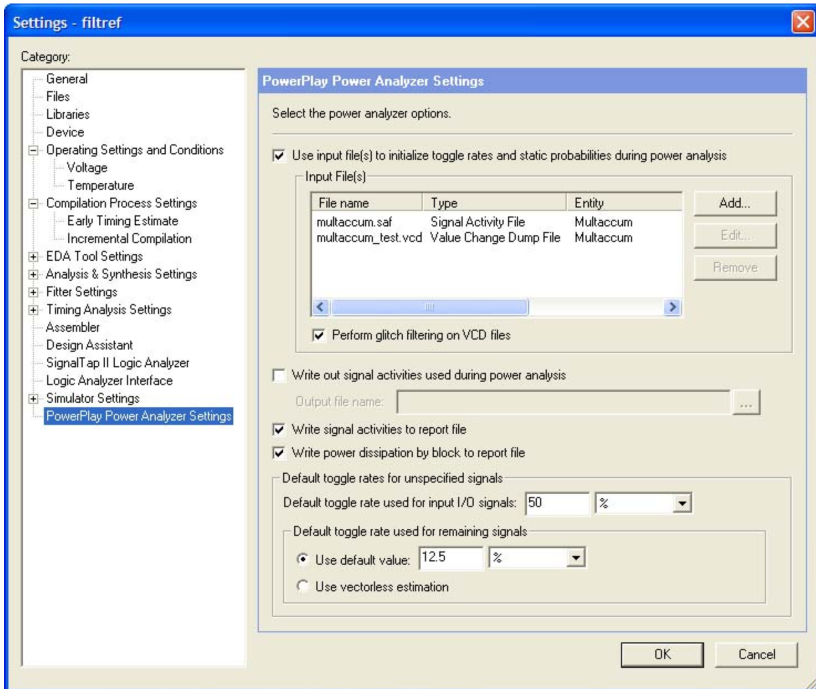
For more information about how to create a **.vcd** file in other third-party EDA simulation tools, refer to *Section I. Simulation*, in volume 3 of the *Quartus II Handbook*.

Running the PowerPlay Power Analyzer Using the Quartus II GUI

To run the PowerPlay Power Analyzer using the Quartus II GUI, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **PowerPlay Power Analyzer Settings**. The **PowerPlay Power Analysis** page appears (Figure 12–10).

Figure 12–10. PowerPlay Power Analyzer Settings

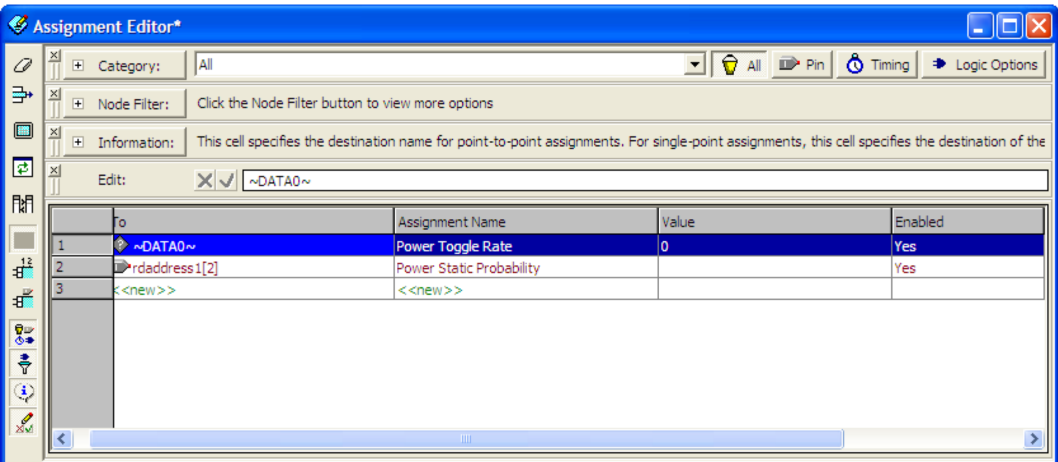


3. (Optional) If you want to use either **.saf** file(s) or **.vcd** file(s) or both as an input to the PowerPlay Power Analyzer, turn on **Use input file(s) to initialize toggle rates and static probabilities during power analysis**.

   (Optional) The **Edit** button allows you to change the settings for a selected file from the list. The **Remove** button allows you to remove a selected file from the list.
4. Click **Add**. The **Add Power Input File** dialog box appears.
5. Add your **.saf** file(s) or **.vcd** file(s) by clicking the browse button for the **File name** field.
6. The **Entity** field enables you to specify the design entity (hierarchy) to which the entered power input file applies. To enter the entity, you can type in the box or browse through the list of your design entities. To browse your design entities, click the browse button. The **Select Hierarchy** dialog box appears. You can specify multiple entities in the entity text box by using comma delimiters. You can specify whether the input file is a **.vcd** or **.saf** file under **Input File Type**.

7. (Optional) **Limit VCD period** is enabled only when **VCD file** is selected. This enables you to specify the simulation period to use when calculating the signal activities. For more information, refer to “Generating a .saf or .vcd File Using the Quartus II Simulator” on page 12–18.
8. Click **OK**.
9. Click **OK** in the **Add Power Input File** dialog box.
10. (Optional) Turn on **Perform glitch filtering on VCD files**. This option is recommended. For more information, refer to “Glitch Filtering” on page 12–14.
11. (Optional) Turn on **Write out signal activities used during power analysis**. In the **Output file name** list, select the output file name. This file contains all the signal activities information used during the power estimation of your design. This is recommended if you used a **.vcd** file as input into the PowerPlay Power Analyzer, because it reduces the run time of any subsequent power estimation. You can use the generated **.saf** file as input instead of the original **.vcd** file.
12. (Optional) Turn on **Write signal activities to report file**.
13. (Optional) Turn on **Write power dissipation by block to report file** to enable the output of detailed thermal power dissipation by block to be included in the PowerPlay Power Analyzer report.
14. (Optional) You can also use the Assignment Editor to enter the Power Toggle Rate or Power Toggle Rate Percentage, and the Power Static Probability for a node or entity in your design, shown in Figure 12–11.

Figure 12–11. Assignment Editor, *(Note 1)*, *(Note 2)*



Notes to Figure 12–11:

(1) The assignments made with the Assignment Editor override the values already existing in the **.saf** file or **.vcd** file.

(2) You can also use Tcl script commands to make these assignments.

For more information about how to use the Assignment Editor in the Quartus II software, see the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*. For information about scripting, see the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

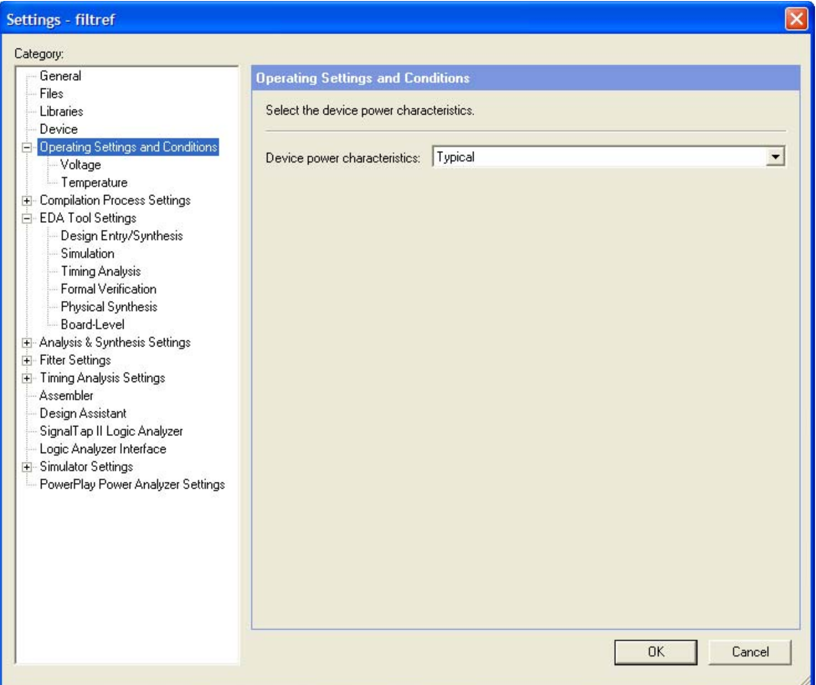
© March 2009 Altera Corporation

15. Specify the toggle rate in the **Default toggle rate used for input I/O signals** field. This toggle rate is used for all unspecified input I/O signal toggle rates regardless of whether or not the device family supports vectorless estimation. By default, its value is set to 12.5%. The default static probability for unspecified input I/O signals is 0.5 and cannot be changed.

16. Select either **Use default value** or **Use vectorless estimation** for Arria GX, Stratix IV, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, or MAX II device families. For all other device families, only **Use default value** is available. This setting controls how the remainder of the unspecified signal activities are calculated. For more information, refer to "Vectorless Estimation" on page 12–17 and "Default Toggle Rate Assignment" on page 12–17.

17. In the **Category** list, select **Operating Settings and Conditions**. This option is available only for the Arria GX, Stratix IV, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families (Figure 12–12).

Figure 12–12. Operating Conditions



18. In the **Device power characteristics** list, select **Typical** or **Maximum**. The default is **Typical**.

19. In the **Category** list, click the "+" icon to expand **Operating Settings and Conditions** and click **Voltage**. The **Voltage** page appears.

20. For the devices with selectable core voltage support, in the **Core supply voltage** list, select the core supply voltage for your device. This option is available for the latest devices with variable voltage selection.

21. In the **Category** list, under **Operating Settings and Conditions,** select **Temperature**. The **Temperature** page appears.

22. Under **Junction temperature range**, specify a junction temperature in degrees Celsius and specify the junction temperature range. Select the **Low temperature** and **High temperature** range for your selected device.
23. Specify the junction temperature and cooling solution settings. You can select **Specify junction temperature** or **Auto compute junction temperature using cooling solution**.
24. (Optional) Under **Board thermal modeling**, select the **Board thermal model** and type the **Board temperature**. This feature can only be turned on when you have selected **Auto compute junction temperature using cooling solution**.

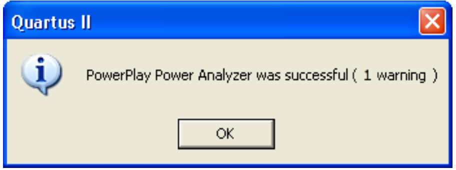
For more information about how to use the operating condition settings, refer to "Operating Conditions" on page 12–8.

26. Click **OK** to close the **Settings** dialog box.
27. On the Processing menu, click **PowerPlay Power Analyzer Tool**. The **PowerPlay Power Analyzer Tool** dialog box appears.
28. Click **Start** to run the PowerPlay Power Analyzer. Be sure that all the settings are correct.

You can also make changes to some of your settings in this dialog box. For example, you can click the **Add Power Input File(s)** button to make changes to your input file(s), or you can click the **Cooling Solution and Temperature** button to make changes to your design temperature and cooling solution selection.

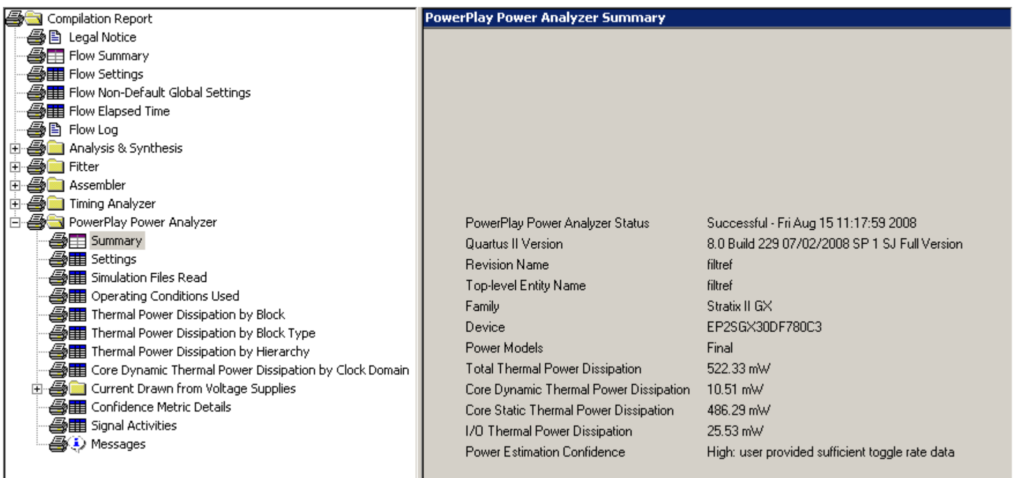
29. After the PowerPlay Power Analyzer runs successfully, a message appears (Figure 12–13).

Figure 12–13. PowerPlay Power Analyzer Message



30. Click **OK**.
31. In the **PowerPlay Power Analyzer Tool** dialog box, click **Report** to open the PowerPlay Power Analyzer Summary window. You can also view the summary in the **PowerPlay Power Analyzer Summary** page of the Compilation Report (Figure 12–14).

Figure 12–14. PowerPlay Power Analyzer Summary



PowerPlay Power Analyzer Compilation Report

The PowerPlay Power Analyzer section of the Compilation Report is divided into the following sections.

Summary

This section of the report shows your design's estimated total thermal power consumption. This includes dynamic, static, and I/O thermal power consumption. The I/O thermal power consumption is the total I/O power contributed by both the V_{CCIO} power supplies and some portion of the V_{CCINT}. The report also includes a confidence metric that reflects the overall quality of the data sources for the signal activities. For example, a **Low** power estimation confidence value reflects that the user has provided insufficient toggle rate data, or most of the signal activity information used for power estimation is from default or vectorless estimation settings (see the PowerPlay Power Analyzer Confidence Metric report for details of the input data).

Settings

This section of the report shows your design's PowerPlay Power Analyzer settings information. This includes default input toggle rates, operating conditions, and other relevant setting information.

Simulation Files Read

This section of the report lists simulation output files (**.vcd** or **.saf** file) used for power estimation. It also includes the file ID, file type, entity, VCD start time, VCD end time, the unknown %, and the toggle %. The unknown % indicates the portion of the design module that is not exercised by the simulation vectors.

Operating Conditions Used

This section of the report shows device characteristics, voltages, temperature, and cooling solution, if any, that were used during the power estimation. It also shows the entered junction temperature or auto-computed junction temperature that was used during the power analysis. This page is created only for Arria GX, Stratix IV, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families.

Thermal Power Dissipated by Block

This section of the report shows estimated thermal dynamic power and thermal static power consumption categorized by atoms. This information provides designers with an estimated power consumption for each atom in their design.

Thermal Power Dissipation by Block Type (Device Resource Type)

This section of the report shows the estimated thermal dynamic power and thermal static power consumption categorized by block types. This information is further categorized by estimated dynamic and static power that was used, as well as providing an average toggle rate by block type. Thermal power is the power dissipated as heat from the FPGA device.

Thermal Power Dissipation by Hierarchy

This section of the report shows an estimated thermal dynamic power and thermal static power consumption categorized by design hierarchy. This is further categorized by the dynamic and static power that was used by the blocks and routing within that hierarchy. This information is very useful in locating problem modules in your design.

Core Dynamic Thermal Power Dissipation by Clock Domain

This section of the report shows the estimated total core dynamic power dissipation by each clock domain. This provides designs with estimated power consumption for each clock domain in their design. If the clock frequency for a domain is unspecified by a constraint, the clock frequency is listed as “unspecified.” For all the combinational logic, the clock domain is listed as no clock with 0 MHz.

Current Drawn from Voltage Supplies

This section of the report lists the current that was drawn from each voltage supply. The VCCIO voltage supply is further categorized by I/O bank and by voltage. The minimum safe power supply size (current supply ability) is also listed for each supply voltage. This page is created only for Arria GX, Stratix IV, Stratix III, Stratix II, Stratix II GX, Cyclone III, Cyclone II, HardCopy II, and MAX II device families.

The transceiver-based devices have multiple voltage supplies: V_{CCH}, V_{CCT}, V_{CCR}, V_{CCA}, V_{CCP}, and so on. The report also shows the static and dynamic current (in mA) drawn from each voltage supply. Total static and dynamic power consumed by the transceivers on all voltage supplies is listed under the “Thermal Power Dissipation by Block Type” report section, which contains a row that starts with “GXB Transceiver.”

The I/O thermal power dissipation which is listed on the summary page does not correlate directly to the power drawn from the V_{CCIO} voltage supply listed in this report. This is because the I/O thermal power dissipation value also includes portions of the V_{CCINT} power, such as the I/O element (IOE) registers which are modeled as I/O power, but do not draw from the V_{CCIO} supply.

Confidence Metric Details

The confidence metric indicates the quality of the signal toggle rate data used to compute a power estimate. The confidence metric is low if the signal toggle rate data comes from sources that are considered poor predictors of real signal toggle rates in the device during an operation. Toggle rate data that comes from simulation, user-entered assignments on specific signals, or entities are considered reliable. Toggle rate data from default toggle rates (for example, 12.5% of the clock period) or vectorless estimation are considered relatively inaccurate. This section gives an overall confidence rating in the toggle rate data, from low to high. It also summarizes how many pins, registers, and combinational nodes obtained their toggle rates from each of simulation, user entry, vectorless estimation, or default toggle rate estimations. This detailed information can help you understand how to increase the confidence metric, letting you decide on your own confidence in the toggle rate data.

Signal Activities

This section lists toggle rate and static probabilities assumed by power analysis for all signals with fan-out and pins. The signal type is provided (Pin, Registered, or Combinational), as well as the data source for the toggle rate and static probability. By default, all signal activities are reported. This can be turned off on the **PowerPlay Power Analyzer Settings** page by turning off the **Write signal activities to report file** option.

Turning this option off might be advisable for a large design because of the large number of signals present. You can use the Assignment Editor to specify that activities for individual nodes or entities are reported by assigning an on value to those nodes for the Power Report Signal Activities assignment.

Messages

This section lists messages generated by the Quartus II software during the analysis.

Specific Rules for Reporting

In a Stratix GX device, the XGM II State Machine block is always used together with GXB transceivers, so its power is lumped into the power for the transceivers. Therefore, the power for the XGM II State Machine block is reported as 0 Watts.

Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF format.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Running the PowerPlay Power Analyzer from the Command Line

The separate executable that can be used to run the PowerPlay Power Analyzer is `quartus_pow`. For a complete listing of all command line options supported by `quartus_pow`, type the following at a system command prompt:

`quartus_pow --help` or `quartus_sh --qhelp` ↵

The following is an example of using the `quartus_pow` executable with project **sample.qpf**:

- To instruct the PowerPlay Power Analyzer to generate a PowerPlay Early Power Estimator file, type the following at a system command prompt:

```
quartus_pow sample --output_epe=sample.csv ↵
```

- To instruct the PowerPlay Power Analyzer to generate a PowerPlay Early Power Estimator file without doing the power estimate, type the following command at a system command prompt:

```
quartus_pow sample --output_epe=sample.csv --estimate_power=off ↵
```

- To instruct the PowerPlay Power Analyzer to use a **.saf** file as input (**sample.saf**), type the following at a system command prompt:

```
quartus_pow sample --input_saf=sample.saf ↵
```

- To instruct the PowerPlay Power Analyzer to use two **.vcd** files as input (**sample1.vcd** and **sample2.vcd**), perform glitch filtering on the **.vcd** file, and use a default input I/O toggle rate of 10,000 transitions/second, type the following at a system command prompt:

```
quartus_pow sample --input_vcd=sample1.vcd --input_vcd=sample2.vcd \
--vcd_filter_glitches=on --\
default_input_io_toggle_rate=10000transitions/s ↵
```

- To instruct the PowerPlay Power Analyzer to not use any input file, a default input I/O toggle rate of 60%, no vectorless estimation, and a default toggle rate of 20% on all remaining signals, type the following at a system command prompt:

```
quartus_pow sample --no_input_file --default_input_io_toggle_rate=60% \
--use_vectorless_estimation=off --default_toggle_rate=20% ↵
```

There are no command line options to specify the information found on the **PowerPlay Power Analyzer Settings Operating Conditions** page. The easiest way to specify these options is to use the Quartus II GUI.

A report file, <*revision name*>**.pow.rpt**, is created by the `quartus_pow` executable and saved in the main project directory. The report file contains the same information as described in the "PowerPlay Power Analyzer Compilation Report" on page 12–27.

Conclusion

PowerPlay power analysis tools are designed for accurate estimation of power consumption from early design concept through design implementation. Designers can use the PowerPlay Early Power Estimator to estimate power consumption during the design concept stage. Power estimations can be refined during design implementation using the Quartus II PowerPlay Power Analyzer feature. The Quartus II PowerPlay Power Analyzer produces detailed reports that you can use to optimize designs for lower power consumption and verify that the design is within your power budget.

Referenced Documents

This chapter references the following documents:

- *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Quartus II Simulator* chapter in volume 3 of the *Quartus II Handbook*
- *Section I. Simulation* in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 12–5 shows the revision history for this chapter.

Table 12–5. Document Revision History

Date and Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ This chapter was chapter 11 in version 8.1. ■ Removed Figures 11-10, 11-11, 11-13, 11-14, and 11-17 from 8.1 version.	Updated for the Quartus II software version 9.0.
November 2008 v8.1.0	■ Updated for the Quartus II software version 8.1. ■ Replaced Figure 11-3. ■ Replaced Figure 11-14.	Updated for the Quartus II software version 8.1.
May 2008 v8.0.0	■ Updated Figure 11–5. ■ Updated “Types of Power Analyses” on page 11–5. ■ Updated “Operating Conditions” on page 11–9. ■ Updated “PowerPlay Power Analyzer Compilation Report” on page 11–31. ■ Updated “Current Drawn from Voltage Supplies” on page 11–32.	Updated for the Quartus II software version 8.0.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

Introduction

Debugging today's FPGA designs can be a daunting task. As your product requirements continue to increase in complexity, the time you spend on design verification continues to rise. To get your product to market as quickly as possible, you must minimize design verification time. To help alleviate the time-to-market pressure, a set of verification tools that are powerful and easy to use are required.

The Quartus® II software provides a portfolio of in-system design debugging tools for real-time verification of your design. Each tool in the on-chip debugging portfolio uses a combination of available memory, logic, and routing resources to assist in the debugging process. The tools provide visibility by routing (or "tapping") signals in your design to debugging logic. The debugging logic is then compiled with your design and downloaded into the FPGA or CPLD for analysis. Because different designs can have different constraints and requirements, such as the number of spare pins available or the amount of logic or memory resources remaining in the physical device, you can choose a tool from the available debugging tools that matches the specific requirements for your design.

This section provides a quick overview on the tools available in the on-chip debugging suite and discusses the criteria for selecting the best tool for your design.

On-Chip Debugging Ecosystem

Table IV–1 summarizes the tools in the In-System verification tool suite that are covered in this section of the *Quartus II Handbook*.

Table IV–1. Available Tools in the In-System Verification Tools Suite (Part 1 of 2)

Tool	Description	Typical Circumstances of Use
SignalTap® II Logic Analyzer	This embedded logic analyzer uses FPGA resources to sample tests nodes and outputs the information to the Quartus II software for display and analysis.	You have spare on-chip memory and you want functional verification of your design running in hardware.
SignalProbe	This tool incrementally routes internal signals to I/O pins while preserving results from your last place-and-routed design.	You have spare I/O pins and you would like to check the operation of a small set of control pins using either an external logic analyzer or an oscilloscope.
Logic Analyzer Interface (LAI)	This tool multiplexes a larger set of signals to a smaller number of spare I/O pins. LAI allows you to select which signals are switched onto the I/O pins over a JTAG connection.	You have limited on-chip memory, and have a large set of internal data buses that you would like to verify using an external logic analyzer. Logic analyzer vendors, such as Tektronics and Agilent, provide integration with the tool to improve the usability.
In-System Memory Content Editor	This tool displays and allows you to edit on-chip memory.	You would like to view and edit the contents of the either the instruction cache or data cache of a Nios® II processor application.

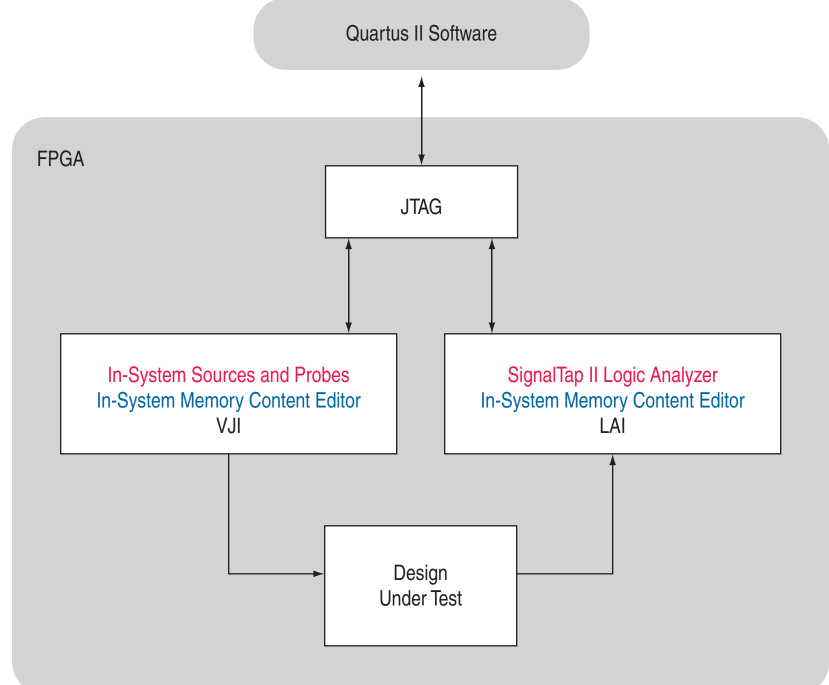
Table IV–1. Available Tools in the In-System Verification Tools Suite (Part 2 of 2)

Tool	Description	Typical Circumstances of Use
In-System Sources and Probes	This feature provides an easy way to drive and sample logic values to and from internal nodes using the JTAG interface.	You want to prototype a front panel with virtual buttons for your FPGA design.
Virtual JTAG Interface	This megafunction opens up the JTAG interface so that you can develop your own custom applications.	You want to generate a large set of test vectors and send them to your device over the JTAG port to functionally verify your design running in hardware.

With the exception of SignalProbe, each of the on-chip debugging tools uses the JTAG port to control and read back data from debugging logic and signals under test. The JTAG resource is shared among all of the on-chip debugging tools. The Quartus II software compiles logic into your design automatically to distinguish between data and control information and each of the debugging logic blocks when the JTAG resource is required. This arbitration logic, also known as the System-Level Debugging (SLD) infrastructure, is shown in the design hierarchy of your compiled project as **sld_hub:sld_hub_inst**. The SLD logic allows you to instantiate multiple debugging blocks into your design and run them simultaneously.

To maximize debugging closure, the Quartus II software allows you to use a combination of the debugging tools in tandem to fully exercise and analyze the logic under test. All of the tools described in Table IV–1 have basic analysis features built in; that is, all of the tools enable you to read back information collected from the design nodes that are connected to the debugging logic. Out of the set of debugging tools, the SignalTap II Logic Analyzer, the LAI, and the SignalProbe feature are general-purpose debugging tools optimized for probing signals in your RTL netlist. In-System Sources and Probes, the Virtual JTAG Interface, and In-System Memory content editor, in addition to being able to read back data from the debugging points, allow you to input values into your design during runtime. Taken together, the set of on-chip debugging tools form a debugging ecosystem. The set of tools can generate a stimulus to and solicit a response from the logic under test, providing a complete debugging solution (Figure IV–1).

Figure IV–1. Quartus II Debugging Ecosystem *(Note 1)*



Note to Figure IV–1:

(1) The set of debugging tools offer end-to-end debugging coverage.

The tools in the toolchain offer different advantages and different trade-offs. To understand the selection criteria between the different tools, the following sections analyze the tools according to their typical applications.

The first section, "Analysis Tools for RTL Nodes", compares the SignalTap II Logic Analyzer, SignalProbe, and the LAI. These three tools are logically grouped since they are intended for debugging nodes from your RTL netlist at system speed.

The next section, "Stimulus-Capable Tools" on page 13–8, compares the In-System Memory Content Editor, Virtual JTAG Interface Megafunction, and In-System Sources and Probes. These tools are logically grouped since they offer the ability to both read and write transactions through the JTAG port.

Analysis Tools for RTL Nodes

The SignalTap II Embedded Logic Analyzer, the SignalProbe feature, and the LAI are designed specifically for probing and debugging RTL signals at system speed. They are general-purpose analysis tools that enable you to tap and analyze any routable node from the FPGA or CPLD. These three tools satisfy a range of requirements. If you have spare logic and memory resources, the SignalTap II Logic Analyzer is useful for providing fast functional verification of your design running on actual hardware.

On the other hand, if logic and memory resources are tight and you require the large sample depths associated with external logic analyzers, both the LAI and the SignalProbe feature make it easy to view internal design signals using external equipment.

The most important selection criteria for these three tools are the available resources remaining on your device after implementing your design and the number of spare pins available. It is worthwhile to evaluate your preferred debugging option early on in the design planning process to ensure that your board, your Quartus II project, and your design are all set up to support the appropriate options. Planning early can reduce time spent during debugging and eliminate the necessary late changes to accommodate your preferred debugging methodologies. The following two sections provide information to assist you in choosing the appropriate tool by comparing the tools according to their resource usage and their pin consumption.

The SignalTap II Logic Analyzer is not supported on CPLDs, because there are no memory resources available on these devices.

Resource Usage

Any debugging tool that requires the use of a JTAG connection requires the SLD infrastructure logic mentioned earlier, for communication with the JTAG interface and arbitration between any instantiated debugging modules. This overhead logic uses around 200 LEs, a small fraction of the resources available in any of the supported devices. The overhead logic is shared between all available debugging modules in your design. Both the SignalTap II Logic Analyzer and the LAI use a JTAG connection.

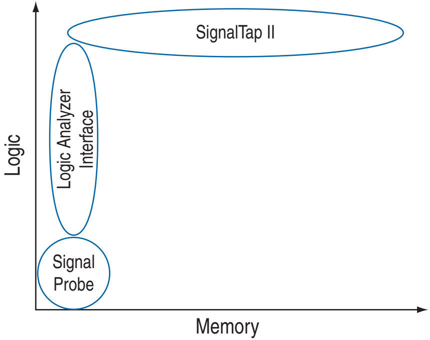
SignalProbe requires very few on-chip resources. Because it requires no JTAG connection, SignalProbe uses no logic or memory resources—it uses only routing resources to route an internal signal to a debugging test point.

The LAI requires a small amount of logic to implement the multiplexing function between the signals under test, in addition to the SLD infrastructure logic. Because no data samples are stored on the chip, the LAI uses no memory resources.

The SignalTap II Logic Analyzer requires both logic and memory resources. The number of logic resources used depends on the number of signals tapped and the complexity of the trigger logic. However, the amount of Logic Resources that the SignalTap II Logic Analyzer uses is typically a small percentage of most designs. A baseline configuration consisting of the SLD arbitration logic and a single node with basic triggering logic contains approximately 300–400 logic elements (LEs). Each additional node you add to the baseline configuration adds about 11 LEs. Compared with logic resources, memory resources are a more important factor to consider for your design. Memory usage can be significant and depends on how you configure your SignalTap II Logic Analyzer instance to capture data and the sample depth that your design requires for debugging. For the SignalTap II Logic Analyzer, there is the added benefit of requiring no external equipment, as all of the triggering logic and storage is on the chip.

Figure IV–2 shows a conceptual graph of the resource usage of the three analysis tools relative to each other.

Figure IV–2. Resource Usage per Debugging Tool *(Note 1)*



Note to Figure IV–2:

(1) Though resource usage is highly dependent on the design, this graph provides a rough guideline for tool selection.

The resource estimation feature for the SignalTap II Logic Analyzer and the LAI allows you to quickly judge if enough on-chip resources are available before compiling the tool with your design. Figure IV–3 shows the resource estimation feature for the SignalTap II Logic Analyzer and the Logic Analyzer Interface.

Figure IV–3. Resource Estimator

Pin Usage

The ratio of the number of pins used to the number of signals tapped for the SignalProbe feature is one-to-one. Because this feature can consume free pins quickly, a typical application for this feature is for routing control signals to spare debugging pins for debugging.

The ratio of the number of pins used to the number of signals tapped for the LAI is many-to-one. It can map up to 256 signals to each debugging pin, depending on available routing resources. The control of the active signals that are mapped to the the spare I/O pins is performed via the JTAG port. The LAI is ideal for routing data buses to a set of test pins for analysis.

Other than the JTAG test pins, the SignalTap II Logic Analyzer uses no additional pins. All data is buffered using on-chip memory and communicated to the SignalTap II GUI via the JTAG test port.

Usability Enhancements

The SignalTap II Embedded Logic Analyzer, the SignalProbe feature, and the LAI tools can be added to your existing design with minimal effects. With the node finder, you can find signals to route to a debugging module without making any changes to your HDL. SignalProbe inserts signals directly from your post-fit database. The SignalTap II Logic Analyzer and LAI support inserting signals from both pre-synthesis and post-fit netlists. All three tools allow you to find and configure your debugging setup quickly. In addition, the Quartus II incremental compilation feature and the Quartus II incremental routing feature allow for a fast turnaround time for your programming file, increasing productivity and enabling fast debugging closure.

Both LAI and the SignalTap II Logic Analyzer support incremental compilation. With incremental compilation, you can add a SignalTap II Logic Analyzer instance or an LAI instance incrementally into your placed-and-routed design. This has the benefit of both preserving your timing and area optimizations from your existing design, and decreasing the overall compilation time when any changes are necessary during the debugging process. With incremental compilation, you can save up to 70% compile time of a full compilation.

SignalProbe uses the incremental routing feature. The incremental routing feature runs only the Fitter stage of the compilation. This also leaves your compiled design untouched, except for the newly routed node or nodes. With SignalProbe, you can save as much as 90% compile time of a full compilation.

As another productivity enhancement, all tools in the on-chip debugging tool set support scripting via the `quartus_stp` Tcl package. For the SignalTap II Logic Analyzer and the LAI, scripting enables user-defined automation for data collection while debugging in the lab.

In addition, the JTAG server allows you to debug a design that is running on a device attached to a PC in a remote location. This allows you to set up your hardware in the lab environment, download any new **.sof** files, and perform any analysis from your desktop.

Table IV–2 compares common debugging features between these tools and provides suggestions about which is the best tool to use for a given feature.

Table IV–2. Suggested On-Chip Debugging Tools for Common Debugging Features (Part 1 of 2) *(Note 1)*

Feature	SignalProbe	Logic Analyzer Interface (LAI)	SignalTap II Logic Analyzer	Description
Large Sample Depth	N/A	✓	—	An external logic analyzer used with the LAI has a bigger buffer to store more captured data than the SignalTap II Logic Analyzer. No data is captured or stored with SignalProbe.
Ease in Debugging Timing Issue	✓	✓	—	External equipment, such as oscilloscopes and Mixed Signal Oscilloscopes (MSOs), can be used with either LAI or SignalProbe used with the LAI to provide you with access to timing mode, enabling you to debug combined streams of data.

Table IV–2. Suggested On-Chip Debugging Tools for Common Debugging Features (Part 2 of 2) *(Note 1)*

Feature	SignalProbe	Logic Analyzer Interface (LAI)	SignalTap II Logic Analyzer	Description
Minimal Effect on Logic Design	✓	✓ *(2)*	✓ *(2)*	The LAI adds minimal logic to a design, requiring fewer device resources. The SignalTap II Logic Analyzer has little effect on the design, as it is set as a separate design partition. SignalProbe incrementally routes nodes to pins, not affecting the design at all.
Short Compile and Recompile Time	✓	✓ *(2)*	✓ *(2)*	SignalProbe attaches incrementally routed signals to previously reserved pins, requiring very little recompilation time to make changes to source signal selections. The SignalTap II Logic Analyzer and the LAI can take advantage of incremental compilation to refit their own design partitions to decrease recompilation time.
Triggering Capability	N/A	N/A	✓	The SignalTap II Logic Analyzer offers triggering capabilities that are comparable to commercial logic analyzers.
I/O Usage	—	—	✓	No additional output pins are required with the SignalTap II Logic Analyzer. Both the LAI and SignalProbe require I/O pin assignments.
Acquisition Speed	N/A	—	✓	The SignalTap II Logic Analyzer can acquire data at speeds of over 200 MHz. The same acquisition speeds are obtainable with an external logic analyzer used with the LAI, but signal integrity issues may limit this.
No JTAG Connection Required	✓	—	—	An FPGA design with the SignalTap II Logic Analyzer or the LAI requires an active JTAG connection to a host running the Quartus II software. SignalProbe does not require a host for debugging purposes.
No External Equipment Required	—	—	✓	The SignalTap II Logic Analyzer logic is completely internal to the programmed FPGA device. No extra equipment is required other than a JTAG connection from a host running the Quartus II software or the stand-alone SignalTap II software. SignalProbe and the LAI require the use of external debugging equipment, such as multimeters, oscilloscopes, or logic analyzers.

Notes to Table IV–2:

(1) ✓ indicates the recommended tools for the feature.
— indicates that while the tool is available for that feature, that tool may not give the best results.
N/A indicates that the feature is not applicable for the selected tool.

(2) When used with incremental compilation.

Stimulus-Capable Tools

The In-System Memory Content Editor, the In-System Sources and Probes, and the Virtual JTAG interface each enable you to use the JTAG interface as a general-purpose communication port. Though all three tools can be used to achieve the same results, there are some considerations that make one tool easier to use in certain applications than others. In-System Sources and Probes is ideal for toggling control signals. The In-System Memory Content Editor is useful for inputting large sets of test data. Finally, the Virtual JTAG megafunction is well suited for more advanced users who want to develop their own customized JTAG solution.

In-System Sources and Probes

In-System Sources and Probes is an easy way to access JTAG resources to both read and write to your design. You can start by instantiating a megafunction into your HDL. The megafunction contains source ports and probe ports for driving values into and sampling values from the signals that are connected to the ports, respectively. Transaction details of the JTAG interface are abstracted away by the megafunction. During runtime, a GUI displays each source and probe port by instance and allows you to read from each probe port and drive to each source port. The GUI makes this tool ideal for toggling a set of control signals during the debugging process.

A good application of In-System Sources and Probes is to use the GUI as a replacement for the push buttons and LEDs used during the development phase of a project. Furthermore, In-System Sources and Probes supports a set of scripting commands for reading and writing using `quartus_stp`. When used with the Tk toolkit, you can build your own graphical interfaces—a feature that is ideal for building a virtual front panel during the prototyping phase of the design.

In-System Memory Content Editor

The In-System Memory Content Editor allows you to quickly view and modify memory contents either through a GUI interface or through Tcl scripting commands. The In-System Memory Content Editor works by turning single-port RAM blocks into dual-port RAM blocks. One port is connected to your clock domain and data signals, and the other port is connected to the JTAG clock and data signals for editing or viewing.

Because you can modify a large set of data easily, a useful application for the In-System Memory Content Editor is to generate test vectors for your design. For example, you can instantiate a free memory block, connect the output ports to the logic under test (using the same clock as your logic under test on the system side), and create the glue logic for the address generation and control of the memory. At runtime, you can modify the contents of the memory using either a script or the In-System Memory Content Editor GUI and perform a burst transaction of the data contents in the modified RAM block synchronous to the logic being tested.

Virtual JTAG Interface Megafunction

The Virtual JTAG Interface megafunction provides the finest level of granularity for manipulating the JTAG resource. This megafunction allows you to build your own JTAG scan chain by exposing all of the JTAG control signals and configuring your JTAG Instruction Registers (IRs) and JTAG Data Registers (DRs). During runtime, you control the IR/DR chain through a Tcl API. This feature is meant for users who have a thorough understanding of the JTAG interface and want precise control over the number and type of resources used.

Conclusion

The Quartus II on-chip debugging tool suite allows you to reach debugging closure quickly by providing you a set of powerful analysis tools and a set of tools that open up the JTAG port as a general purpose communication interface. The Quartus II software further broadens the scope of applications by giving you a comprehensive Tcl/Tk API. With the Tcl/Tk API, you cannot only increase the level of automation for all of the analysis tools, but you can also build virtual front panel applications quickly early in the prototyping phase.

In addition, all of the on-chip debugging tools have a tight integration with the rest of the productivity features within the Quartus II software. The incremental compile and incremental routing features enable a fast turnaround time for programming file generation. The cross-probing feature allows you to find and identify nodes quickly. The SignalTap II Logic Analyzer, when used with the TimeQuest Timing Analyzer, is a best-in-class timing verification suite that allows fast functional and timing verification.

This section contains the detailed usage for each of the On-Chip Debugging tools. This section contains the following chapters:

- Chapter 13, Quick Design Debugging Using SignalProbe
- Chapter 14, Design Debugging Using the SignalTap II Embedded Logic Analyzer
- Chapter 15, In-System Debugging Using External Logic Analyzers
- Chapter 16, In-System Updating of Memory and Constants
- Chapter 17, Design Debugging Using In-System Sources and Probes

13. Quick Design Debugging Using SignalProbe

QII53008-9.0.0

Introduction

Hardware verification can be a lengthy and expensive process. The SignalProbe incremental routing feature helps reduce the hardware verification process and time-to-market for system-on-a-programmable-chip (SOPC) designs.

Easy access to internal device signals is important in the design or debugging process. The SignalProbe feature makes design verification more efficient by routing internal signals to I/O pins quickly without affecting the design. When you start with a fully routed design, you can select and route signals for debugging to either previously reserved or currently unused I/O pins.

SignalProbe feature is fully functional with Arria® GX, the Stratix® series, Cyclone® series, MAX® II, and APEX™ series device families.

This chapter is divided into two sections. If you are using the SignalProbe feature to debug your Stratix series, Cyclone series, and MAX II device, refer to "Debugging Using the SignalProbe Feature". If you are using the SignalProbe feature to debug your APEX series device, refer to "Using SignalProbe with the APEX Device Family" on page 13–12.

The Quartus® II software provides a portfolio of on-chip debugging solutions. For an overview and comparison of all of the tools available in the Quartus II software on-chip debugging tool suite, refer to *Section V. In-System Design Debugging* in volume 3 of the *Quartus II Handbook*.

Debugging Using the SignalProbe Feature

The SignalProbe feature allows you to reserve available pins and route internal signals to those reserved pins, while preserving the behavior of your design. SignalProbe is an effective debugging tool that provides visibility into your FPGA.

This section describes the SignalProbe process for the Stratix series, Cyclone series, and MAX II device families. Using SignalProbe with APEX devices is described in "Using SignalProbe with the APEX Device Family" on page 13–12. APEX devices do not support post-fit netlist changes made as engineering change orders (ECOs).

You can reserve pins for SignalProbe and assign I/O standards before or after a full compilation. Each SignalProbe-source to SignalProbe-pin connection is implemented as an ECO change that is applied to your netlist after a full compilation.

To route the internal signals to the device's reserved pins for SignalProbe, perform the following tasks:

1. Reserve the SignalProbe Pins, described on page 13–2.
2. Perform a Full Compilation, described on page 13–3.
3. Assign a SignalProbe Source, described on page 13–3.
4. Add Registers to the Pipeline Path to SignalProbe Pin, described on page 13–4.

5. Perform a SignalProbe Compilation, described on page 13–5.
6. Analyze the Results of the SignalProbe Compilation, described on page 13–5.

Reserve the SignalProbe Pins

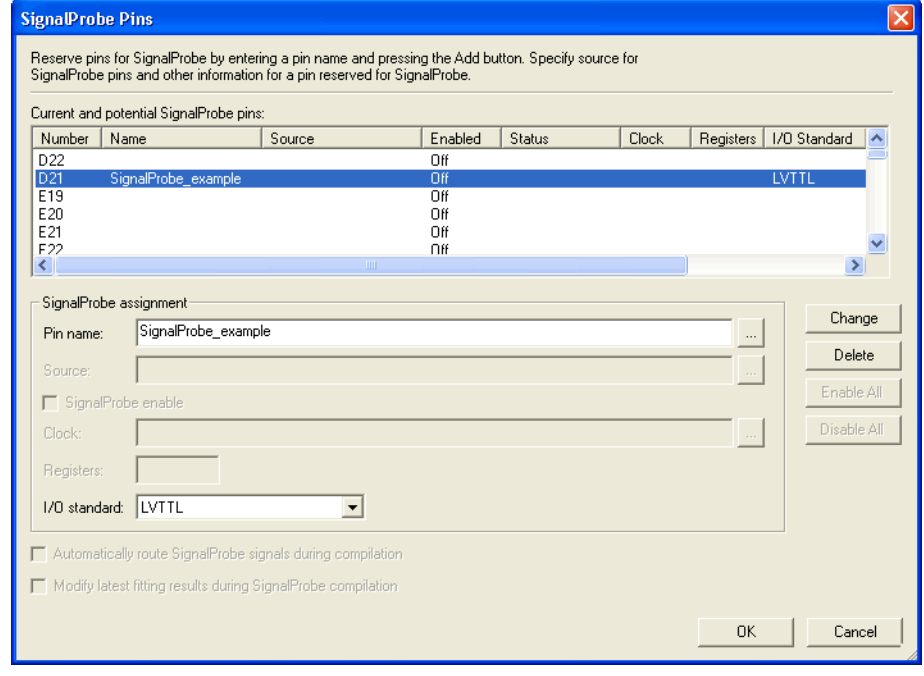
SignalProbe pins can be reserved before or after compiling your design. Reserving SignalProbe pins before a compilation is optional. You can also reserve any unused I/Os of the device for SignalProbe pins after compilation. Assigning sources is a simple process after reserving SignalProbe pins. The sources for SignalProbe pins are the internal nodes and registers in the post-compilation netlist that you want to probe.

Although you can reserve SignalProbe pins using many features within the Quartus II software, including the Pin Planner and the Tcl interface, you should use the **SignalProbe Pins** dialog box to create and edit your SignalProbe pins.

To reserve an available package pin as a SignalProbe pin using the **SignalProbe Pins** dialog box, perform the following steps:

1. On the Tools menu, click **SignalProbe Pins**. The **SignalProbe Pins** dialog box appears (Figure 13–1). The SignalProbe Pin name and I/O Standard appear as the only fields that are editable if place-and-route or fitting have not been performed.

Figure 13–1. Reserving a SignalProbe Pin in the SignalProbe Pins Dialog Box



2. In the **Current and potential SignalProbe pins** list, click a pin from the **Number** column and type your SignalProbe pin name in the **Pin name** box.
3. Select an I/O standard from the **I/O standard** drop-down list.

4. To add a new SignalProbe pin, click **Add**. To edit or change a previously reserved pin for SignalProbe, click **Change**. (Figure 13–1 shows the dialog box editing a previously reserved pin; if you were adding a new SignalProbe pin, the **Add** button appears instead of the **Change** button.)
5. Click **OK**.

Perform a Full Compilation

You must complete a full compilation to generate an internal netlist containing a list of internal nodes to probe to a SignalProbe outpin.

To perform a full compilation, on the processing menu, click **Start Compilation**.

Assign a SignalProbe Source

A SignalProbe source can be any combinational node, register, or pin in your post-compilation netlist. To find a SignalProbe source, in the Node Finder, use the SignalProbe filter to remove all sources that cannot be probed. You might not be able to find a particular internal node because the node can be optimized away during synthesis, or the node cannot be routed to the SignalProbe pin, as it is untappable. For example, internal nodes and registers within the Gigabit transceivers cannot be probed because there are no physical routes to the pins available.

To probe virtual I/O pins generated in low-level partitions in an incremental compilation flow, select the source of the logic that feeds the Virtual Pin as your SignalProbe source pin.

To assign a SignalProbe source to your SignalProbe reserved pin, perform the following steps:

1. On the Tools menu, click **SignalProbe Pins**. The **SignalProbe Pins** dialog box appears (Figure 13–1 on page 13–2).
2. If a SignalProbe reserved pin is shown, in the **Current and potential SignalProbe pins** list, click the pin. Alternately, you can click an available pin number in the **Current and potential SignalProbe pins** list and type a new SignalProbe pin name into the **Pin name** box.
3. In the **Source** box, specify the source name. Click the browse button. The **Node Finder** dialog box appears.
4. When you open the **Node Finder** dialog box from the **SignalProbe Pins** dialog box, **SignalProbe** is selected by default in the **Filter** list. To show a set of nodes that can be probed in the **Nodes Found** list, click **List**.
5. In the **Nodes Found** list, select your source node and click the **>** button. The selected node appears in the **Selected Nodes** list.
6. Click **OK**.
7. After a source is selected, the **SignalProbe enable** option is turned on. Click **Change** or **Add** to accept the changes.

Because SignalProbe pins are implemented and routed as ECOs, turning the **SignalProbe enable** option on or off is the same as selecting **Apply Selected Change** or **Restore Selected Change** in the Change Manager window. (If the Change Manager window is not visible at the bottom of your screen, on the View menu, point to **Utility Windows** and click **Change Manager**.)

For more information about the Change Manager for the Chip Planner and Resource Property Editor, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

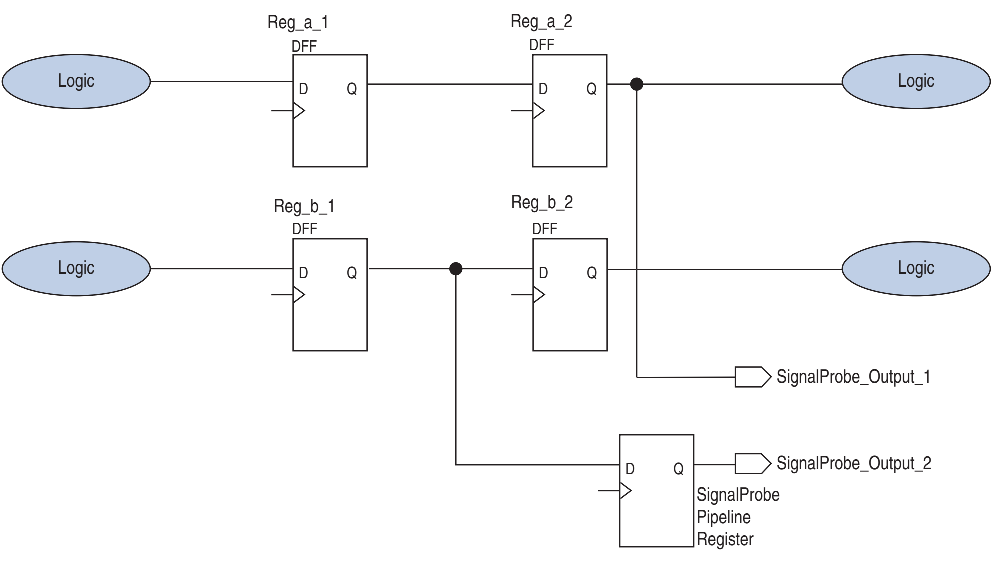
Add Registers to the Pipeline Path to SignalProbe Pin

You can specify the number of registers placed between a SignalProbe source and a SignalProbe pin to synchronize the data with a clock and to control the latency of the SignalProbe outputs. The SignalProbe feature automatically inserts the number of registers specified into the SignalProbe path.

Figure 13–2 shows a single register between the SignalProbe source `Reg_b_1` and SignalProbe `SignalProbe_Output_2` output pin added to synchronize the data between the two SignalProbe output pins.

When you add a register to a SignalProbe pin, the SignalProbe compilation attempts to place the register to best fit timing requirements. You can place SignalProbe registers either near the SignalProbe source to meet f_{MAX} requirements, or near the I/O to meet t_{CO} requirements.

Figure 13–2. Synchronizing SignalProbe Outputs with a SignalProbe Register



To pipeline an existing SignalProbe, perform the following steps:

1. On the Tools menu, click **SignalProbe Pins**. The **SignalProbe Pins** dialog box appears.
2. Select a SignalProbe pin and in the **Clock** dialog box, type the clock name used to drive your registers, or click the browse button to use the Node Finder to select your clock source.
3. In the **Registers** dialog box, specify the number of registers you want to add in between the SignalProbe source and the SignalProbe output.
4. Click **Change.**
5. Click **OK**.

In addition to clock input for pipeline registers, you can also specify a reset signal pin for pipeline registers. To specify a reset pin for pipeline registers, use the Tcl command `make_sp`, as described in "Scripting Support" on page 13–11.

Perform a SignalProbe Compilation

Perform a SignalProbe compilation to route your SignalProbe pins. A SignalProbe compilation saves and checks all netlist changes without recompiling the other parts of the design and completes compilation in a fraction of the time of a full compilation. The design's current placement and routing are preserved.

To perform a SignalProbe compilation, on the Processing menu, point to **Start** and click **Start SignalProbe Compilation**.

Analyze the Results of the SignalProbe Compilation

After a SignalProbe compilation, the results are available in the compilation report file. Each SignalProbe pin is displayed in the **SignalProbe Fitting Result** page in the **Fitter** section of the Compilation Report. To view the status of each SignalProbe pin in the **SignalProbe Pins** dialog box, on the Tools menu, click **SignalProbe Pins**.

The status of each SignalProbe pin appears in the Change Manager window (Figure 13–3). (If the Change Manager window is not visible at the bottom of your GUI, from the View menu, point to **Utility Windows** and click **Change Manager**.)

Figure 13–3. Change Manager Window with SignalProbe Pins

For more information about how to use the Change Manager, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

To view the timing results of each successfully routed SignalProbe pin, on the Processing menu, point to **Start** and click **Start Timing Analysis**.

SignalProbe ECO Flows

Beginning with Quartus II software version 6.0, SignalProbe pins are implemented using the same flow as other post-compilation changes made as ECOs. The following section describes SignalProbe ECO flows with and without the Quartus II incremental compilation feature.

SignalProbe ECO Flow with Quartus II Incremental Compilation

Beginning with Quartus II software version 6.1, the incremental compilation feature is turned on by default. The top-level design is automatically set to a design partition when the incremental compilation feature is on. A design partition during incremental compilation can have different netlist types. (Netlist types can be set to source HDL, post synthesis, or post-fit.) The netlist type indicates whether that partition should be resynthesized or refit during Quartus II incremental compilation. Incremental compilation saves you time and preserves the placement of unchanged partitions in your design if small changes must be made to some partitions late in the design cycle.

For more information about the Quartus II incremental compilation feature, refer to the *Quartus II Incremental Compilation Feature for Hiearchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

The behavior of SignalProbe pins during an incremental compilation depends on the Netlist Type setting. When the top-level partition netlist type is set to **post-fit**, SignalProbe ECOs are retained if the partition being probed is preserved when you recompile the design.

SignalProbe connections always link the partition being probed with the top-level partition. As such, a SignalProbe connection might change the preservation attributes in a lower-level partition. This is known as *partition linking*. When partition linking occurs, all partitions that become linked share the attribute for the preservation level that is the strictest among all of the affected partitions. As a result, when you tap any partitions that are not post-fit and the top level is set to a netlist type of post-fit**,** your SignalProbe connection is still preserved.

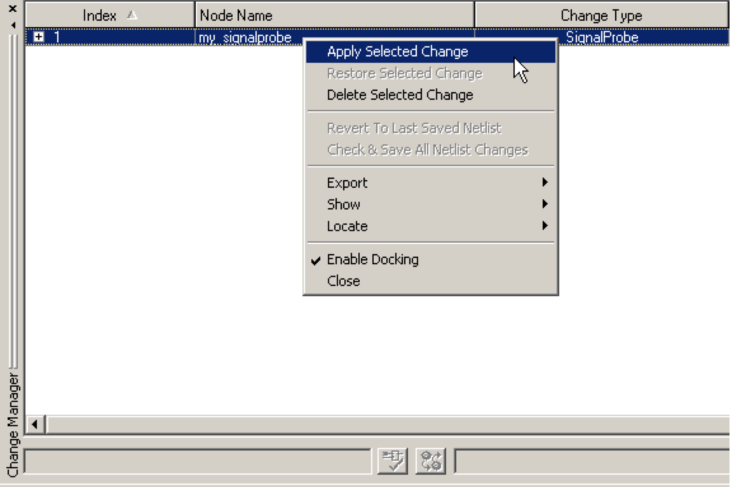
The behavior is different in the case that your top-level partition netlist type is set to **post-synthesis** and you have no other lower-level partitions defined. In this case, the partition with the strictest preservation type is set to **post-synthesis**. If you create SignalProbe ECOs and recompile the design, your SignalProbe ECOs are not retained and a warning message appears in the messages window. The warning indicates that ECO modifications are discarded; however, all of the ECO information is retained in the Change Manager. In this case, apply SignalProbe ECOs from the Change Manager and perform the **Check and Save All Netlist Changes** step, as described in “SignalProbe ECO Flow without Quartus Incremental Compilation” on page 13–7.

SignalProbe ECO Flow without Quartus Incremental Compilation

If you do not use the Quartus II incremental compilation feature and you implement SignalProbe pins after the initial compilation of your design, SignalProbe ECOs are not retained during recompilation. However, all of the SignalProbe ECOs remain in the Change Manager.

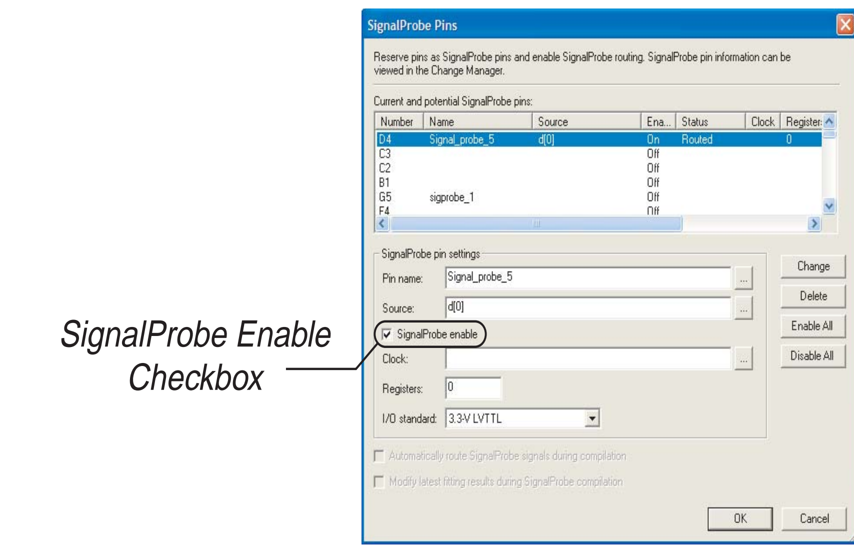
To apply a SignalProbe ECO, right-click Change Manager and select **Apply Selected Change** (Figure 13–4). (If the Change Manager window is not visible at the bottom of your screen, from the View menu, point to **Utility Windows** and click **Change Manager**.)

Figure 13–4. Applying SignalProbe ECOs



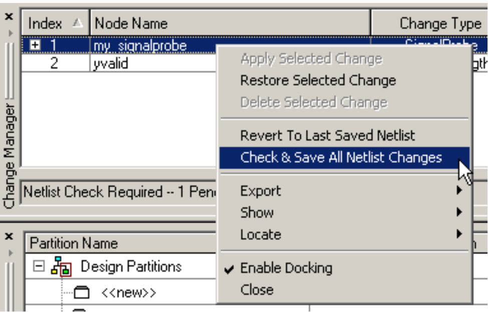
Alternately, you can use the **SignalProbe Pins** dialog box to enable the ECOs, as shown in Figure 13–5. This has the same effect as applying the SignalProbe ECOs within the Change Manager.

Figure 13–5. Enabling ECOs in the SignalProbe Pins Dialog Box



After applying the selected SignalProbe ECO, either right-click anywhere in the Change Manager and select **Check and Save All Netlist Changes** (Figure 13–6), or, from Processing menu, point to **Start** and click **Start Check and Save All Netlist Changes** to perform the ECO compilation.

Figure 13–6. Check and Save All Netlist Changes



Common Questions About the SignalProbe Feature

The following are answers to common questions about the SignalProbe feature.

Why Did I Get the Following Error Message, "Error: There are No Enabled SignalProbes to Process"?

This error message is generated when a SignalProbe compilation was attempted with either no SignalProbe pins to route, or with all SignalProbe pins disabled.

This might occur if you perform a SignalProbe compilation after a full compilation. For example, when a full compilation is performed, all SignalProbe pins are disabled. You can create or re-enable your SignalProbe pins in the **SignalProbe Pins** dialog box.

How Can I Retain My SignalProbe ECOs during Re-Compilation of My Design?

To retain your existing ECOs during recompilation of your design, you must use Quartus II incremental compilation. To learn more about the flow, refer to "SignalProbe ECO Flow with Quartus II Incremental Compilation" on page 13–6.

Why Did My SignalProbe Source Disappear in the Change Manager?

The SignalProbe source information for each SignalProbe is stored in the project database (**db** directory). SignalProbe pins are post-compilation changes to your netlist and are interpreted as ECOs. These changes are stored in the project **db** and if the project database is removed, the SignalProbe source information is lost and does not appear in the **SignalProbe Pins** dialog box. To restore your SignalProbe pins after the design compilation step, source the **signalprobe_qsf.tcl** script located in your project directory.

To restore your SignalProbe source information after compilation, type the following command from a command-line prompt:

```
quartus_cdb -t signalprobe_qsf.tcl ↵
```

Before typing this command, you must close your design project. When the command finishes, you can open your design project again. The change manager shows the sources for SignalProbe pins.

What is an ECO and Where Can I Find More Information about ECOs?

ECOs are late design cycle changes made to your design that do not alter functionality and timing.

For more information about ECOs and using the Change Manager, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

How Do I Migrate My Previous SignalProbe Assignments in the Quartus II Software Version 5.1 and Earlier to Version 6.0 and Later?

In earlier versions of the Quartus II software, SignalProbe pins were stored in the Quartus II Settings File (**.qsf**). These assignments are automatically converted into ECO changes when you open the **SignalProbe** dialog box or when you start a SignalProbe compilation in the Quartus II software versions 6.0 and higher.

For example, the SignalProbe source assignment from a **.qsf** file is removed and added to the Change Manager as an ECO after the **SignalProbe** dialog box is opened, or when you perform a SignalProbe compilation. Example 13–1 shows SignalProbe assignments in the **.qsf** file. Example 13–2 shows the same assignments after opening the **SignalProbe Pins** dialog box.

Example 13–1. SignalProbe Assignments in the Quartus II Settings File

```
set_location_assignment PIN_C22 -to my_signalprobe_pin
set_instance_assignment -name RESERVE_PIN "AS SIGNALPROBE OUTPUT" -to my_signalprobe_pin
set_instance_assignment -name IO_STANDARD LVTTL -to my_signalprobe_pin
set_instance_assignment -name SIGNALPROBE_ENABLE ON -to my_signalprobe_pin
set_instance_assignment -name SIGNALPROBE_SOURCE inst5[0] -to my_signalprobe_pin
```

Example 13–2. SignalProbe Assignments in the Quartus II Settings File after Opening the SignalProbe Pins Dialog Box

```
set_location_assignment PIN_C22 -to my_signalprobe_pin
set_instance_assignment -name RESERVE_PIN "AS SIGNALPROBE OUTPUT" -to my_signalprobe_pin
set_instance_assignment -name IO_STANDARD LVTTL -to my_signalprobe_pin
set_instance_assignment -name SIGNALPROBE_ENABLE ON -to my_signalprobe_pin
```

What are all the Changes for the SignalProbe Feature between the Quartus II Software Version 5.1 and Earlier, and Version 6.0 and Later?

The following list highlights the changes that affect users of the SignalProbe feature in the Quartus II software versions 5.1 and below. This applies to Stratix series, Cyclone series, and MAX II device families.

For more information about the changes that pertain to each release of the Quartus II software, refer to the Release Notes on the Altera website (www.altera.com).

- In Quartus II software versions 5.1 and earlier, the **SignalProbe Pins** dialog box was accessed on the Assignments menu. To access it with the Quartus II software version 6.0 and later, on the Tools menu, click **SignalProbe Pins**.

- A full compilation is required before making SignalProbe connections. However, you can still reserve pins before compilation for later use by SignalProbe. You can reserve pins by creating a SignalProbe in the **SignalProbe** dialog box without specifying a source. This is the same behavior as in the Quartus II software version 5.1.
- To route the SignalProbe pins, you must perform a SignalProbe compilation after a full compilation. The **Automatically route SignalProbe signals during compilations** and **Modify latest fitting results during SignalProbe compilation** options are no longer supported.
- After subsequent compiles, full or incremental, existing SignalProbe pins are disabled and are not present in the post-compilation netlist. To add them back, enable the SignalProbe pins and perform a SignalProbe compilation.
- SignalProbe pins are not controlled via assignments in the **.qsf** file because they are now ECOs. Existing **.qsf** files automatically convert to ECOs when a SignalProbe compilation is performed or when the **SignalProbe** dialog box is opened.
- The Tcl interface for creating SignalProbe pins has improved and is a part of the Chip Planner package `::quartus::chip_editor`. Refer to "Scripting Support" on page 13–11.
- Previously, the `quartus_fit --signalprobe` command was used to perform a SignalProbe compilation. This is not supported in the Quartus II software version 6.0 and later, and is replaced by the improved Tcl interface and the `check_netlist_and_save` Tcl command.
- The SignalProbe timing report generated after a successful SignalProbe compilation is not available in the Quartus II software version 6.0 and later. You can view the timing results of your SignalProbe pins in the SignalProbe Fitting Results, under the Fitter report, or in the t_{CO} results page of the Timing report.
- You cannot make SignalProbe pins in the Assignment Editor. Use the **SignalProbe Pins** dialog box to make and edit your SignalProbe pins.

Why Can't I Reserve a SignalProbe Pin?

If you cannot reserve a SignalProbe pin in the Quartus II software, it is likely that one of the following is true:

- You have selected multiple pins.
- A compile is running in the background. Wait until the compilation is complete before reserving the pin.
- You have the Quartus II Web Edition software, in which the SignalProbe feature is not enabled by default. You must turn on TalkBack to enable the SignalProbe feature in the Quartus II Web Edition software.
- You have not set the pin reserve type to **As Signal Probe Output**. To reserve a pin, on the Assignments menu, in the **Assign Pins** dialog box, select **As SignalProbe Output**.

- The pin is reserved from a previous compilation. During a compilation, the Quartus II software reserves each pin on the targeted device. If you end the Quartus II process during a compilation, for example, with the **Windows Task Manager End Process** command or the UNIX `kill` command, perform a full recompilation before reserving pins as SignalProbe outputs.
- The pin does not support the SignalProbe feature. Select another pin.
- The current family does not support the SignalProbe feature.

Scripting Support

Running procedures and make settings using a Tcl script are described in this chapter. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II command-line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Scripting Reference Manual* includes the same information in PDF format.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about all settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Reference Manual*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Make a SignalProbe Pin

To make a SignalProbe pin, type the following command:

```
make_sp [-h | -help] [-long_help] [-clk <clk>] [-io_std <io_std>] \
-loc <loc> -pin_name <pin name> [-regs <regs>] [-reset <reset>] \
-src_name <source name> ↵
```

Delete a SignalProbe Pin

To delete a SignalProbe pin, type the following command:

```
delete_sp [-h | -help] [-long_help] -pin_name <pin name> ↵
```

Enable a SignalProbe Pin

To enable a SignalProbe pin, type the following command:

```
enable_sp [-h | -help] [-long_help] -pin_name <pin name> ↵
```

Disable a SignalProbe Pin

To disable a SignalProbe pin, type the following command:

```
disable_sp [-h | -help] [-long_help] -pin_name <pin name> ↵
```

Perform a SignalProbe Compilation

To perform a SignalProbe compilation, type the following command:

```
check_netlist_and_save ↵
```

Migrate Previous SignalProbe Pins to the Quartus II Software Versions 6.0 and Later

To migrate previous SignalProbe pins to the Quartus II software versions 6.0 and later, type the following command:

```
convert_signal_probes ↵
```

Script Example

Example 13–3 shows a script that creates a SignalProbe pin called `sp1` and connects the `sp1` pin to source node `reg1` in a project that was already compiled.

Example 13–3. Creating a SignalProbe Pin Called sp1

```
Package require ::quartus::chip_editor
Project_open project
Read_netlist
Make_sp -pin_name sp1 -src_name reg1
Check_netlist_and_save
Project_close
```

Using SignalProbe with the APEX Device Family

APEX devices do not support post-fit netlist changes made as ECOs. SignalProbe compilation can route internal signals to output pins incrementally. The SignalProbe incremental routing feature does not affect design behavior.

To use the SignalProbe feature, follow these steps:

1. Reserve SignalProbe pins. For more information, refer to "Reserve the SignalProbe Pins" on page 13–2.
2. Assign a SignalProbe source to each SignalProbe pin.
3. Perform a SignalProbe compilation.
4. Analyze the results of a SignalProbe compilation.

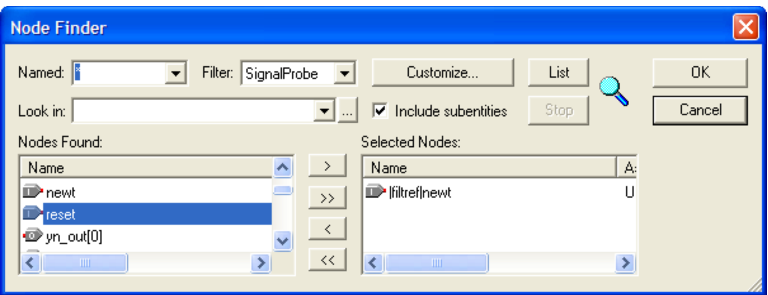
Adding SignalProbe Sources

A SignalProbe source is a signal in the post-compilation design database with a possible route to an output pin. To assign a SignalProbe source to a SignalProbe pin or an unused output pin, perform the following steps:

1. On the Tools menu, click **SignalProbe Pins**. The **SignalProbe Pins** dialog box appears.
2. In the **Current and potential SignalProbe pins** list, select the SignalProbe pin to which you want to add a SignalProbe source.
3. Click Browse and select a SignalProbe source.
4. Click **OK**.
5. In the **Assign SignalProbe Pins** dialog box, if a source has not been assigned to the SignalProbe pin, click **Add**. If a SignalProbe pin has been already assigned, click **Change**.
6. Click **OK.**

The **Node Finder** dialog box displays with the SignalProbe filter selected (Figure 13–7). Click **List** to view all of the available SignalProbe sources. If you cannot find a specific node with the SignalProbe filter, the node has either been removed by the Quartus II software during optimization or placed in the device where there are no possible routes to a pin.

Figure 13–7. Available SignalProbe Sources in the Node Finder



When the source of the SignalProbe pin is added or changed, the SignalProbe pin is automatically enabled. To disable a SignalProbe pin, turn off SignalProbe **enable**.

Performing a SignalProbe Compilation

After a full compilation, you can start a SignalProbe compilation either manually. A SignalProbe compilation performs the following functions:

- Validates SignalProbe pins
- Validates your specified SignalProbe sources
- If applicable, adds registers into SignalProbe paths
- Attempts to route from SignalProbe sources through registers to SignalProbe pins

To run the SignalProbe compilation automatically after a full compilation, on the Tools menu, click **SignalProbe Pins**. In the **SignalProbe Pins** dialog box, turn on **Automatically route SignalProbe signals during compilation**.

To run a SignalProbe compilation manually after a full compilation, on the Processing menu, point to **Start** and click **Start SignalProbe Compilation**.

You must run the Fitter before a SignalProbe compilation. The Fitter generates a list of all internal nodes that can be used as SignalProbe sources.

To enable or disable each SignalProbe pin, in the **SignalProbe Pins** dialog box, turn the **SignalProbe enable** option on or off.

Running SignalProbe with Smart Compilation

Running a smart compilation reduces compilation time by running only necessary modules during compilation. However, a full compilation is required if any design files, Analysis and Synthesis settings, or Fitter settings have changed.

To turn on smart compilation, on the Assignments menu, click **Settings**. In the **Category** list, select **Compilation Process Settings** and turn on **Use Smart compilation**.

If you run a SignalProbe compilation with smart compilation turned on, and there are changes to a design file or settings related to the Analysis and Synthesis or Fitter modules, the following message is displayed:

```
Error: Can't perform SignalProbe compilation because design requires a
full compilation.
```

You should turn smart compilation on, which allows you to work with the latest settings and design files.

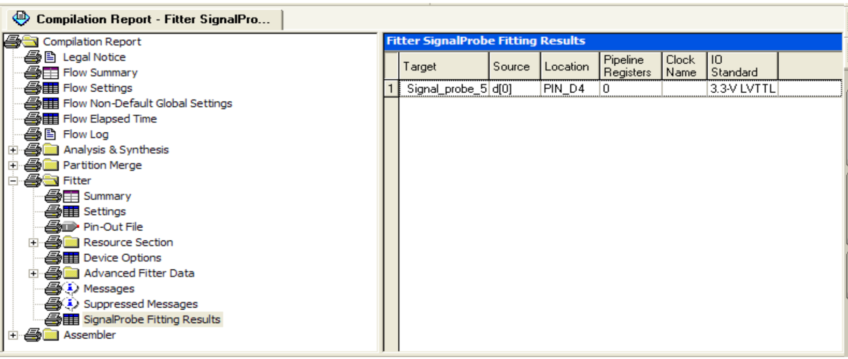
Understanding the Results of a SignalProbe Compilation

After a SignalProbe compilation, the results appear in two sections of the compilation report file. The fitting results and status (Table 13–1) of each SignalProbe pin is displayed in the **SignalProbe Fitting Result** screen in the Fitter section of the compilation report (Figure 13–8).

Table 13–1. Status Values

Status	Description
Routed	Connected and routed successfully
Not Routed	Not enabled
Failed to Route	Failed routing during last SignalProbe compilation
Need to Compile	Assignment changed since last SignalProbe compilation

Figure 13–8. SignalProbe Fitting Results Page in the Compilation Report Window



The timing results of each successfully routed SignalProbe pin is displayed in the **SignalProbe source to output delays** screen in the Timing Analysis section of the compilation report (Figure 13–9).

Figure 13–9. SignalProbe Source to Output Delays Page in the Compilation Report Window

Compilation Report

SignalProbe Source to Output Delays

	Source Name	Pin Location	Pin Name	Enable	Status	Delay (ns)
1	inst5[2]	Pin_H17	probe1	On	Routed	5.135 ns
2	inst5[3]	Pin_H20	probe2	On	Routed	4.939 ns
3	inst5[4]	Pin_G22	probe3	On	Routed	4.475 ns

After a SignalProbe compilation, the processing screen of the Messages window also provides the results of each SignalProbe pin and displays slack information for each successfully routed SignalProbe pin.

Analyzing SignalProbe Routing Failures

The SignalProbe can begin compilation; however, one of the following reasons can prevent complete compilation:

- **Route unavailable**—the SignalProbe compilation failed to find a route from the SignalProbe source to the SignalProbe pin because of routing congestion
- **Invalid or nonexistent SignalProbe source**—you entered a SignalProbe source that does not exist or is invalid
- **Unusable output pin**—the output pin selected is found to be unusable

Routing failures can occur if the SignalProbe pin's I/O standard conflicts with other I/O standards in the same I/O bank.

If routing congestion prevents a successful SignalProbe compilation, you can allow the compiler to modify routing to the specified SignalProbe source. On the Tools menu, click **SignalProbe Pins** and turn on **Modify latest fitting results during SignalProbe compilation**. This setting allows the Fitter to modify existing routing channels used by your design.

Turning on **Modify latest fitting results during SignalProbe compilation** can change the performance of your design.

SignalProbe Scripting Support for APEX Devices

Running procedures and make settings using a Tcl script are described in this chapter. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II command-line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Scripting Reference Manual* includes the same information in PDF format.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. Refer to the *Quartus II Settings File Reference Manual* for information about all settings and constraints in the Quartus II software. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Reserving SignalProbe Pins

To reserve a SignalProbe pin, type the commands shown in Example 13–4.

Example 13–4. Reserving a SignalProbe Pin

```
set_location_assignment <location> -to <SignalProbe pin name> ↵
set_instance_assignment -name RESERVE_PIN \
"AS SIGNALPROBE OUTPUT" -to <SignalProbe pin name> ↵
```

Valid locations are pin location names, such as `Pin_A3`.

For more information about reserving SignalProbe pins, refer to "Reserve the SignalProbe Pins" on page 13–2.

Adding SignalProbe Sources

Use the following Tcl commands to add SignalProbe sources. For more information about adding SignalProbe sources, refer to "Adding SignalProbe Sources" on page 13–12.

To assign the node name to a SignalProbe pin, type the following command:

```
set_instance_assignment -name SIGNALPROBE_SOURCE <node name> -to \
<SignalProbe pin name> ↵
```

The next command turns on SignalProbe routing. To turn off individual SignalProbe pins, specify `OFF` instead of `ON` with the following command:

```
set_instance_assignment -name SIGNALPROBE_ENABLE ON -to \
<SignalProbe pin name> ↵
```

Assigning I/O Standards

To assign an I/O standard to a pin, type the following Tcl command:

```
set_instance_assignment -name IO_STANDARD <I/O standard> -to \
<SignalProbe pin name> ↵
```

For a list of valid I/O standards, refer to the I/O Standards general description in the Quartus II Help.

Adding Registers for Pipelining

To add registers for pipelining, type the following Tcl commands:

```
set_instance_assignment -name SIGNALPROBE_CLOCK <clock name> -to \
<SignalProbe pin name> ↵

set_instance_assignment \
-name SIGNALPROBE_NUM_REGISTERS <number of registers> -to \
<SignalProbe pin name> ↵
```

Run SignalProbe Automatically

To run SignalProbe automatically after a full compile, type the following Tcl command:

```
set_global_assignment -name SIGNALPROBE_DURING_NORMAL_COMPILATION ON ↵
```

For more information about running SignalProbe automatically, refer to "Performing a SignalProbe Compilation" on page 13–13.

Run SignalProbe Manually

To run SignalProbe manually with a Tcl command or the `quartus_fit` command, type the following at a command prompt.

```
execute_flow -signalprobe ↵
```

The `execute_flow` command is in the flow package. At a command prompt, type the following command:

```
quartus_fit <project name> --signalprobe ↵
```

For more information about running SignalProbe manually, refer to "Performing a SignalProbe Compilation" on page 13–13.

Enable or Disable All SignalProbe Routing

Use the Tcl command in Example 13–5 to turn on or turn off SignalProbe routing. When using this command, to turn SignalProbe routing on, specify `ON`. To turn SignalProbe routing off, specify `OFF`.

Example 13–5. Turning SignalProbe On or Off with Tcl Commands

```
set spe [get_all_assignments -name SIGNALPROBE_ENABLE] \
foreach_in_collection asgn $spe {
   set signalprobe_pin_name [lindex $asgn 2]
   set_instance_assignment -name SIGNALPROBE_ENABLE -to \
$signalprobe_pin_name <ON|OFF> } ↵
```

For more information about enabling or disabling SignalProbe routing, refer to page 13–13.

Running SignalProbe with Smart Compilation

To turn on **Smart Compilation**, type the following Tcl command:

```
set_global_assignment -name SMART_RECOMPILE ON ↵
```

For more information, refer to "Running SignalProbe with Smart Compilation" on page 13–13.

Allow SignalProbe to Modify Fitting Results

To turn on **Modify latest fitting results**, type the following Tcl command:

```
set_global_assignment -name SIGNALPROBE_ALLOW_OVERUSE ON ↵
```

For more information, refer to "Analyzing SignalProbe Routing Failures" on page 13–15.

Conclusion

Using the SignalProbe feature can significantly reduce the time required compared to a full recompilation. Use the SignalProbe feature for quick access to internal design signals to perform system-level debugging.

Referenced Documents

This chapter references the following documents:

- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*
- Release Notes on the Altera website (www.altera.com)
- *Section V. In-System Design Debugging* in the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hiearchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 13–2 shows the revision history for this chapter.

Table 13–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Removed the "Generate the Programming File" section ■ Removed unnecessary screenshots ■ Minor editorial updates	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	■ Modified description for preserving SignalProbe connections when using Incremental Compilation ■ Added plausible scenarios where SignalProbe connections are not reserved in the design	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Added "Arria GX" to the list of supported devices ■ Removed the "On-Chip Debugging Tool Comparison" and replaced with a reference to the Section V Overview on page 13–1 ■ Added hyperlinks to referenced documents throughout the chapter ■ Minor editorial updates	Organizational changes for the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

14. Design Debugging Using the SignalTap II Embedded Logic Analyzer

QII53009-9.0.0

Introduction

To help with the process of design debugging, Altera provides a solution that allows you to examine the behavior of internal signals, without using extra I/O pins, while the design is running at full speed on an FPGA device.

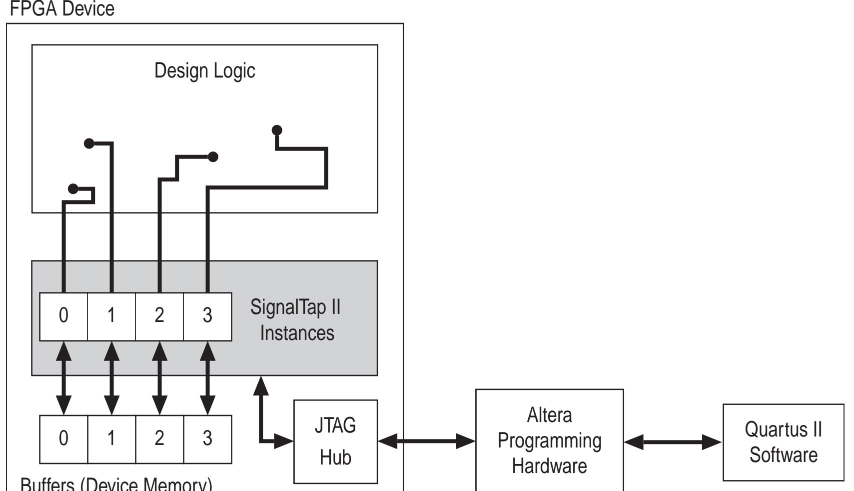
The SignalTap® II Embedded Logic Analyzer is scalable, easy to use, and is included with the Quartus® II software subscription. This logic analyzer helps debug an FPGA design by probing the state of the internal signals in the design without the use of external equipment. Defining custom trigger-condition logic provides greater accuracy and improves the ability to isolate problems. The SignalTap II Embedded Logic Analyzer does not require external probes or changes to the design files to capture the state of the internal nodes or I/O pins in the design. All captured signal data is conveniently stored in device memory until you are ready to read and analyze the data.

The topics in this chapter include:

- "Design Flow Using the SignalTap II Embedded Logic Analyzer" on page 14–4
- "SignalTap II Embedded Logic Analyzer Task Flow" on page 14–4
- "Add the SignalTap II Embedded Logic Analyzer to Your Design" on page 14–6
- "Configure the SignalTap II Embedded Logic Analyzer" on page 14–14
- "Define Triggers" on page 14–33
- "Compile the Design" on page 14–53
- "Program the Target Device or Devices" on page 14–59
- "Run the SignalTap II Embedded Logic Analyzer" on page 14–60
- "View, Analyze, and Use Captured Data" on page 14–66
- "Other Features" on page 14–71
- "SignalTap II Scripting Support" on page 14–76
- "Design Example: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems" on page 14–79
- "Custom Triggering Flow Application Examples" on page 14–79

The SignalTap II Embedded Logic Analyzer is a next-generation, system-level debugging tool that captures and displays real-time signal behavior in a system-on-a-programmable-chip (SOPC) or any FPGA design. The SignalTap II Embedded Logic Analyzer supports the highest number of channels, largest sample depth, and fastest clock speeds of any embedded logic analyzer in the programmable logic market. Figure 14–1 shows a block diagram of the components that make up the SignalTap II Embedded Logic Analyzer.

Figure 14–1. SignalTap II Embedded Logic Analyzer Block Diagram *(Note 1)*



Note to Figure 14–1:

(1) This diagram assumes that the SignalTap II Embedded Logic Analyzer was compiled with the design as a separate design partition using the Quartus II incremental compilation feature. This is the default setting for new projects in the Quartus II software. If incremental compilation is disabled or not used, the SignalTap II logic is integrated with the design. For information about the use of incremental compilation with SignalTap II, refer to "Faster Compilations with Quartus II Incremental Compilation" on page 14–53.

This chapter is intended for any designer who wants to debug their FPGA design during normal device operation without the need for external lab equipment. Because the SignalTap II Embedded Logic Analyzer is similar to traditional external logic analyzers, familiarity with external logic analyzer operations is helpful but not necessary. To take advantage of faster compile times when making changes to the SignalTap II Embedded Logic Analyzer, knowledge of the Quartus II incremental compilation feature is helpful.

For information about using the Quartus II incremental compilation feature, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Hardware and Software Requirements

The following components are required to perform logic analysis with the SignalTap II Embedded Logic Analyzer:

- Quartus II design software

or
Quartus II Web Edition (with the TalkBack feature enabled)
or
SignalTap II Embedded Logic Analyzer standalone software

- Download/upload cable
- Altera® development kit or user design board with JTAG connection to device under test

The Quartus II software Web Edition does not support the SignalTap II Embedded Logic Analyzer with the incremental compilation feature.

Captured data is stored in the device's memory blocks and transferred to the Quartus II software waveform display with a JTAG communication cable, such as EthernetBlaster or USB-Blaster™. Table 14–1 summarizes some of the features and benefits of the SignalTap II Embedded Logic Analyzer.

Table 14–1. SignalTap II Features and Benefits

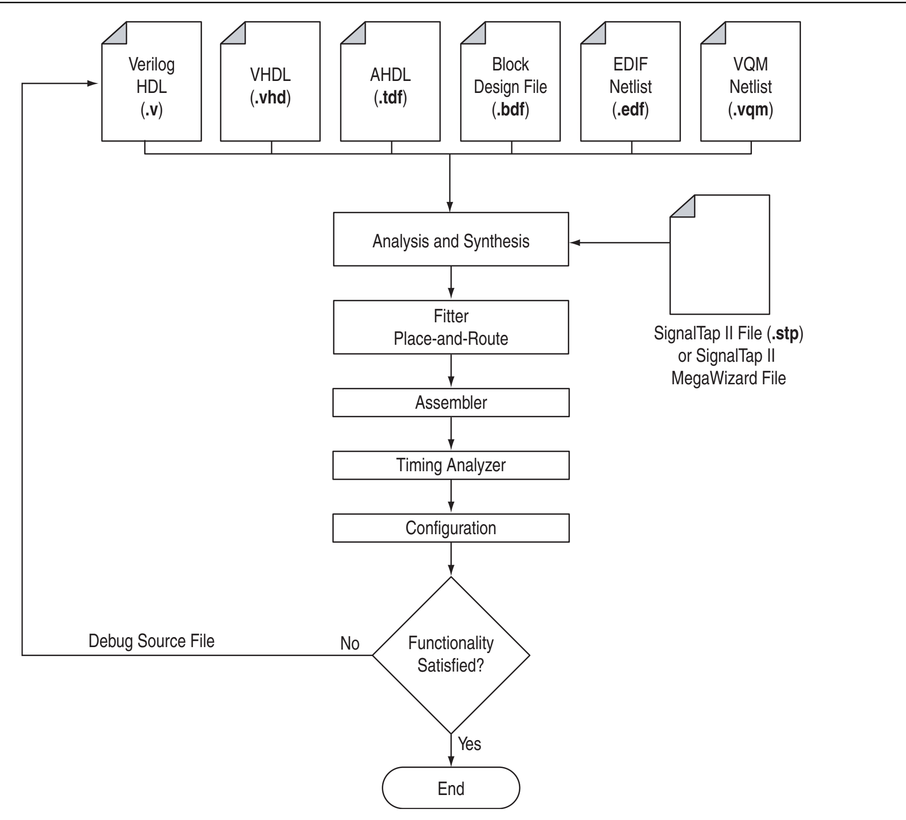
Feature	Benefit
Multiple logic analyzers in a single device	Captures data from multiple clock domains in a design at the same time.
Multiple logic analyzers in multiple devices in a single JTAG chain	Simultaneously captures data from multiple devices in a JTAG chain.
Plug-In Support	Easily specifies nodes, triggers, and signal mnemonics for IP, such as the Nios® II embedded processor.
Up to 10 basic or advanced trigger conditions for each analyzer instance	Enables more complex data capture commands to be sent to the logic analyzer, providing greater accuracy and problem isolation.
Power-Up Trigger	Captures signal data for triggers that occur after device programming but before manually starting the logic analyzer.
State-based Triggering Flow	Enables you to organize your triggering conditions to precisely define what your embedded logic analyzer will capture.
Incremental compilation	Modifies the SignalTap II Embedded Logic Analyzer monitored signals and triggers without performing a full compilation, saving time.
Flexible buffer acquisition modes	The buffer acquisition control allows you to precisely control the data that is written into the acquisition buffer. Both segmented buffers and non-segmented buffers with storage qualification allow you to discard data samples that are not relevant to the debug of your design.
MATLAB integration with included MEX function	Collects the SignalTap II Embedded Logic Analyzer captured data into a MATLAB integer matrix.
Up to 2,048 channels per logic analyzer instance	Samples many signals and wide bus structures.
Up to 128K samples in each device	Captures a large sample set for each channel.
Fast clock frequencies	Synchronous sampling of data nodes using the same clock tree driving the logic under test.
Resource usage estimator	Provides estimate of logic and memory device resources used by SignalTap II Embedded Logic Analyzer configurations.
No additional cost	The SignalTap II Embedded Logic Analyzer is included with a Quartus II subscription and with the Quartus II Web Edition (with TalkBack enabled).
Compatibility with other on-chip debugging utilities	The SignalTap II Embedded Logic Analyzer can be used in tandem with any JTAG based on-chip debugging tool, such as an in-system memory content editor. This ability to share the JTAG chain allows you to change signal values in real-time while you are running an analysis with the SignalTap II Embedded Logic Analyzer.

The Quartus II software offers a portfolio of on-chip debugging solutions. For an overview and comparison of all of the tools available in the In-System Verification Tool set, refer to *Section V. In-System Design Debugging*.

Design Flow Using the SignalTap II Embedded Logic Analyzer

Figure 14–2 shows a typical overall FPGA design flow for using the SignalTap II Embedded Logic Analyzer in your design. A SignalTap II file (**.stp**) is added to and enabled in your project, or a SignalTap II HDL function, created with the MegaWizard™ Plug-In Manager, is instantiated in your design. The diagram shows the flow of operations from initially adding the SignalTap II Embedded Logic Analyzer to your design to final device configuration, testing, and debugging.

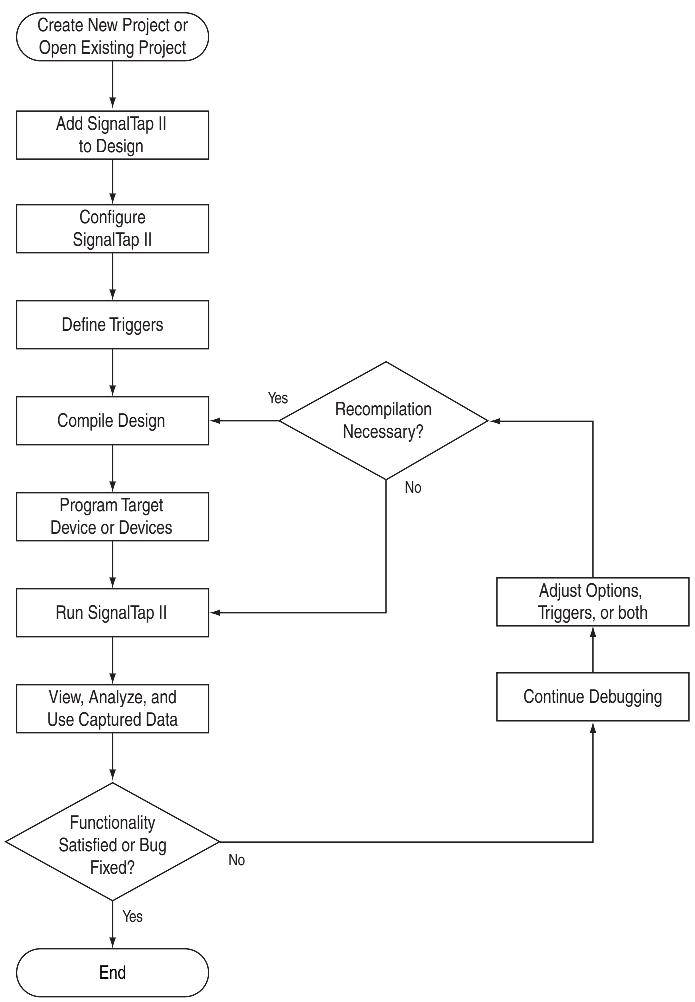
Figure 14–2. SignalTap II FPGA Design and Debugging Flow



SignalTap II Embedded Logic Analyzer Task Flow

To use the SignalTap II Embedded Logic Analyzer to debug your design, you perform a number of tasks to add, configure, and run the logic analyzer. Figure 14–3 shows a typical flow of the tasks you complete to debug your design. Refer to the appropriate section of this chapter for more information about each of these tasks.

Figure 14–3. SignalTap II Embedded Logic Analyzer Task Flow



Add the SignalTap II Embedded Logic Analyzer to Your Design

Create an **.stp** file or create a parameterized HDL instance representation of the logic analyzer using the MegaWizard Plug-In Manager. If you want to monitor multiple clock domains simultaneously, add additional instances of the logic analyzer to your design, limited only by the available resources in your device.

Configure the SignalTap II Embedded Logic Analyzer

After the SignalTap II Embedded Logic Analyzer is added to your design, configure it to monitor the signals you want. You can manually add signals or use a plug-in, such as the Nios II embedded processor plug-in, to quickly add entire sets of associated signals for a particular intellectual property (IP). You can also specify settings for the data capture buffer, such as its size, the method in which data is captured and stored, and the device memory type to use for the buffer in devices that support memory type selection.

Define Trigger Conditions

The SignalTap II Embedded Logic Analyzer captures data continuously while it is running. To capture and store specific signal data, set up triggers that tell the logic analyzer under what conditions to stop capturing data. The SignalTap II Embedded Logic Analyzer lets you define trigger conditions that range from very simple, such as the rising edge of a single signal, to very complex, involving groups of signals, extra logic, and multiple conditions. Power-Up Triggers give you the ability to capture data from trigger events occurring immediately after the device enters user-mode after configuration.

Compile the Design

With the **.stp** file configured and trigger conditions defined, compile your project as usual to include the logic analyzer in your design. Because you may need to change monitored signal nodes or adjust trigger settings frequently during debugging, Altera recommends that you use the incremental compilation feature built into the SignalTap II Embedded Logic Analyzer, along with Quartus II incremental compilation, to reduce recompile times.

Program the Target Device or Devices

When you are debugging a design with the SignalTap II Embedded Logic Analyzer, you can program a target device directly from the **.stp** file without using the Quartus II Programmer. You can also program multiple devices with different designs and simultaneously debug them.

Run the SignalTap II Embedded Logic Analyzer

In normal device operation, you control the logic analyzer through the JTAG connection, specifying when to start looking for trigger conditions to begin capturing data. With Runtime or Power-Up Triggers, read and transfer the captured data from the on-chip buffer to the **.stp** file for analysis.

View, Analyze, and Use Captured Data

After you have captured data and read it into the **.stp** file, it is available for analysis and use in the debugging process. Either manually or with a plug-in, set up mnemonic tables to make it easier to read and interpret the captured signal data. To speed up debugging, use the Locate feature in the **SignalTap II node** list to find the locations of problem nodes in other tools in the Quartus II software. Save the captured data for later analysis, or convert it to other formats for sharing and further study.

Add the SignalTap II Embedded Logic Analyzer to Your Design

Because the SignalTap II Embedded Logic Analyzer is implemented in logic on your target device, it must be added to your FPGA design as another part of the design itself. There are two ways to generate the SignalTap II Embedded Logic Analyzer and add it to your design for debugging:

- Create an **.stp** file and use the SignalTap II Editor to configure the details of the logic analyzer

or

- Create and configure the **.stp** file with the MegaWizard Plug-In Manager and instantiate it in your design

Creating and Enabling a SignalTap II File

To create an embedded logic analyzer, use an existing **.stp** file or create a new file. After a file is created or selected, it must be enabled in the project where it is used.

Creating a SignalTap II File

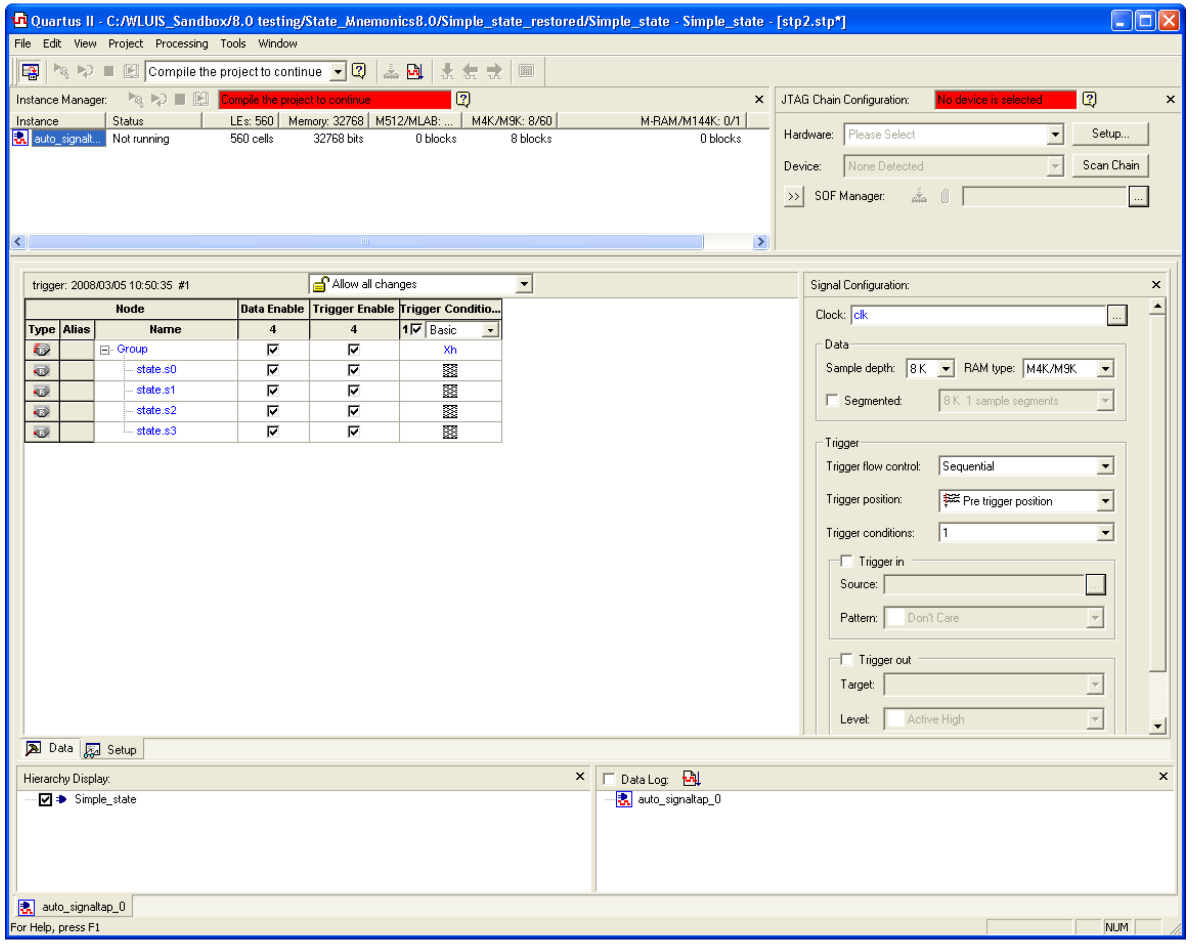
The **.stp** file contains the SignalTap II Embedded Logic Analyzer settings and the captured data for viewing and analysis. To create a new **.stp** file, perform the following steps:

1. On the File menu, click **New**.
2. In the **New** dialog box, click the **Other Files** tab and select **SignalTap II Logic Analyzer File**.
3. Click **OK**.

To open an existing **.stp** file already associated with your project, on the Tools menu, click **SignalTap II Logic Analyzer**. You can also use this method to create a new **.stp** file if no **.stp** file exists for the current project.

To open an existing file, on the File menu, click **Open** and select an **.stp** file (Figure 14–4).

Figure 14–4. SignalTap II Editor



Enabling and Disabling a SignalTap II File for the Current Project

Whenever you save a new **.stp** file, the Quartus II software asks you if you want to enable the file for the current project. However, you can add this file manually, change the selected **.stp** file, or completely disable the logic analyzer by performing the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **SignalTap II Logic Analyzer**. The **SignalTap II Logic Analyzer** page appears.
3. Turn on **Enable SignalTap II Logic Analyzer.** Turn off this option to disable the logic analyzer, completely removing it from your design.
4. In the **SignalTap II File name** box, type the name of the **.stp** file you want to include with your design, or browse to and select a file name.
5. Click **OK**.

Embedding Multiple Analyzers in One FPGA

The SignalTap II Editor includes support for adding multiple logic analyzers using a single **.stp** file. This feature is well-suited for creating a unique logic analyzer for each clock domain in the design.

To create multiple analyzers, on the Edit menu, click **Create Instance**, or right-click in the Instance Manager window and click **Create Instance**.

You can configure each instance of the SignalTap II Embedded Logic Analyzer independently. The icon in the Instance Manager for the currently active instance that is available for configuration is highlighted by a blue box. To configure a different instance, double-click the icon or name of another instance in the Instance Manager.

Monitoring FPGA Resources Used by the SignalTap II Embedded Logic Analyzer

The SignalTap II Embedded Logic Analyzer has a built-in resource estimator that calculates the logic resources and amount of memory that each logic analyzer instance uses. Furthermore, because the most demanding on-chip resource for the embedded logic analyzer is memory usage, the resource estimator reports the ratio of total RAM usage in your design to the total amount of RAM available, given the results of the last compilation. The resource estimator provides a warning if a potential for a “no-fit” occurs.

You can see resource usage of each logic analyzer instance and total resources used in the columns of the Instance Manager section of the SignalTap II Editor. Use this feature when you know that your design is running low on resources.

The logic element value reported in the resource usage estimator may vary by as much as 10% from the actual resource usage.

Table 14–2 shows the SignalTap II Embedded Logic Analyzer M4K memory block resource usage for the listed devices per signal width and sample depth.

Table 14–2. SignalTap II Embedded Logic Analyzer M4K Block Utilization for Stratix® II, Stratix, Stratix GX, and Cyclone® Devices *(Note 1)*

Signals (Width)	Samples (Depth)			
	256	**512**	**2,048**	**8,192**
8	< 1	1	4	16
16	1	2	8	32
32	2	4	16	64
64	4	8	32	128
256	16	32	128	512

Note to Table 14–2:

(1) When you configure a SignalTap II Embedded Logic Analyzer, the Instance Manager reports an estimate of the memory bits and logic elements required to implement the given configuration.

Using the MegaWizard Plug-In Manager to Create Your Embedded Logic Analyzer

You can create a SignalTap II Embedded Logic Analyzer instance by using the MegaWizard Plug-In Manager. The MegaWizard Plug-In Manager generates an HDL file that you instantiate in your design.

The State-based trigger flow, the state machine debugging feature, and the storage qualification feature are not supported when using the MegaWizard Plug-In Manager to create the embedded logic analyzer. These features are described in the following sections:

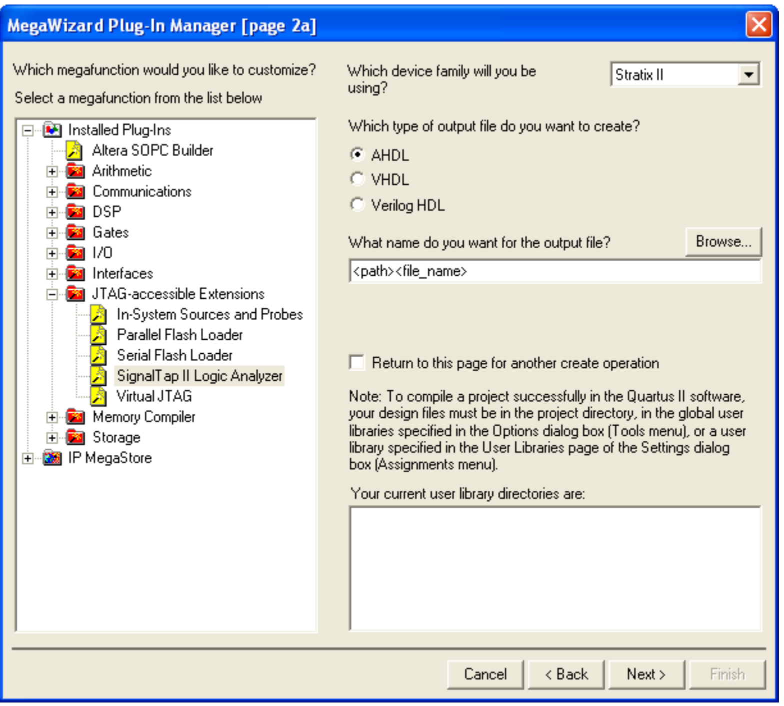
- "Adding Finite State Machine State Encoding Registers" on page 14–20
- "Using the Storage Qualifier Feature" on page 14–25
- "Custom State-Based Triggering" on page 14–38

Creating an HDL Representation Using the MegaWizard Plug-In Manager

The Quartus II software allows you to easily create your SignalTap II Embedded Logic Analyzer using the MegaWizard Plug-In Manager. To implement the SignalTap II megafunction, perform the following steps:

1. On the Tools menu, click **MegaWizard Plug-In Manager**. Page 1 of the **MegaWizard Plug-In Manager** appears.
2. Select **Create a new custom megafunction variation**.
3. Click **Next**.
4. In the **Installed Plug-Ins** list, expand the **JTAG-accessible Extensions** folder and select SignalTap II Embedded Logic Analyzer. Select an output file type and enter the desired name of the SignalTap II megafunction. You can choose AHDL (**.tdf**), VHDL (**.vhd**), or Verilog HDL (**.v**) as the output file type (Figure 14–5).

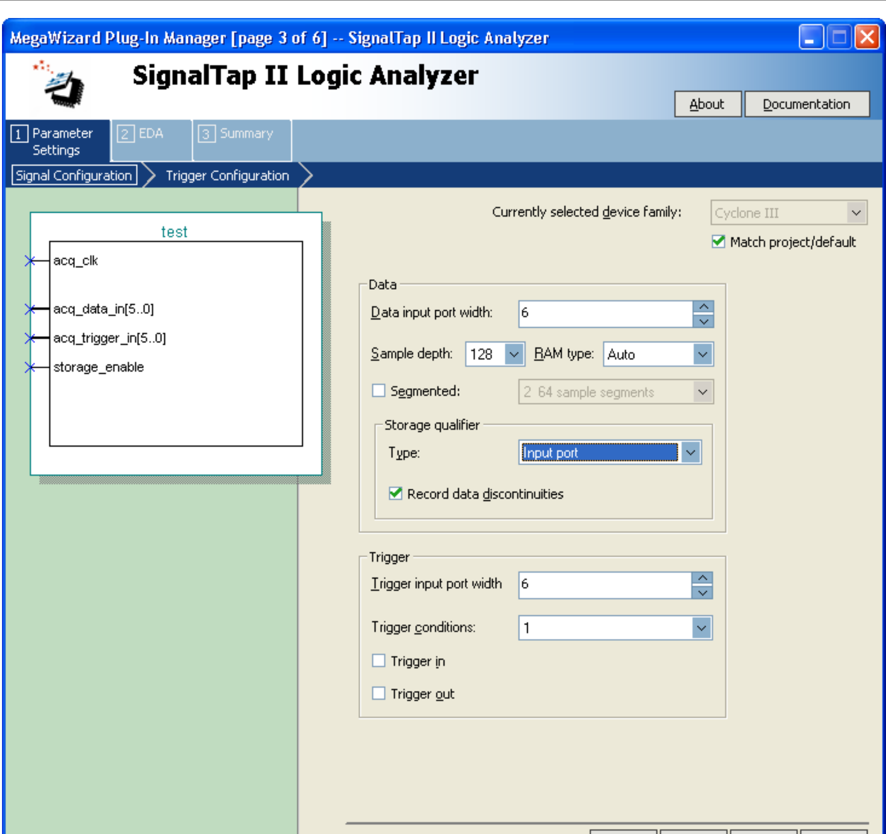
Figure 14–5. Creating the SignalTap II Embedded Logic Analyzer in the MegaWizard Plug-In Manager



5. Click **Next**.
6. Configure the analyzer by specifying the **Sample depth**, **RAM Type**, **Data input port width**, **Trigger levels**, **Trigger input port width**, whether to enable an external **Trigger in** or **Trigger out,** whether to enable the **Segmented** memory buffer option, and whether to enable the Storage Qualifier for non-segmented buffers (Figure 14–6).

   For information about these settings, refer to "Configure the SignalTap II Embedded Logic Analyzer" on page 14–14 and "Define Triggers" on page 14–33.

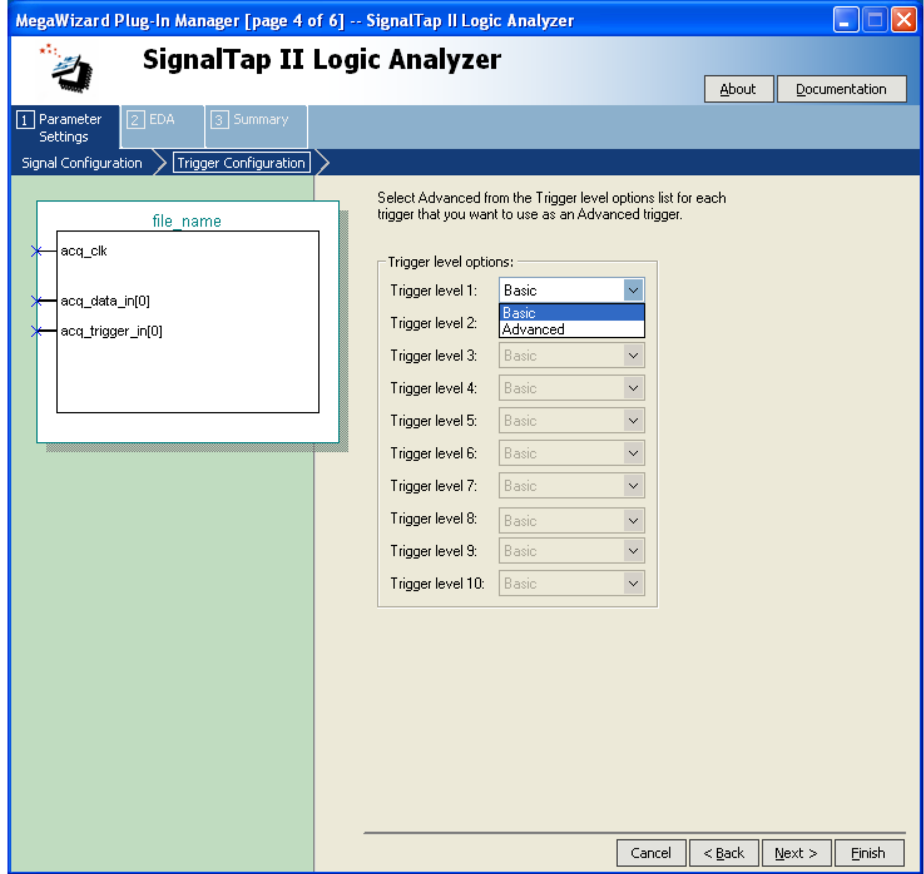
Figure 14–6. Select Embedded Logic Analyzer Parameters



7. Click **Next**.
8. Set the **Trigger level** options by selecting **Basic** or **Advanced** (Figure 14–7). If you select **Advanced** for any trigger level, the next page of the MegaWizard Plug-In Manager displays the Advanced Trigger Condition Editor. You can configure an advanced trigger expression using the number of signals you specified for the trigger input port width.

☞ You cannot define a Power-Up Trigger using the MegaWizard Plug-In Manager. Refer to "Define Triggers" on page 14–33 to learn how to do this using the **.stp** file.

Figure 14–7. MegaWizard Basic and Advanced Trigger Options



9. On the final page of the MegaWizard Plug-In Manager, select any additional files you want to create and click **Finish** to create an HDL representation of the SignalTap II Embedded Logic Analyzer.

For information about the configuration settings options in the MegaWizard Plug-In Manager, refer to "Configure the SignalTap II Embedded Logic Analyzer" on page 14–14. For information about defining triggers, refer to "Define Triggers" on page 14–33.

SignalTap II Megafunction Ports

Table 14–3 provides information about the SignalTap II megafunction ports.

For the most current information about the ports and parameters for this megafunction, refer to the latest version of the Quartus II Help.

Table 14–3. SignalTap II Megafunction Ports (Part 1 of 2)

Port Name	Type	Required	Description
acq_data_in	Input	No	This set of signals represents signals that are monitored in the SignalTap II Embedded Logic Analyzer.
acq_trigger_in	Input	No	This set of signals represents signals that are used to trigger the analyzer.

Table 14–3. SignalTap II Megafunction Ports (Part 2 of 2)

Port Name	Type	Required	Description
acq_clk	Input	Yes	This port represents the sampling clock that the SignalTap II Embedded Logic Analyzer uses to capture data.
trigger_in	Input	No	This signal is used to trigger the SignalTap II Embedded Logic Analyzer.
trigger_out	Output	No	This signal is enabled when the trigger event occurs.
storage_enable	Input	No	This signal is used to enable a write transaction into the acquisition buffer.

Instantiating the SignalTap II Embedded Logic Analyzer in Your HDL

Add the code from the files that are generated by the MegaWizard Plug-In Manager to your design, mapping the signals in your design to the appropriate SignalTap II megafunction ports. You can instantiate up to 127 analyzers in your design, or as many as physically fit in the FPGA. Once you have instantiated the **.stp** file in your HDL file, compile your Quartus II project to fit the logic analyzer in the target FPGA.

To capture and view the data, create an **.stp** file from your SignalTap II HDL output file. To do this, on the File menu, point to **Create/Update** and click **Create SignalTap II File from Design Instance(s)**.

If you make any changes to your design or the SignalTap II instance, recreate or update the **.stp** file using the **Create/Update** command. This ensures that the **.stp** file is always compatible with the SignalTap II instance in your design. If the **.stp** file is not compatible with the SignalTap II instance in your design, you may not be able to control the SignalTap II Embedded Logic Analyzer after it is programmed into your device.

For information about **.stp** file compatibility with programmed SignalTap II instances, refer to "Program the Target Device or Devices" on page 14–59.

Configure the SignalTap II Embedded Logic Analyzer

The **.stp** file provides many options for configuring instances of the logic analyzer. Some of the settings are similar to those found on traditional external logic analyzers. Other settings are unique to the SignalTap II Embedded Logic Analyzer because of the requirements for configuring an embedded logic analyzer. All settings give you the ability to configure the logic analyzer the way you want to help debug your design.

Some settings can only be adjusted when you are viewing Run-Time Trigger conditions instead of Power-Up Trigger conditions. To learn about Power-Up Triggers and viewing different trigger conditions, refer to "Creating a Power-Up Trigger" on page 14–49.

Assigning an Acquisition Clock

Assign a clock signal to control the acquisition of data by the SignalTap II Embedded Logic Analyzer. The logic analyzer samples data on every positive (rising) edge of the acquisition clock. The logic analyzer does not support sampling on the negative (falling) edge of the acquisition clock. You can use any signal in your design as the acquisition clock. However, for best results, Altera recommends that you use a global,

non-gated clock synchronous to the signals under test for data acquisition. Using a gated clock as your acquisition clock can result in unexpected data that does not accurately reflect the behavior of your design. The Quartus II static timing analysis tools show the maximum acquisition clock frequency at which you can run your design. Refer to the Timing Analysis section of the Compilation Report to find the maximum frequency of the logic analyzer clock.

To assign an acquisition clock, perform the following steps:

1. In the SignalTap II Logic Analyzer window, click the **Setup** tab.
2. In the **Signal Configuration** pane, next to the **Clock** field, click **Browse**. The **Node Finder** dialog box appears.
3. From the **Filter** list, select **SignalTap II: post-fitting**
   or
   SignalTap II: pre-synthesis.
4. In the **Named** field, type the exact name of a node that you want to use as your sample clock, or search for a node using a partial name and wildcard characters.
5. To start the node search, click **List**.
6. In the **Nodes Found** list, select the node that represents the design's global clock signal.
7. Add the selected node name to the **Selected Nodes** list by clicking "**>**" or by double-clicking the node name.
8. Click **OK**. The node is now specified as the acquisition clock in the SignalTap II Editor.

If you do not assign an acquisition clock in the SignalTap II Editor, the Quartus II software automatically creates a clock pin called `auto_stp_external_clk`.

You must make a pin assignment to this pin independently from the design. Ensure that a clock signal in your design drives the acquisition clock.

For information about assigning signals to pins, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Adding Signals to the SignalTap II File

While configuring the logic analyzer, add signals to the node list in the **.stp** file to select which signals in your design you want to monitor. Selected signals are also used to define triggers. You can assign the following two types of signals to your **.stp** file:

- **Pre-synthesis**—This signal exists after design elaboration, but before any synthesis optimizations are done. This set of signals should reflect your Register Transfer Level (RTL) signals.
- **Post-fitting**—This signal exists after physical synthesis optimizations and place-and-route.

If you are not using incremental compilation, add only pre-synthesis signals to your **.stp** file. Using pre-synthesis is particularly useful if you want to add a new node after you have made design changes. Source file changes appear in the Node Finder after an Analysis and Elaboration has been performed. On the Processing Menu, point to **Start** and click **Start Analysis & Elaboration**.

The Quartus II software does not limit the number of signals available for monitoring in the SignalTap II window waveform display. However, the number of channels available is directly proportional to the number of logic elements (LEs) or adaptive logic modules (ALMs) in the device. Therefore, there is a physical restriction on the number of channels that are available for monitoring. Signals shown in blue text are post-fit node names. Signals shown in black text are pre-synthesis node names.

After successful Analysis and Elaboration, the signals shown in red text are invalid signals. Unless you are certain that these signals are valid, remove them from the **.stp** file for correct operation. The SignalTap II Status Indicator also indicates if an invalid node name exists in the **.stp** file.

As a general guideline, signals can be tapped if a routing resource (row or column interconnects) exists to route the connection to the SignalTap II instance. For example, signals that exist in the I/O element (IOE) cannot be directly tapped because there are no direct routing resources from the signal in an IOE to a core logic element. For input pins, you can tap the signal that is driving a logic array block (LAB) from an IOE, or, for output pins, you can tap the signal from the LAB that is driving an IOE.

When adding pre-synthesis signals, all connections made to the SignalTap II Embedded Logic Analyzer are made prior to synthesis. Logic and routing resources are allocated during recompilation to make the connection as if a change in your design files had been made. As such, pre-synthesis signal names for signals driving to and from IOEs coincide with the signal names assigned to the pin.

In the case of post-fit signals, connections that you make to the SignalTap II Embedded Logic Analyzer are the signal names from the actual atoms in your post-fit netlist. A connection can only be made if the signals are part of the existing post-fit netlist and existing routing resources are available from the signal of interest to the SignalTap II Embedded Logic Analyzer. In the case of post-fit output signals, tap the `COMBOUT` or `REGOUT` signal that drives the IOE block. For post-fit input signals, signals driving into the core logic coincide with the signal name assigned to the pin.

If you are tapping the signal from the atom that is driving an IOE, be aware that the signal may be inverted due to `NOT`-gate push back. You can check this by locating the signal in either the Resource Property Editor or the Technology Map Viewer. The Technology Map viewer and the Resource Property Editor are also helpful in finding post-fit node names.

For information about cross-probing to source design file and other Quartus II windows, refer to the *Analyzing Designs with Quartus II Netlist Viewers* chapter in volume 1 of the *Quartus II Handbook*.

For more information about the use of incremental compilation with the SignalTap II Embedded Logic Analyzer, refer to "Faster Compilations with Quartus II Incremental Compilation" on page 14–53.

Signal Preservation

Many of the RTL signals are optimized during the process of synthesis and place-and-route. RTL signal names frequently may not appear in the post-fit netlist after optimizations. For example, the compilation process can add tildes ("~") to nets that are fanning out from a node, making it difficult to decipher which signal nets they actually represent. This can cause a problem when you use the incremental compilation flow with the SignalTap II Embedded Logic Analyzer. Because only post-fitting signals can be added to the SignalTap II Embedded Logic Analyzer in partitions of type **post-fit**, RTL signals that you want to monitor may not be available, preventing their usage. To avoid this issue, use synthesis attributes to preserve signals during synthesis and place-and-route. When the Quartus II software encounters these synthesis attributes, it does not perform any optimization on the specified signals, forcing them to continue to exist in the post-fit netlist. However, if you do this, you could see an increase in resource utilization or a decrease in timing performance. The two attributes you can use are:

- `keep`—Ensures that combinational signals are not removed
- `preserve`—Ensures that registers are not removed

For more information about using these attributes, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

If you are debugging an IP core, such as the Nios II CPU or other encrypted IP, you might need to preserve nodes from the core to make them available for debugging with the SignalTap II Embedded Logic Analyzer. This is often necessary when a plug-in is used to add a group of signals for a particular IP.

To prevent the Quartus II software from optimizing away debugging signals on IP cores, perform the following steps:

1. In the Quartus II GUI, on the Assignments menu, click **Settings**.
2. In the **Category** list, select **Analysis & Synthesis Settings**.
3. Turn on **Create debugging nodes** for IP cores to make these nodes available to the SignalTap II Embedded Logic Analyzer.

Assigning Data Signals Using the Node Finder

To assign data signals, perform the following steps:

1. Perform Analysis and Elaboration, Analysis and Synthesis, or fully compile your design.
2. In the SignalTap II Logic Analyzer window, click the **Setup** tab.
3. Double-click anywhere in the node list of the SignalTap II Editor to open the **Node Finder** dialog box.
4. In the **Fitter** list, select **SignalTap II: pre-synthesis** or **SignalTap II: post-fitting**. Only signals listed under one of these filters can be added to the **SignalTap II node** list. Signals cannot be selected from any other filters.

Altera recommends that you do not add a mix of pre-synthesis and post-fitting signals within the same partition. For more details, refer to "Using Incremental Compilation with the SignalTap II Embedded Logic Analyzer" on page 14–55.

If you use incremental compilation flow with the SignalTap II Embedded Logic Analyzer, pre-synthesis nodes may not be connected to the SignalTap II Embedded Logic Analyzer if the affected partition is of the post-fit type. A critical warning is issued for all pre-synthesis node names that are not found in the post-fit netlist.

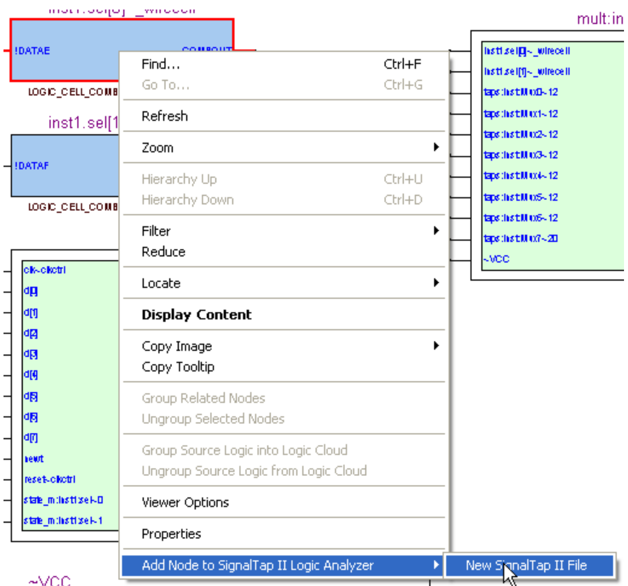
1. In the **Named** field, type a node name, or search for a particular node by entering a partial node name along with wildcard characters. To start the node name search, click **List**.
2. In the **Nodes Found** list, select the node or bus you want to add to the **.stp** file.
3. Add the selected node name(s) to the **Selected Nodes** list by clicking “>” or by double-clicking the node name(s).
4. To insert the selected nodes in the **.stp** file, click **OK**. With the default colors set for the SignalTap II Embedded Logic Analyzer, a pre-synthesis signal in the list is shown in black; a post-fitting signal is shown in blue.

You can also drag and drop signals from the **Node Finder** dialog box into an **.stp** file.

Assigning Data Signals Using the Technology Map Viewer

Starting with Quartus II software version 8.0, you can easily add post-fit signal names that you find in the Technology map viewer. To do so, launch the Technology map viewer (post-fitting) after compiling your design. When you find the desired node, copy the node to either the active **.stp** file for your design or a new **.stp** file. Figure 14–8 shows the right-click menu for adding a node using the Technology map viewer.

Figure 14–8. Finding Data Signals Using the Technology Map Viewer



Node List Signal Use Options

When a signal is added to the node list, you can select options that specify how the signal is used with the logic analyzer. You can turn off the ability of a signal to trigger the analyzer by disabling the Trigger Enable option for that signal in the node list in the **.stp** file. This option is useful when you want to see only the captured data for a signal and you are not using that signal as part of a trigger.

You can turn off the ability to view data for a signal by disabling the **Data Enable** column. This option is useful when you want to trigger on a signal, but have no interest in viewing data for that signal.

For information about using signals in the node list to create SignalTap II trigger conditions, refer to "Define Triggers" on page 14–33.

Untappable Signals

Not all of the post-fitting signals in your design are available in the SignalTap II: post-fitting filter in the **Node Finder** dialog box. The following signal types cannot be tapped:

- **Post-fit output pins**—You cannot tap a post-fit output pin directly. To make an output signal visible, tap the register or buffer that drives the output pin. This includes pins defined as bidirectional.
- **Signals that are part of a carry chain**—You cannot tap the carry out (`cout0` or `cout1`) signal of a logic element. Due to architectural restrictions, the carry out signal can only feed the carry in of another LE.
- **JTAG Signals**—You cannot tap the JTAG control (`TCK`, `TDI`, `TDO`, and `TMS`) signals.
- **ALTGXB megafunction**—You cannot directly tap any ports of an ALTGXB instantiation.
- **LVDS**—You cannot tap the data output from a serializer/deserializer (SERDES) block.
- **DQ**, **DQS Signals**—You cannot directly tap the `DQ` or `DQS` signals in a DDR/DDRII design.

Adding Signals with a Plug-In

Instead of adding individual or grouped signals through the **Node Finder**, you can add groups of relevant signals of a particular type of IP through the use of a plug-in. The SignalTap II Embedded Logic Analyzer comes with one plug-in already installed for the Nios II processor. Besides easy signal addition, plug-ins also provide a number of other features, such as pre-designed mnemonic tables, useful for trigger creation and data viewing, as well as the ability to disassemble code in captured data.

The Nios II plug-in, for example, creates one mnemonic table in the **Setup** tab and two tables in the **Data** tab:

- **Nios II Instruction** (**Setup** tab)—Capture all the required signals for triggering on a selected instruction address.
- **Nios II Instance Address** (**Data** tab)—Display address of executed instructions in hexadecimal format or as a programming symbol name if defined in an optional Executable and Linking Format (**.elf**) file.

- **Nios II Disassembly** (**Data** tab)—Displays disassembled code from the corresponding address.

For information about the other features plug-ins provided, refer to "Define Triggers" on page 14–33 and "View, Analyze, and Use Captured Data" on page 14–66.

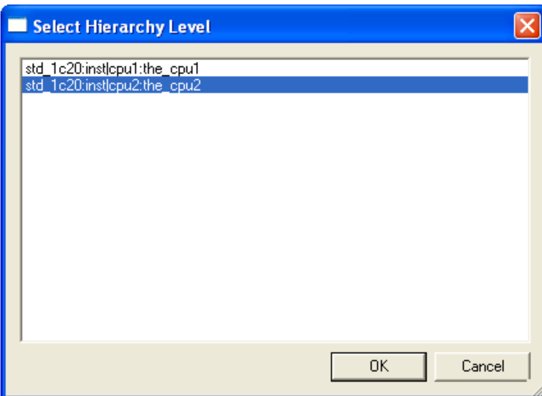
To add signals to the **.stp** file using a plug-in, perform the following steps after running Analysis and Elaboration on your design:

1. Right-click in the node list. On the Add Nodes with Plug-In submenu, click the name of the plug-in you want to use, such as the included plug-in named **Nios II**.

   If the IP for the selected plug-in does not exist in your design, a message appears informing you that you cannot use the selected plug-in.

2. The **Select Hierarchy Level** dialog box appears showing the IP hierarchy of your design (Figure 14–9). Select the IP that contains the signals you want to monitor with the plug-in and click **OK**.

Figure 14–9. IP Hierarchy Selection



3. If all the signals in the plug-in are available, a dialog box might appear, depending on the plug-in selected, where you can set any available options for the plug-in. With the Nios II plug-in, you can optionally select an **.elf** file containing program symbols from your Nios II Integrated Development Environment (IDE) software design. Set options for the selected plug-in as desired and click **OK**.

To make sure all the required signals are available, in the Quartus II **Analysis & Synthesis** settings, turn on the **Create debugging nodes for IP cores** option.

All the signals included in the plug-in are added to the node list.

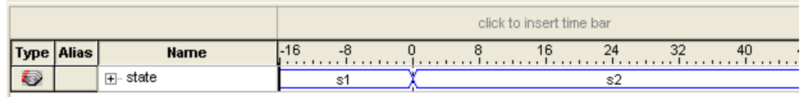
Adding Finite State Machine State Encoding Registers

Finding the signals to debug Finite State Machines (FSM) can be challenging. Finding nodes from the post-fit netlist may be impossible, as FSM encoding signals may be changed or optimized away during synthesis and place-and-route. If you are able to find all of the relevant nodes in the post-fit netlist or you used the nodes from the pre-synthesis netlist, an additional step is required to find and map FSM signal values to the state names that you specified in your HDL.

Beginning with Quartus II software version 8.0, the SignalTap II GUI can detect FSMs in your compiled design. The SignalTap II configuration automatically tracks the FSM state signals as well as state encoding through the compilation process. Right-click dialog boxes from the SignalTap II GUI allow you to add all of the FSM state signals to your embedded logic analyzer with a single command. For each FSM added to your SignalTap II configuration, the FSM debugging feature adds a mnemonic table to map the signal values to the state enumeration that you provided in your source code. The mnemonic tables enable you to visualize state machine transitions in the waveform viewer easily. The FSM debugging feature supports adding FSM signals from both the pre-synthesis and post-fit netlists.

Figure 14–10 shows the waveform viewer with decoded signal values from a state machine added with the FSM debugging feature.

Figure 14–10. Decoded FSM Mnemonics



For coding guidelines for specifying FSM in Verilog and VHDL, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

To add pre-synthesis FSM signals to the configuration file, perform the following steps after running Analysis and Elaboration on your design:

1. Create a new **.stp** file or use an existing **.stp** file.

   Any **.stp** files that the MegaWizard Plug-In Manager creates from instantiations are not supported for this feature.

2. In the SignalTap II setup tab, right-click anywhere on the node list and select **Add State Machine Nodes**. The **Add State Machine Nodes** dialog box appears. This dialog box lists all the FSMs that have been found in your design.

   For the SignalTap II GUI to detect pre-synthesis state-machine signals, perform Analysis and Elaboration of your design.

3. From the Netlist pull-down menu, select **Pre-Synthesis**.
4. Select the desired FSM.
5. Click **OK**. This adds the FSM nodes to the configuration file. A mnemonic table is automatically applied to the FSM signal group.

To add post-fit FSM signals to the configuration file, perform the following steps after performing a full compile of your design:

1. Set the design partition of the FSM that you want to debug to post-fit.
2. Enable the **.stp** file for the Quartus II project using the **SignalTap II Embedded Logic Analyzer** page of the **Settings** dialog box. You can either create a new **.stp** file or use an existing **.stp** file.

For the SignalTap II GUI to detect post-fit state-machine signals, perform a full compile of your design.

3. In the SignalTap II setup tab, right-click anywhere on the node list and select **Add State Machine Nodes**. The **Add State Machine Nodes** dialog box appears. This dialog box lists all the FSMs that have been found in your design.
4. From the Netlist pull-down menu, select **Post-Fit**.
5. Select the desired FSM.
6. Click **OK**. This adds the FSM nodes to the configuration file. A mnemonic table is automatically applied to the FSM signal group.

Modifying and Restoring Mnemonic Tables for State Machines

When you add FSM state signals via the FSM debugging feature, the SignalTap II GUI creates a mnemonic table using the format <*StateSignalName*>`_table`, where **StateSignalName** is the name of the state signals that you have declared in your RTL. You can edit any mnemonic table using the **Mnemonic Table Setup** dialog box.

If you want to restore a mnemonic table that was modified, right-click anywhere in the node list window and select **Recreate State Machine Mnemonics**. By default, restoring a mnemonic table overwrites the existing mnemonic table that you modified. If you would like to restore a FSM mnemonic table to a new record, uncheck the **Overwrite existing mnemonic table** option in the **Recreate State Machine Mnemonics** dialog box.

If you have added or deleted a signal from the FSM state signal group from within the setup tab, delete the modified register group and add the FSM signals back again.

For more information about using Mnemonics, refer to "Creating Mnemonics for Bit Patterns" on page 14–69.

Additional Considerations

The SignalTap II configuration GUI recognizes state machines from your design only if you use Quartus II Integrated Synthesis (QIS). The state machine debugging feature is not able to track the FSM signals or state encoding if you have used a third-party synthesis tool.

If you are adding post-fit FSM signals, the SignalTap II FSM debug feature may not be able to track all of the optimization changes that are a part of the compilation process. If the following two specific optimizations are enabled, the SignalTap II FSM debug feature may not list mnemonic tables for state machines in the design:

- If you have physical synthesis turned on, state registers may be resource balanced (register retiming) to improve f_{MAX}. The FSM debug feature does not list post-fit FSM state registers if register retiming occurs.
- The FSM debugging feature does not list state signals that have been packed into RAM and DSP blocks during QIS or Fitter optimizations.

You are still able to use the FSM debugging feature to add pre-synthesis state signals.

Specifying the Sample Depth

The sample depth specifies the number of samples that are captured and stored for each signal in the captured data buffer. To set the sample depth, select the desired number of samples to store in the **Sample Depth** list. The sample depth ranges from 0 to 128K.

If device memory resources are limited, you may not be able to successfully compile your design with the sample buffer size you have selected. Try reducing the sample depth to reduce resource usage.

Capturing Data to a Specific RAM Type

When you use the SignalTap II Embedded Logic Analyzer with some devices, you have the option to select the RAM type where acquisition data is stored. RAM selection allows you to preserve a specific memory block for your design and allocate another portion of memory for SignalTap II data acquisition. For example, if your design implements a large buffering application such as a system cache, it is ideal to place this application into M-RAM blocks so that the remaining M512 or M4K blocks are used for SignalTap II data acquisition.

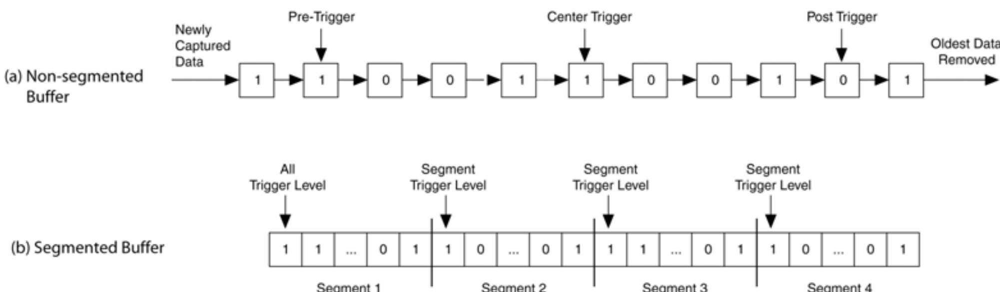
To select the RAM type to use for the SignalTap II buffer, select it from the RAM type list. Use this feature when the acquired data (as reported by the SignalTap II resource estimator) is not larger than the available memory of the memory type that you have selected in the FPGA.

Choosing the Buffer Acquisition Mode

The Buffer Acquisition Type Selection feature in the SignalTap II Embedded Logic Analyzer lets you choose how the captured data buffer is organized and can potentially reduce the amount of memory that is required for SignalTap II data acquisition. There are two types of acquisition buffer within the SignalTap II Embedded Logic Analyzer—a non-segmented buffer and a segmented buffer. With a non-segmented buffer, the SignalTap II Embedded Logic Analyzer treats entire memory space as a single FIFO, continuously filling the buffer until the embedded logic analyzer reaches a defined set of trigger conditions. With a segmented buffer, the memory space is split into a number of separate buffers. Each buffer acts as a separate FIFO with its own set of trigger conditions. Only a single buffer is active during an acquisition. The SignalTap II Embedded Logic Analyzer advances to the next segment after the trigger condition or conditions for the active segment has been reached.

When using a non-segmented buffer, you can use the storage qualification feature to determine which samples are written into the acquisition buffer. Both the segmented buffers and the non-segmented buffer with the storage qualification feature help you maximize the use of the available memory space. Figure 14–11 illustrates the differences between the two buffer types.

Figure 14–11. Buffer Type Comparison in the SignalTap II Embedded Logic Analyzer *(Note 1)*



Note to Figure 14–11:

(1) Both non-segmented and segmented buffers can use a predefined trigger (Pre-Trigger, Center Trigger, Post-Trigger) position or define a custom trigger position using the **State-Based Triggering** tab. Refer to "Specifying the Trigger Position" on page 14–48 for more details.

(2) Each segment is treated like a FIFO, and behaves as the non-segmented buffer shown in (a).

For more information about the storage qualification feature, refer to "Using the Storage Qualifier Feature" on page 14–25.

Non-Segmented Buffer

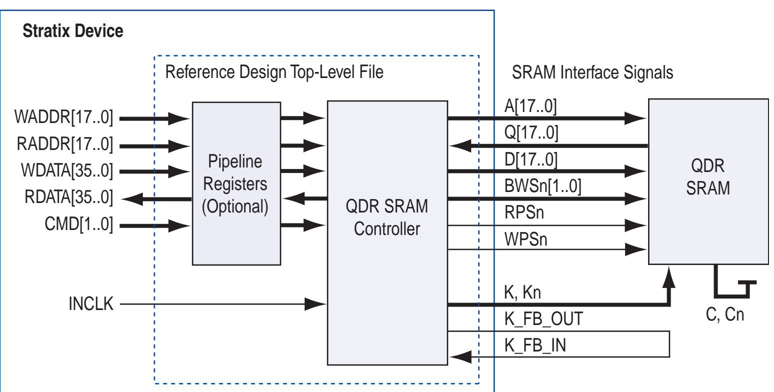
The non-segmented buffer (also known as a circular buffer) shown in Figure 14–11 (a) is the default buffer type used by the SignalTap II Embedded Logic Analyzer. While the logic analyzer is running, data is stored in the buffer until it fills up, at which point new data replaces the oldest data. This continues until a specified trigger event—that is, a set of trigger conditions—occurs. When this happens, the logic analyzer continues to capture data after the trigger event until the buffer is full, based on the trigger position setting in the **Signal Configuration** pane in the **.stp** file. Select a setting from the list to choose whether to capture the majority of the data before (**Post trigger position**), after (**Pre-trigger position**) the trigger occurs, or to center the trigger position in the data (**Center trigger position**). Alternatively, use the custom State-based triggering flow to define a custom trigger position within the capture buffer.

For more information, refer to "Specifying the Trigger Position" on page 14–48.

Segmented Buffer

A segmented buffer makes it easier to debug systems that contain relatively infrequent recurring events. The acquisition memory is split into a set of evenly sized segments, with a set of trigger conditions defined for each segment. Each segment acts as a non-segmented buffer. Figure 14–12 shows an example of this type of buffer system.

Figure 14–12. Example System that Generates Recurring Events



The SignalTap II Embedded Logic Analyzer verifies the functionality of the design shown in Figure 14–12 to ensure that the correct data is written to the SRAM controller. Buffer acquisition in the SignalTap II Embedded Logic Analyzer allows you to monitor the `RDATA` port when `H'0F0F0F0F` is sent into the `RADDR` port. You can monitor multiple read transactions from the SRAM device without running the SignalTap II Embedded Logic Analyzer again. The buffer acquisition feature allows you to segment the memory so you can capture the same event multiple times without wasting allocated memory. The number of cycles that are captured depends on the number of segments specified under the **Data** settings.

To enable and configure buffer acquisition, select Segmented in the SignalTap II Editor and select the number of segments to use. In the example, selecting sixty-four 64-sample segments allows you to capture 64 read cycles when the `RADDR` signal is `H'0F0F0F0F`.

For more information about buffer acquisition mode, refer to *Setting the Buffer Acquisition Mode* in the Quartus II Help.

Using the Storage Qualifier Feature

Both non-segmented and segmented buffers described in the previous section offer a snapshot in time of the data stream being analyzed. The default behavior for writing into acquisition memory with the SignalTap II Embedded Logic Analyzer is to sample data on every clock cycle. With a non-segmented buffer, there is one data window that represents a contiguous snapshot of the datastream. Similarly, segmented buffers use several smaller sampling windows spread out over a larger time scale, with each sampling window representing a contiguous data set.

With carefully chosen trigger conditions and a generous sample depth for the acquisition buffer, analysis using segmented and non-segmented buffers captures a majority of functional errors in a chosen signal set. However, each data window can have a considerable amount of redundancy associated with it; for example, a capture of a data stream containing long periods of idle signals between data bursts. With default behavior using the SignalTap II Embedded Logic Analyzer, there is no way to discard the redundant sample bits.

The Storage Qualification feature allows you to filter out individual samples not relevant to debugging the design. With this feature, a condition acts as a write enable to the buffer each clock cycle during a data acquisition. Through fine tuning the data that is actually stored in acquisition memory, the Storage Qualification feature allows for a more efficient use of acquisition memory and covers a larger time scale.

Use of the Storage Qualification feature is similar to an acquisition using a segmented buffer, in that you can create a discontinuity in the capture buffer. Because you can create a discontinuity between any two samples in the buffer, the Storage Qualification feature is equivalent to being able to create a customized segmented buffer in which the number and size of segment boundaries are adjustable. Figure 14–13 illustrates three ways the SignalTap II Embedded Logic Analyzer writes into acquisition memory.

The Storage Qualification feature can only be used with a non-segmented buffer. The MegaWizard Plug-In Manager instantiated flow only supports the Input Port mode for the Storage Qualification feature.

Figure 14–13. Data Acquisition Using Different Modes of Controlling the Acquisition Buffer

Non-segmented Buffer (1)

Segmented Buffer (2)

Non-segmented Buffer with Storage Qualifier (3)

Notes to Figure 14–13:

(1) Non-segmented Buffers capture a fixed sample window of contiguous data.

(2) Segmented buffers divide the buffer into fixed sized segments, with each segment having an equal sample depth.

(3) Storage Qualification allows you to define a custom sampling window for each segment you create with a qualifying condition. Storage qualification potentially allows for a larger time scale of coverage.

There are five types available under the Storage Qualification feature:

- Continuous
- Input port
- Transitional
- Conditional
- Start/Stop
- State-based

Continuous (the default mode selected) turns the Storage Qualification feature off.

Each selected storage qualifier type is active when an acquisition starts. Upon the start of an acquisition, the SignalTap II Embedded Logic Analyzer examines each clock cycle and writes the data into the acquisition buffer based upon storage qualifier type and condition. The acquisition stops when a defined set of trigger conditions occur.

Trigger conditions are evaluated independently of storage qualifier conditions. The SignalTap II Embedded Logic Analyzer evaluates the data stream for trigger conditions on every clock cycle after the acquisition begins.

Trigger conditions are defined in "Define Trigger Conditions" on page 14–6.

The storage qualifier operates independently of the trigger conditions.

The following subsections describe each storage qualification mode from the acquisition buffer.

Input Port Mode

When using the Input port mode, the SignalTap II Embedded Logic Analyzer takes any signal from your design as an input. When the design is running, if the signal is high on the clock edge, the SignalTap II Embedded Logic Analyzer stores the data in the buffer. If the signal is low on the clock edge, the data sample is ignored. A pin is created and connected to this input port by default if no internal node is specified.

If you are using an **.stp** file to create a SignalTap II Embedded Logic Analyzer instance, specify the storage qualifier signal using the input port field located on the **Setup** tab. This port must be specified for your project to compile.

If you are using the MegaWizard Plug-In Manager flow, the storage qualification input port, if specified, will appear in the MegaWizard-generated instantiation template. This port can then be connected to a signal in your RTL.

Figure 14–14 shows a data pattern captured with a segmented buffer. Figure 14–15 shows a capture of the same data pattern with the storage qualification feature enabled.

Figure 14–14. Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Input port mode)

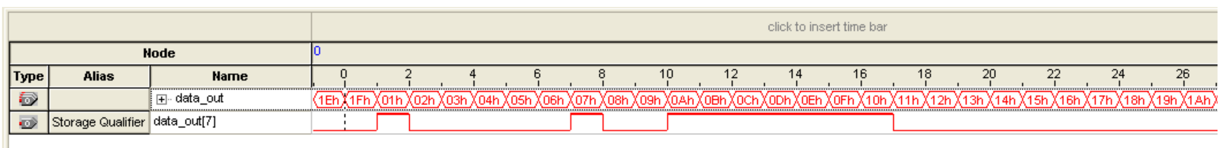


Figure 14–15. Data Acquisition of a Recurring Data Pattern Using an Input Signal as a Storage Qualifier

(1) Markers display samples when the logic analyzer paused a write into acquisition memory. These markers are enabled with the option "Record data discontinuities."

Transitional Mode

In Transitional mode, you choose a set of signals for inspection using the node list check boxes in the storage qualifier column. During acquisition, if any of the signals marked for inspection have changed since the previous clock cycle, new data is written to the acquisition buffer. If none of the signals marked have changed since the previous clock cycle, no data is stored. Figure 14–16 shows the transitional storage qualifier setup. Figure 14–17 and Figure 14–18 show captures of a data pattern in continuous capture mode and a data pattern using the Transitional mode for storage qualification.

Figure 14–16. Transitional Storage Qualifier Setup

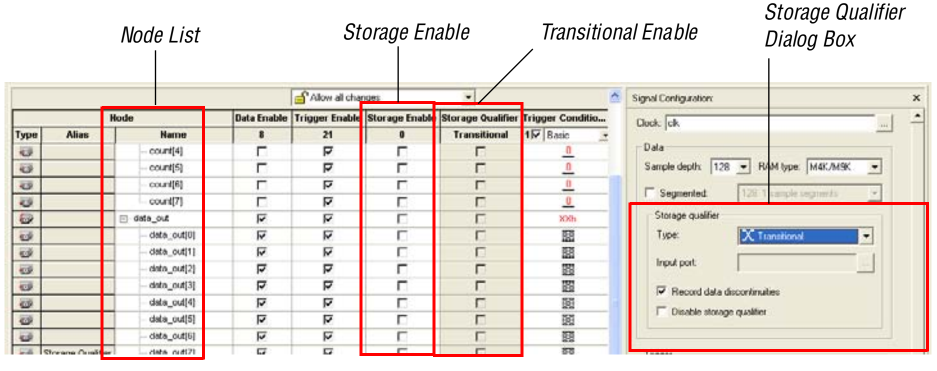


Figure 14–17. Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Transitional mode)

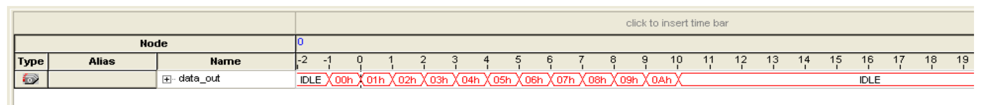
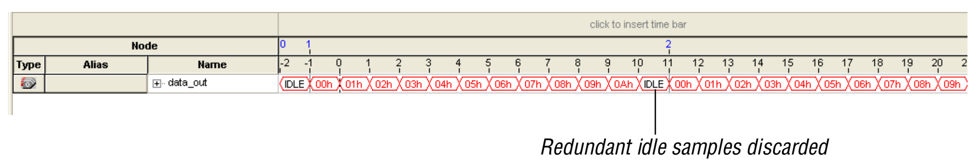


Figure 14–18. Data Acquisition of Recurring Data Pattern Using a Transitional Mode as a Storage Qualifier



Conditional Mode

In Conditional mode, the SignalTap II Embedded Logic Analyzer evaluates a combinational function of storage qualifier enabled signals within the node list to determine whether a sample is stored. The SignalTap II Embedded Logic Analyzer writes into the buffer during the clock cycles in which the condition you specify evaluates `TRUE`.

There are two types of conditions that you can specify: basic and advanced. A basic storage condition matches each signal to one of the following:

- Don't Care
- Low
- High
- Falling Edge
- Either Edge

If a Basic Storage condition is specified for more than one signal, the SignalTap II Embedded Logic Analyzer evaluates the logical AND of the conditions.

Any other combinational or relational operators that you may want to specify with the enabled signal set for storage qualification can be done with an advanced storage condition. Figure 14–19 details the conditional storage qualifier setup in the **.stp** file.

You can set up storage qualification conditions similar to the manner in which trigger conditions are set up. For details about basic and advanced trigger conditions, refer to the sections "Creating Basic Trigger Conditions" on page 14–33 and "Creating Advanced Trigger Conditions" on page 14–34. Figure 14–20 and Figure 14–21 show a data capture with continuous sampling, and the same data pattern using the conditional mode for analysis, respectively.

Figure 14–19. Conditional Storage Qualifier Setup

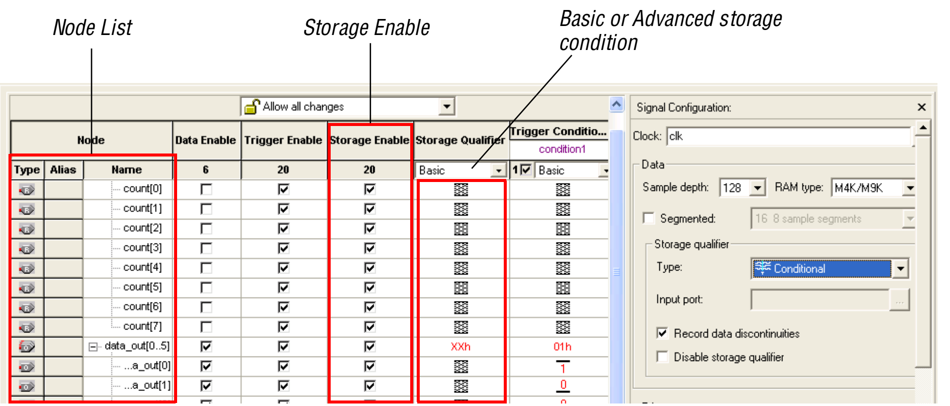
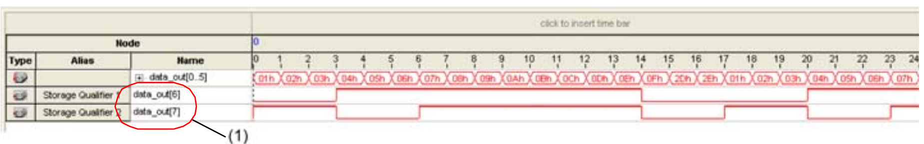
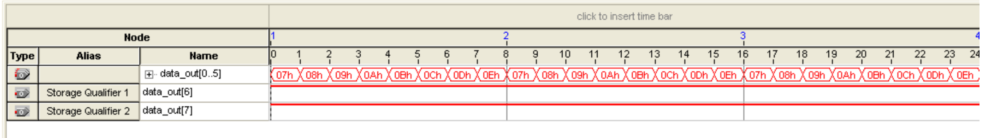


Figure 14–20. Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Conditional capture)



(1) Storage Qualifier condition is set up to pause acquisition when the following occurs:
data_out[6] AND data_out[7] = True. Resultant capture with storage qualifier enabled is shown in Figure 14-21.

Figure 14–21. Data Acquisition of a Recurring Data Pattern in Conditional Capture Mode



Start/Stop Mode

The Start/Stop mode is similar to the Conditional mode for storage qualification. However, in this mode there are two sets of conditions, one for start and one for stop. If the start condition evaluates to `TRUE`, data begins to be stored in the buffer every clock cycle until the stop condition evaluates to `TRUE`, which then pauses the data capture. Additional start signals received after the data capture has started are ignored. If both start and stop evaluate to `TRUE` at the same time, a single cycle is captured.

You can force trigger to the buffer by pressing the S**top** button if the buffer fails to fill to completion due to a stop condition.

Figure 14–22 shows the Start/Stop mode storage qualifier setup. Figure 14–23 and Figure 14–24 show captures data pattern in continuous capture mode and a data pattern in using the Start/Stop mode for storage qualification.

Figure 14–22. Start/Stop Mode Storage Qualifier Setup

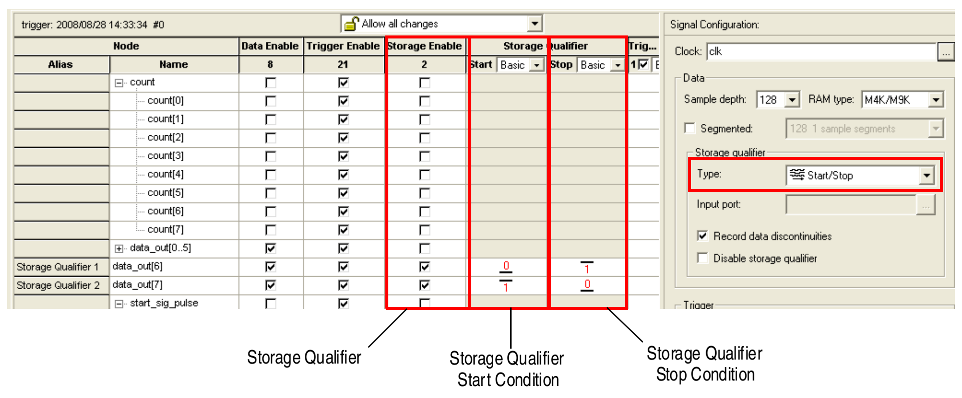


Figure 14–23. Data Acquisition of a Recurring Data Pattern in Continuous Mode (to illustrate Start/Stop mode)

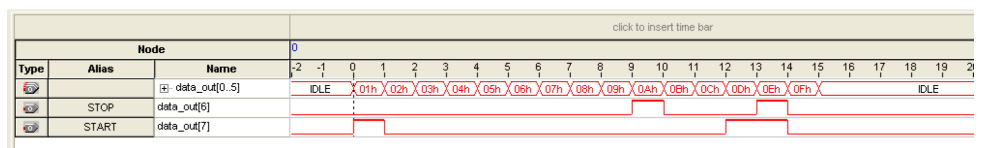
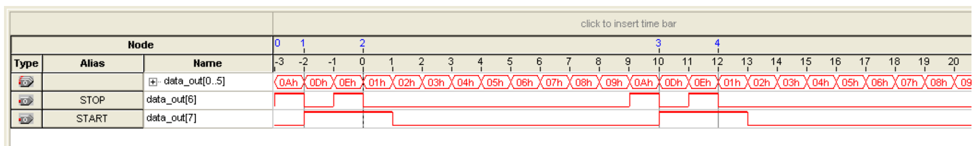


Figure 14–24. Data Acquisition of a Recurring Data Pattern with Start/Stop Storage Qualifier Enabled



State-Based

The State-based storage qualification mode is used with the State-based triggering flow. The state based triggering flow evaluates an if-else based language to define how data is written into the buffer. With the State-based trigger flow, you have command over boolean and relational operators to guide the execution flow for the target acquisition buffer. When the storage qualifier feature is enabled for the State-based flow, two additional commands are available, the `start_store` and `stop_store` commands. These commands operate similarly to the Start/Stop capture conditions described in the previous section. Upon the start of acquisition, data is not written into the buffer until a `start_store` action is performed. The `stop_store` command pauses the acquisition. If both `start_store` and `stop_store` actions are performed within the same clock cycle, a single sample is stored into the acquisition buffer.

For more information about the State-based flow and storage qualification using the State-based trigger flow, refer to the section "Custom State-Based Triggering" on page 14–38.

Showing Data Discontinuities

When you enable the check box option **Record data discontinuities**, the SignalTap II Embedded Logic Analyzer marks the samples during which the acquisition paused from a storage qualifier. This marker is displayed in the waveform viewer after acquisition completes.

Disable Storage Qualifier

The **Disable Storage Qualifier** check box allows you to turn off the storage qualifier quickly and perform a continuous capture. This option is run-time reconfigurable; that is, the setting can be changed without recompiling the project. Changing storage qualifier mode from the Type field requires a recompilation of the project.

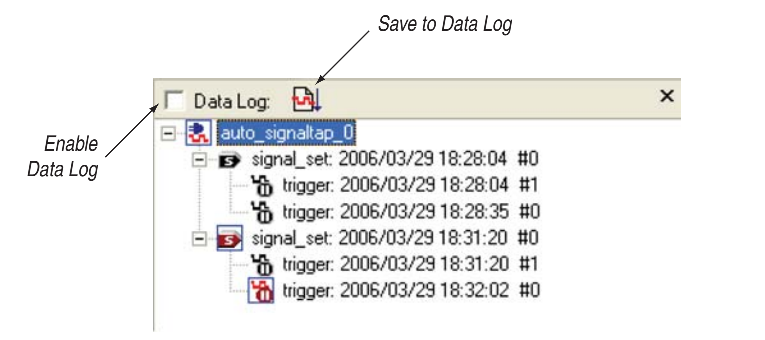
For a detailed explanation of Runtime Reconfigurable options available with the SignalTap II Embedded Logic Analyzer, and storage qualifier application examples using runtime reconfigurable options, refer to "Runtime Reconfigurable Options" on page 14–63.

Managing Multiple SignalTap II Files and Configurations

In some cases you may have more than one **.stp** file in one design. Each file potentially has a different group of monitored signals. These signal groups make it possible to debug different blocks in your design. In turn, each group of signals can also be used to define different sets of trigger conditions. Along with each **.stp** file, there is also an associated programming file (SRAM Object File [**.sof**]). The settings in a selected SignalTap II file must match the SignalTap II logic design in the associated **.sof** file for the logic analyzer to run properly when the device is programmed. Managing all of the **.stp** files and their associated settings and programming files is a challenging task. To help you manage everything, use the Data Log feature and the SOF Manager.

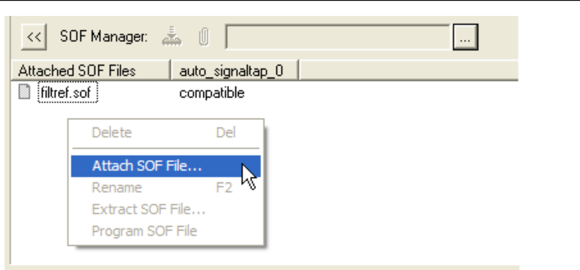
The Data Log allows you to store multiple SignalTap II configurations within a single **.stp** file. Figure 14–25 shows two signal set configurations with multiple trigger conditions in one **.stp** file. To toggle between the active configurations, double-click on an entry in the Data Log. As you toggle between the different configurations, the signal list and trigger conditions change in the **Setup** tab of the **.stp** file. The active configuration displayed in the **.stp** file is indicated by the blue square around the signal set in the Data Log. To store a configuration in the Data Log, on the Edit menu, click **Save to Data Log** or click the **Save to Data Log** button at the top of the Data Log.

Figure 14–25. Data Log



The SOF Manager allows you to embed multiple SOFs into one **.stp** file. Embedding an SOF in an **.stp** file lets you move the **.stp** file to a different location, either on the same computer or across a network, without the need to include the associated **.sof** as a separate file. To embed a new SOF in the **.stp** file, right-click in the SOF Manager, and click **Attach SOF File** (Figure 14–26).

Figure 14–26. SOF Manager



As you switch between configurations in the Data Log, you can extract the SOF that is compatible with that particular configuration and use the programmer in the SignalTap II Embedded Logic Analyzer to download the new SOF to the FPGA. In this way, you ensure that the configuration of your **.stp** file always matches the design programmed into the target device.

Define Triggers

When you start the SignalTap II Embedded Logic Analyzer, it samples activity continuously from the monitored signals. The SignalTap II Embedded Logic Analyzer "triggers"—that is, stops and displays the data—when a condition or set of conditions that you specified has been reached. This section describes the various types of trigger conditions that you can set using the SignalTap II Embedded Logic Analyzer.

Creating Basic Trigger Conditions

The simplest kind of trigger condition is a basic trigger. Select this from the list at the top of the **Trigger Conditions** column in the node list in the SignalTap II Editor. With the trigger type set to Basic, set the trigger pattern for each signal you have added in the **.stp** file. To set the trigger pattern, right-click in the **Trigger Conditions** column and click the desired pattern. Set the trigger pattern to any of the following conditions:

- Don't Care
- Low
- High
- Falling Edge
- Rising Edge
- Either Edge

For buses, type a pattern in binary, or right-click and select **Insert Value** to enter the pattern in other number formats. Note that you can enter `X` to specify a set of "don't care" values in either your hexadecimal or your binary string. For signals added to the **.stp** file that have an associated mnemonic table, you can right-click and select an entry from the table to set pre-defined conditions for the trigger.

For more information about creating and using mnemonic tables, refer to "View, Analyze, and Use Captured Data" on page 14–66, and to the Quartus II Help.

For signals added with certain plug-ins, you can create basic triggers easily using predefined mnemonic table entries. For example, with the Nios II plug-in, if you have specified an **.elf** file from your Nios II IDE design, you can type the name of a function from your Nios II code. The logic analyzer triggers when the Nios II instruction address matches the address of the specified code function name.

Data capture stops and the data is stored in the buffer when the logical `AND` of all the signals for a given trigger condition evaluates to `TRUE`.

Creating Advanced Trigger Conditions

With the SignalTap II Embedded Logic Analyzer's basic triggering capabilities, you can build more complex triggers utilizing extra logic that enables you to capture data when a particular combination of conditions exist. If you set the trigger type to **Advanced** at the top of the **Trigger Conditions** column in the node list of the SignalTap II Editor, a new tab named **Advanced Trigger** appears where you can build a complex trigger expression using a simple GUI. To build the complex trigger condition in an expression tree, drag-and-drop operators into the Advanced Trigger Configuration Editor window. To configure the operators' settings, double-click or right-click the operators that you have placed and select **Properties**. Table 14–4 lists the operators you can use.

Table 14–4. Advanced Triggering Operators *(Note 1)*

Name of Operator	Type
Less Than	Comparison
Less Than or Equal To	Comparison
Equality	Comparison
Inequality	Comparison
Greater Than	Comparison
Greater Than or Equal To	Comparison
Logical NOT	Logical
Logical AND	Logical
Logical OR	Logical
Logical XOR	Logical
Reduction AND	Reduction
Reduction OR	Reduction
Reduction XOR	Reduction
Left Shift	Shift
Right Shift	Shift
Bitwise Complement	Bitwise
Bitwise AND	Bitwise
Bitwise OR	Bitwise
Bitwise XOR	Bitwise
Edge and Level Detector	Signal Detection

Note to Table 14–4:

(1) For more information about each of these operators, refer to the Quartus II Help.

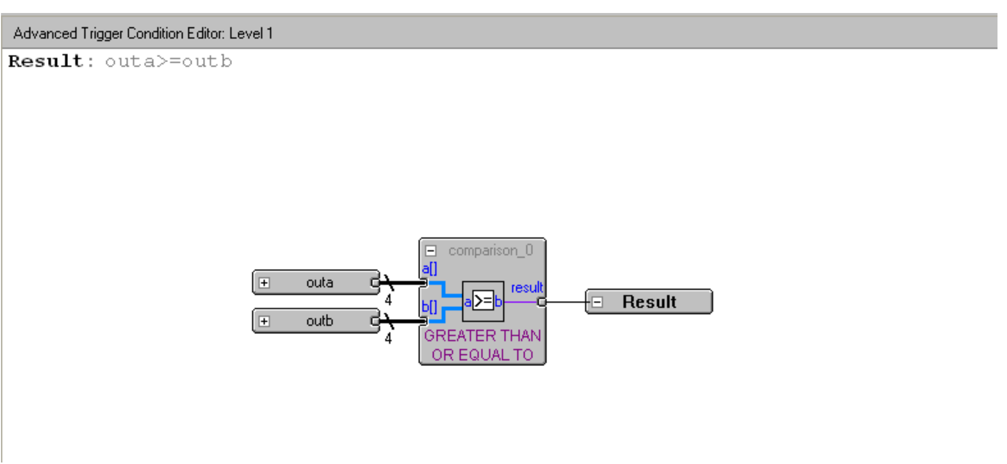
Adding many objects to the Advanced Trigger Condition Editor can make the work space cluttered and difficult to read. To keep objects organized while you build your advanced trigger condition, use the right-click menu and select **Arrange All Objects**. You can also use the **Zoom-Out** command to fit more objects into the Advanced Trigger Condition Editor window.

Examples of Advanced Triggering Expressions

The following examples show how to use Advanced Triggering:

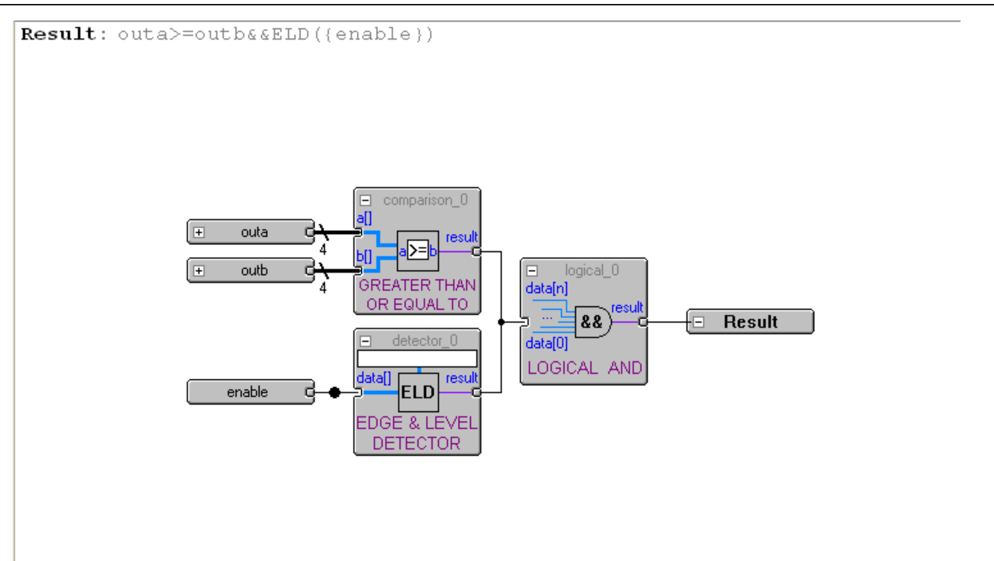
- Trigger when bus `outa` is greater than or equal to `outb` (Figure 14–27).

Figure 14–27. Bus outa is Greater Than or Equal to Bus outb



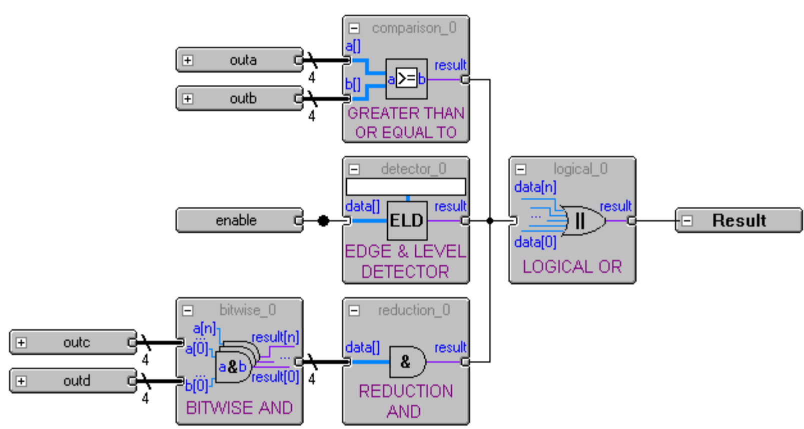
- Trigger when bus `outa` is greater than or equal to bus `outb`, and when the enable signal has a rising edge (Figure 14–28).

Figure 14–28. Enable Signal has a Rising Edge



- Trigger when bus `outa` is greater than or equal to bus `outb`, or when the enable signal has a rising edge. Or, when a bitwise `AND` operation has been performed between bus `outc` and bus `outd`, and all bits of the result of that operation are equal to 1 (Figure 14–29).

Figure 14–29. Bitwise AND Operation



Trigger Condition Flow Control

The SignalTap II Embedded Logic Analyzer offers multiple triggering conditions to give you precise control of the method data is captured into the acquisition buffers. Trigger Condition Flow allows you to define the relationship between a set of triggering conditions. The SignalTap II Embedded Logic Analyzer offers two flow control mechanisms for organizing trigger conditions:

- Sequential Triggering—This is the default triggering flow. Sequential triggering allows you to define up to 10 triggering levels that must be satisfied before the acquisition buffer finishes capturing.
- Custom State-Based Triggering—This flow allows you the greatest control over your acquisition buffer. Custom-based triggering allows you to organize trigger conditions into states based on a conditional flow that you define.

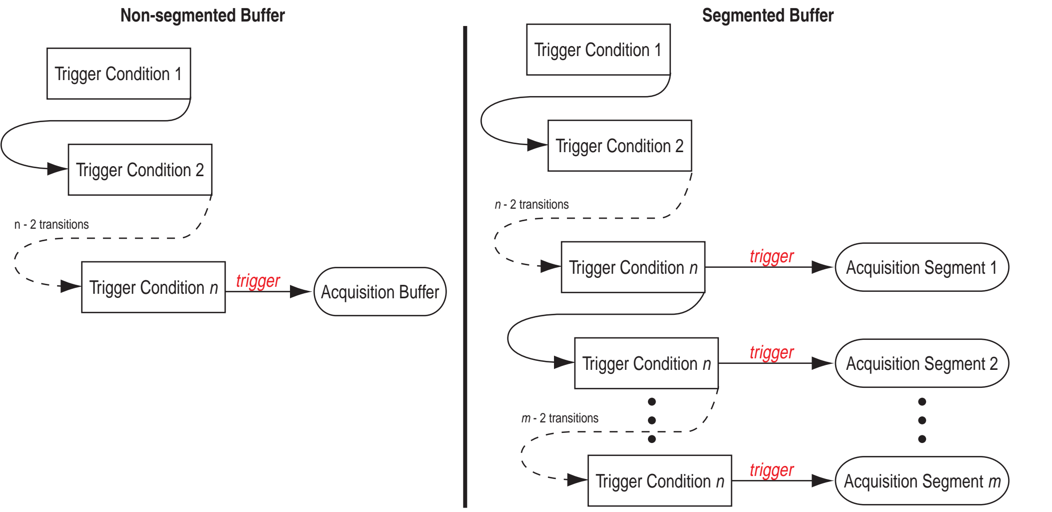
You can use either method with either a segmented or a non-segmented buffer.

Sequential Triggering

Sequential triggering flow allows you to cascade up to 10 levels of triggering conditions. The SignalTap II Embedded Logic Analyzer sequentially evaluates each of the triggering conditions. When the last triggering condition evaluates to `TRUE`, the SignalTap II Embedded Logic Analyzer triggers the acquisition buffer. For segmented buffers, every acquisition segment after the first segment triggers on the last triggering condition that you have specified. Use the Simple Sequential Triggering feature with basic triggers, advanced triggers, or a mix of both. Figure 14–30 illustrates the simple sequential triggering flow for non-segmented and segmented buffers.

The external trigger is considered as trigger level 0. The external trigger must be evaluated before the main trigger levels are evaluated.

Figure 14–30. Sequential Triggering Flow *(Note 1)*, *(2)*



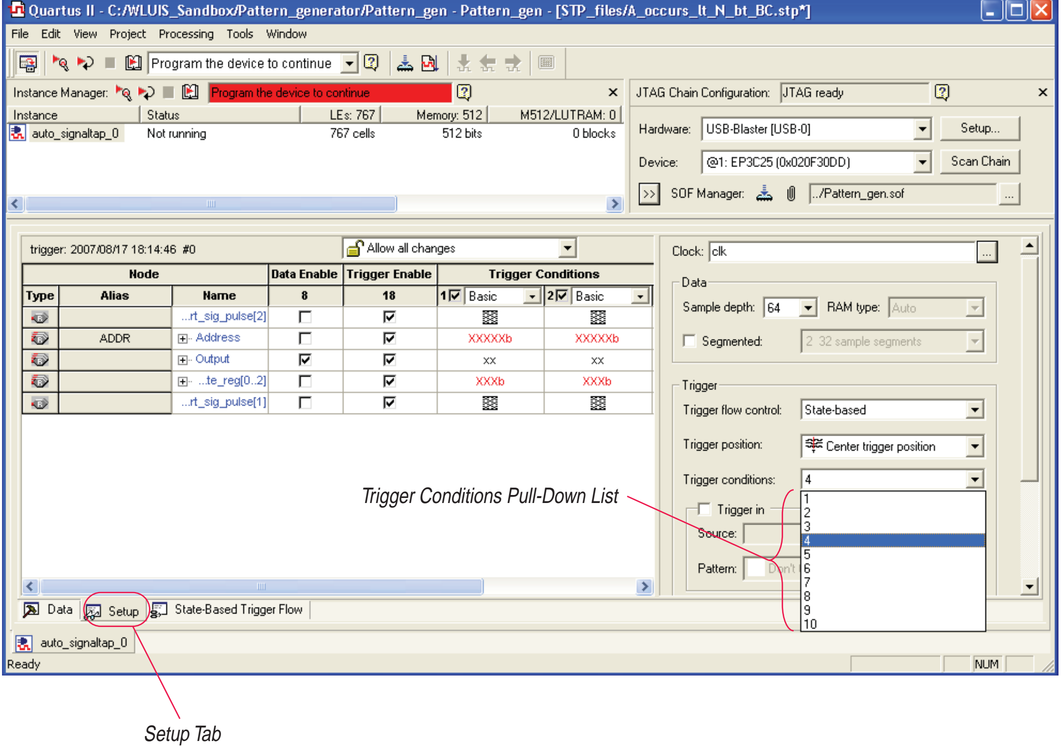
Notes to Figure 14–30:

(1) The acquisition buffer stops capture when all n triggering levels are satisfied, where $n \leq 10$.

(2) An external trigger input, if defined, is evaluated before all other defined trigger conditions are evaluated. For more information about external triggers, refer to "Using External Triggers" on page 14–51.

To configure the SignalTap II Embedded Logic Analyzer for Sequential triggering, in the SignalTap II editor on the **Trigger flow control** list, select **Sequential**. Select the desired number of trigger conditions by using the **Trigger Conditions** pull-down list. After you select the desired number of trigger conditions, configure each trigger condition in the node list. To disable any trigger condition, click the check box next to the trigger condition at the top of the column in the node list. Figure 14–31 shows the **Setup** tab for Sequential Triggering.

Figure 14–31. Setup Tab

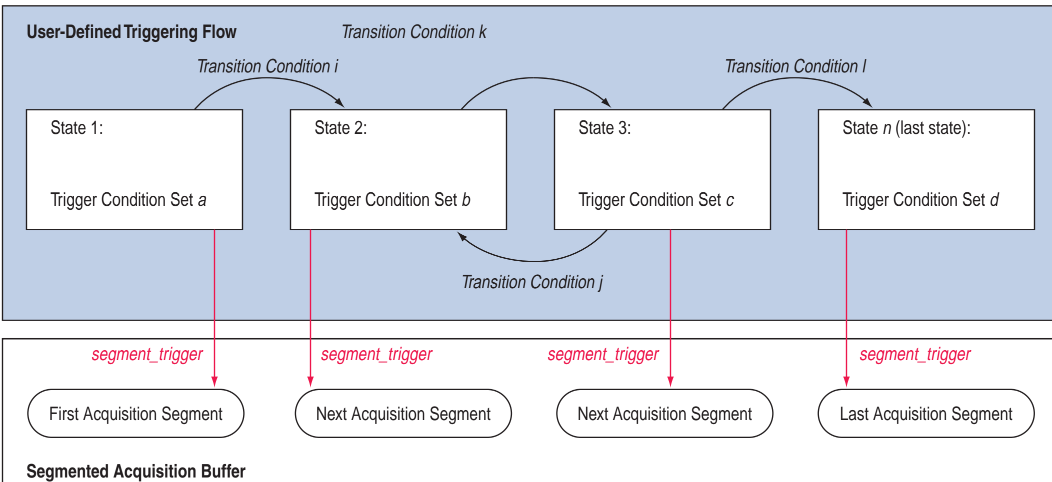


Custom State-Based Triggering

Custom State-based triggering gives you the most control of triggering condition arrangement. This flow gives you the ability to describe the relationship between triggering conditions precisely, using an intuitive GUI and the SignalTap II Trigger Flow Description Language, a simple description language based upon conditional expressions. Tooltips within the custom triggering flow GUI allow you to describe your desired flow quickly. The custom State-based triggering flow allows for more efficient use of the space available in the acquisition buffer because only specific samples of interest are captured.

Figure 14–32 illustrates the custom State-based triggering flow. Events that trigger the acquisition buffer are organized by a user-defined state diagram. All actions performed by the acquisition buffer are captured by the states and all transition conditions between the states are defined by the conditional expressions that you specify within each state.

Figure 14–32. Custom State-Based Triggering Flow *(Note 1)*, *(2)*



Notes to Figure 14–32:

(1) You are allowed up to 20 different states.

(2) An external trigger input, if defined, is evaluated before any conditions in the custom State-based triggering flow are evaluated. For more information, refer to "Using External Triggers" on page 14–51.

Each state allows you to define a set of conditional expressions. Each conditional expression is a Boolean expression dependent on a combination of triggering conditions (configured within the **Setup** tab), counters, and status flags. Counters and status flags are resources provided by the SignalTap II custom-based triggering flow.

Within each conditional expression you define a set of actions. Actions include triggering the acquisition buffer to stop capture, a modification to either a counter or status flag, or a state transition.

Trigger actions can apply to either a single segment of a segmented acquisition buffer or to the entire non-segmented acquisition buffer. Each trigger action provides you with an optional count that specifies the number of samples to be captured before stopping acquisition of the current segment. The count argument allows you to control the amount of data captured precisely before and after triggering event.

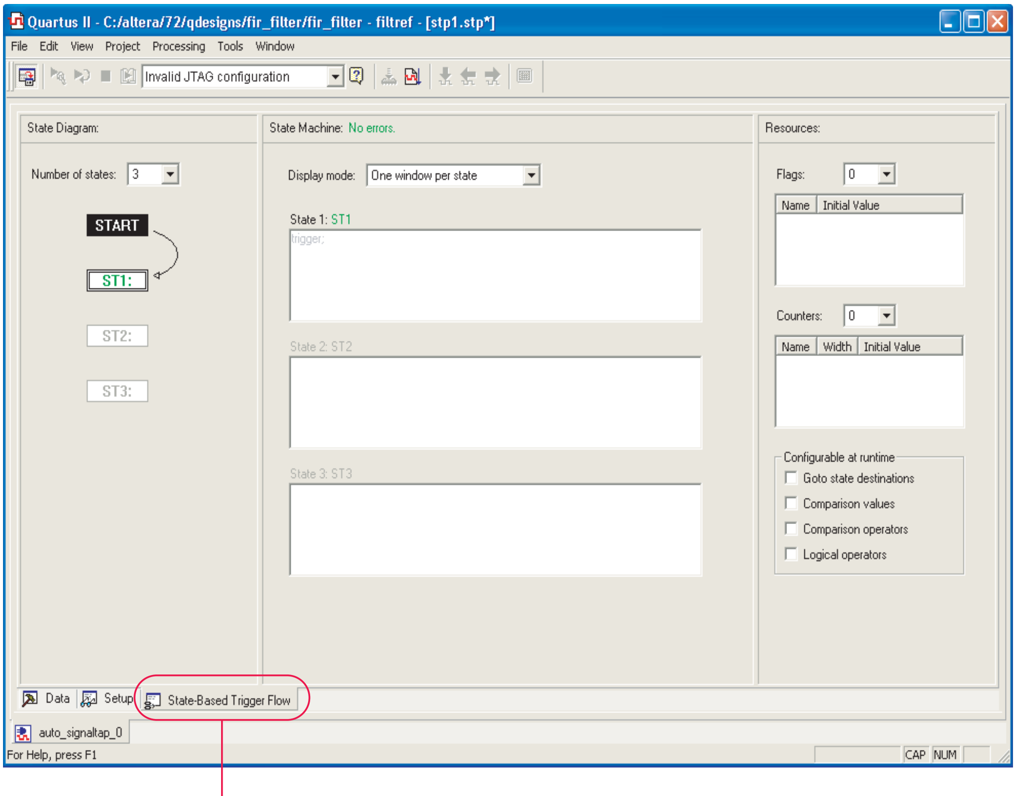
Resource manipulation actions allow you to increment and decrement counters or set and clear status flags. The counter and status flag resources are used as optional inputs in conditional expressions. Counters and status flags are useful for counting the number of occurrences of particular events and for aiding in triggering flow control.

This SignalTap II custom State-based triggering flow allows you to capture a sequence of events that may not necessarily be contiguous in time; for example, capturing a communication transaction between two devices that includes a handshaking protocol containing a sequence of acknowledgements.

The **State-Based Trigger Flow** tab is the control interface for the custom state-based triggering flow. To enable this tab, on the **Trigger Flow Control** pull-down list, select **State-based**. (Note that when the **Trigger Flow Control** option is set to **Sequential**, the **State-Based Trigger Flow** tab is hidden.)

Figure 14–33 shows the **State-Based Trigger Flow** tab.

Figure 14–33. State-Based Trigger Flow Tab



The **State-Based Trigger Flow** tab is partitioned into the following three panes:

- State Diagram Pane
- Resources Pane
- State Machine Pane

State Diagram Pane

The **State Diagram** pane provides a graphical overview of the triggering flow that you define. It shows the number of states available and the state transitions between all of the states. You can adjust the number of available states by using the pull-down menu above the graphical overview.

State Machine Pane

The **State Machine** pane contains the text entry boxes where you can define the triggering flow and actions associated with each state. You can define the triggering flow using the SignalTap II Trigger Flow Description Language, a simple language based on "if-else" conditional statements. Tooltips appear when you move the mouse over the cursor, to guide command entry into the state boxes. The GUI provides a syntax check on your flow description in real-time and highlights any errors in the text flow.

For a full description of the SignalTap II Trigger Flow Description Language, refer to "SignalTap II Trigger Flow Description Language" on page 14–42. You can also refer to the Quartus II Help.

The State Machine description text boxes default to show one text box per state. You can optionally have the entire flow description shown in a single text field. This option can be useful when copying and pasting a flow description from a template or an external text editor. To toggle between one window per state, or all states in one window, select the appropriate option under **State Display mode.**

Resources Pane

The **Resources** pane allows you to declare Status Flags and Counters for use in the conditional expressions in the Custom Triggering Flow. Actions to decrement and increment counters or to set and clear status flags are performed within the triggering flow that you define.

You can set up to 20 counters and 20 status flags. Counter and status flags values may be initialized by right-clicking the status flag or counter name after selecting a number of them from the respective pull-down list, and selecting **Set Initial Value.** To set counter width, right-click the counter name and select **Set Width**. Counters and flag values are updated dynamically after acquisition has started to assist in debugging your trigger flow specification.

Runtime Reconfigurability—The **configurable at runtime** options in the **Resources** pane allows you to configure the custom-flow control options that can be changed at runtime without requiring a recompilation. Table 14–5 contains a description of options for the State-based trigger flow that can be reconfigured at runtime.

For a broader discussion about all options that can be changed without incurring a recompile refer to "Runtime Reconfigurable Options" on page 14–63.

Table 14–5. Runtime Reconfigurable Settings, State-Based Triggering Flow

Setting	Description
Destination of goto action	Allows you to modify the destination of the state transition at runtime.
Comparison values	Allows comparison values in Boolean expressions to be modifiable at runtime. In addition, it allows the `segment_trigger` and trigger action post-fill count argument to be modifiable at runtime.
Comparison operators	Allows comparison operators in Boolean expressions to be modifiable at runtime.
Logical operators	Allows the logical operators in Boolean expressions to be modifiable at runtime.

You can restrict changes to your SignalTap configuration to include only the options that do not require a recompilation by using the pull-down menu above the trigger list in the **Setup** tab. The option **Allow trigger condition changes only** restricts changes to only the configuration settings that have the **configurable at runtime** set. With this option enabled, to modify Trigger Flow conditions in the **Custom Trigger Flow** tab, click the desired parameter in the text box and select a new parameter from the menu that appears.

The runtime configurable settings for the **Custom Trigger Flow** tab are on by default. You may get some performance advantages by disabling some of the runtime configurable options. For details about the effects of turning off the runtime modifiable options, refer to "Performance and Resource Considerations" on page 14–57.

SignalTap II Trigger Flow Description Language

The Trigger Flow Description Language is based on a list of conditional expressions per state to define a set of actions. Each line in Example 14–1 shows a language format. Keywords are shown in bold. Non-terminals are delimited by "<>" and are further explained in the following sections. Optional arguments are delimited by "[ ]" (Example 14–1).

Examples of Triggering Flow descriptions for common scenarios using the SignalTap II Custom Triggering Flow are provided in "Custom Triggering Flow Application Examples" on page 14–79.

Example 14–1. Trigger Flow Description Language Format *(Note 1)*

```
state <State_label>:
<action_list>

if( <Boolean_expression> )
<action_list>
[else if ( <boolean_expression> )
<action_list>] (1)
[else
<action_list>]
```

Note to Example 14–1:

(1) Multiple `else if` conditions are allowed.

The priority for evaluation of conditional statements is assigned from top to bottom. The *<boolean_expression>* in an if statement can contain a single event, or it can contain multiple event conditions. The `action_list` embedded within an `if` or an `else if` clause must be delimited by the begin and end tokens when the action list contains multiple statements. When the boolean expression is evaluated `TRUE`, the logic analyzer analyzes all of the commands in the action list concurrently. The possible actions include:

- Triggering the acquisition buffer
- Manipulating a counter or status flag resource
- Defining a state transition

State Labels

State labels are identifiers that can be used in the action `goto`.

`state` *<state_label>*`:` begins the description of the actions evaluated when this state is reached.

The description of a state ends with the beginning of another state or the end of the whole trigger flow description.

Boolean_expression

`Boolean_expression` is a collection of logical operators, relational operators, and their operands that evaluate into a Boolean result. Depending on the operator, the operand can be a reference to a trigger condition, a counter and a register, or a numeric value. Within an expression, parentheses can be used to group a set of operands.

Logical operators accept any boolean expression as an operand. The supported logical operators are shown in Table 14–6.

Table 14–6. Logical Operators

Operator	Description	Syntax
!	NOT operator	! expr1
&&	AND operator	expr1 && expr2
||	OR operator	expr1 || expr2

Relational operators are performed on counters or status flags. The comparison value—the right operator—must be a numerical value. The supported relational operators are shown in Table 14–7.

Table 14–7. Relational Operators

Operator	Description	Syntax *(Note 1) (2)*
>	Greater than	*<identifier> > <numerical_value>*
>=	Greater than or Equal to	*<identifier> >= <numerical_value>*
==	Equals	*<identifier> == <numerical_value>*
!=	Does not equal	*<identifier> != <numerical_value>*
<=	Less than or equal to	*<identifier> <= <numerical_value>*
<	Less than	*<identifier> < <numerical_value>*

Notes to Table 14–7:

(1) *<identifier>* indicates a counter or status flag.

(2) *<numerical_value>* indicates an integer.

Action_list

Action_list is a list of actions that can be performed when a state is reached and a condition is also satisfied. If more than one action is specified, they must be enclosed by `begin` and `end`. The actions can be categorized as resource manipulation actions, buffer control actions, and state transition actions. Each action is terminated by a semicolon (`;`).

Resource Manipulation Action

The resources used in the trigger flow description can be either counters or status flags. Table 14–8 shows the description and syntax of each action.

Table 14–8. Resource Manipulation Action (Part 1 of 2)

Action	Description	Syntax
increment	Increments a counter resource by 1	increment *<counter_identifier>*;
decrement	Decrements a counter resource by 1	decrement *<counter_identifier>*;

Table 14–8. Resource Manipulation Action (Part 2 of 2)

Action	Description	Syntax
reset	Resets counter resource to initial value	`reset <counter_identifier>;`
set	Sets a status Flag to 1	`set <register_flag_identifier>;`
clear	Sets a status Flag to 0	`clear <register_flag_identifier>;`

Buffer Control Action

Buffer control actions specify an action to control the acquisition buffer. Table 14–9 shows the description and syntax of each action.

Table 14–9. Buffer Control Action

Action	Description	Syntax
`trigger`	Stops the acquisition for the current buffer and ends analysis. This command is required in every flow definition.	`trigger <post-fill_count>;`
`segment_trigger`	Ends the acquisition of the current segment. The SignalTap II Embedded Logic Analyzer starts acquiring from the next segment on evaluating this command. If all segments are filled, the oldest segment is overwritten with the latest sample. The acquisition stops when a trigger action is evaluated. This action cannot be used in non-segmented acquisition mode.	`segment_trigger <post-fill_count>;`
`start_store`	Asserts the `write_enable` to the SignalTap II acquisition buffer. This command is active only when the State-based storage qualifier mode is enabled.	`start_store`
`stop_store`	De-asserts the `write_enable` signal to the SignalTap II acquisition buffer. This command is active only when the State-based storage qualifier mode is enabled.	`stop_store`

Both `trigger` and `segment_trigger` actions accept an optional post-fill count argument. If provided, the current acquisition acquires the number of samples provided by post-fill count and then stops acquisition. If no post-count value is specified, the trigger position for the affected buffer defaults to the trigger position specified in the **Setup** tab.

In the case of `segment_trigger`, acquisition of the current buffer stops immediately if a subsequent triggering action is issued in the next state, regardless of whether or not the post-fill count has been satisfied for the current buffer. The remaining unfilled post-count acquisitions in the current buffer are discarded and displayed as grayed-out samples in the data window.

State Transition Action

The State Transition action specifies the next state in the custom state control flow. It is specified by the `goto` command. The syntax is as follows:

```
goto <state_label>;
```

Using the State-Based Storage Qualifier Feature

When you select State-based for the storage qualifier type, the `start_store` and `stop_store` actions are enabled in the State-based trigger flow. These commands, when used in conjunction with the expressions of the State-based trigger flow, give you maximum flexibility to control data written into the acquisition buffer.

The `start_store` and `stop_store` commands can only be applied to a non-segmented buffer.

The `start_store` and `stop_store` commands function similar to the start and stop conditions when using the **start**/**stop** storage qualifier mode conditions. If storage qualification is enabled, the `start_store` command must be issued for SignalTap II to write data into the acquisition buffer. No data is acquired until the `start_store` command is performed. Also, a `trigger` command must be included as part of the trigger flow description. The `trigger` command is necessary to complete the acquisition and display the results on the waveform display.

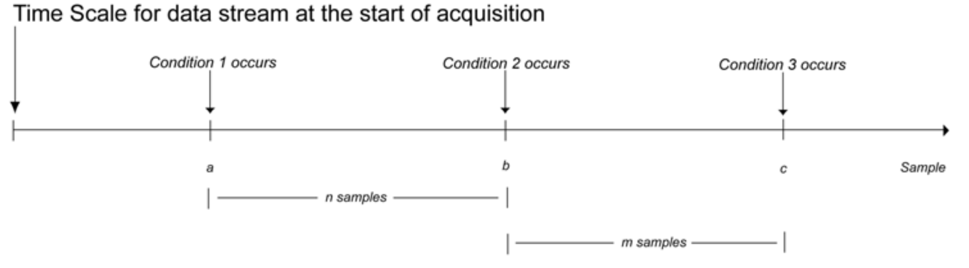
The following examples illustrate the behavior of the State-based trigger flow with the storage qualification commands.

Figure 14–34 shows a hypothetical scenario with three trigger conditions that happen at different points in time after the run analysis button is pushed. The trigger flow description in Example 14–2, when applied to the scenario shown in Figure 14–34, illustrates the functionality of the storage qualification feature for the state-based trigger flow.

Example 14–2. Trigger Flow Description 1

```
State 1: ST1:
if ( condition1 )
   start_store;
else if ( condition2 )
   trigger value;
else if ( condition3 )
   stop_store;
```

Figure 14–34. Capture Scenario for Storage Qualification with the State-Based Trigger Flow



In this example, the SignalTap II Embedded Logic Analyzer does not write into the acquisition buffer until sample a, when Condition 1 occurs. Once sample b is reached, the `trigger value` command is evaluated. The logic analyzer continues to write into the buffer to finish the acquisition. The trigger flow specifies a `stop_store` command at sample c, *m* samples after the trigger point occurs.

The logic analyzer will be able to finish the acquisition and display the contents of the waveform if it can successfully finish the post-fill acquisition samples before Condition 3 occurs. In this specific case, the capture ends if the post-fill count value is less than *m*.

If the post-fill count value specified in Trigger Flow description 1 is greater than *m* samples, the buffer pauses acquisition indefinitely, provided there is no recurrence of Condition 1 to trigger the logic analyzer to start capturing data again. The SignalTap II Embedded Logic Analyzer continues to evaluate the **stop_store** and **start_store** commands even after the **trigger** command is evaluated. If the acquisition has paused, you can manually stop and force the acquisition to trigger by using the **Stop Analysis** button. You can use counter values, flags, and the State diagram to help you gauge the execution of the trigger flow. The counter values, flags, and the current state are updated in real-time during a data acquisition.

Figure 14–35 and Figure 14–36 show a real data acquisition of the scenario described in Figure 14–33. Figure 14–35 illustrates a scenario where the data capture finishes successfully. It uses a buffer with a sample depth of 64, *m* = *n* = 10, and the post-fill count value = 5. Figure 14–36 illustrates a scenario where the logic analyzer pauses indefinitely even after a trigger condition occurs due to a `stop_store` condition. This scenario uses a sample depth of 64, with *m* = *n* = 10 and post-fill count = 15.

Figure 14–35. Storage Qualification with Post-Fill Count Value Less than *m* (Acquisition successfully completes)

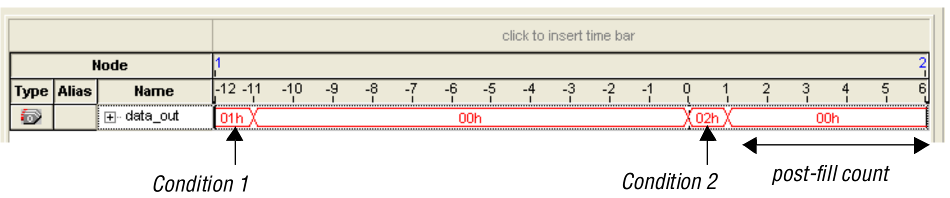


Figure 14–36. Storage Qualification with Post-Fill Count Value Greater than *m* (Acquisition indefinitely paused)

Quartus II
File Edit View Project Processing Tools Window
Instance Manager: Acquisition in progress

Instance	Status	LEs: 862	Memory: 896	M512,MLAB: 0/94	M4K,M9K: 2/
auto_signaltap_0	Acquiring post-trigger data	862 cells	896 bits	0 blocks	1 bloc

(2)

State Diagram:
Number of states: 1
Current state: ST1
START
ST1:

State Machine: No errors.
Display mode: One window per state
State 1: ST1

```
if (condition1 )
begin
    start_store;
    set f1;          (1)
end

else if (condition2 )
begin
    trigger 10;
    set f2;          (1)
end

else if ( condition3 )
begin
    stop_store;
    set f3;          (1)
end
```

Resources:
Flags: 3

Name	Initial Va...	Current Value
f1	0	1
f2	0	1
f3	0	1

(3)

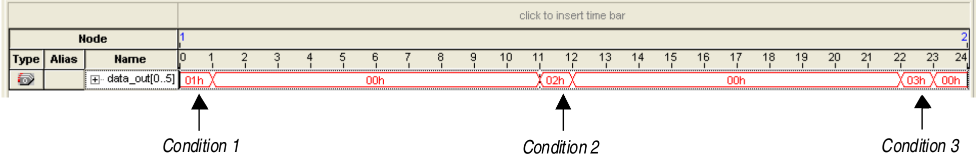
Counters: 1

Name	Width	Initial Value	Cur
c1	8	0	0

Configurable at runtime
Goto state destinations

Data Setup State-Based Trigger Flow
auto_signaltap_0
For Help, press F1 NUM

Waveform after forcing the analysis to stop



(1) Flags added to trigger flow description to help gauge the execution during runtime.
(2), (3) Status bar and current value fields update during an acquisition to provide real time status of the data acquisition.

The combination of using counters, Boolean and relational operators in conjunction with the `start_store` and `stop_store` commands can give a clock-cycle level of resolution to controlling the samples that get written into the acquisition buffer. Example 14–3 shows a trigger flow description that skips three clock cycles of samples after hitting condition 1. Figure 14–37 shows the data transaction on a continuous capture and Figure 14–39 shows the data capture with the Trigger flow description in Example 14–3 applied.

Example 14–3. Trigger Flow Description 2

```
State 1: ST1
start_store
if ( condition1 )
begin
   stop_store;
   goto ST2;
end


State 2: ST2
if (c1 < 3)
   increment c1; //skip three clock cycles; c1 initialized to 0

else if (c1 == 3)
begin
   start_store; //start_store necessary to enable writing to finish
               //acquisition
   trigger;
end
```

Figure 14–37. Continuous Capture of Data Transaction for Example 2

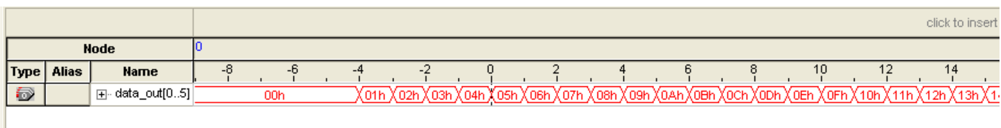
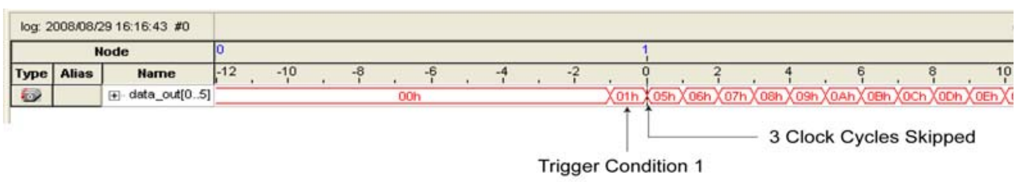


Figure 14–38. Capture of Data Transaction with Trigger Flow Description Applied



Specifying the Trigger Position

The SignalTap II Embedded Logic Analyzer allows you to specify the amount of data that is acquired before and after a trigger event. You can set the trigger position independently between a Runtime and Power-Up Trigger. Select the desired ratio of pre-trigger data to post-trigger data by choosing one of the following ratios:

- **Pre**—This selection saves signal activity that occurred after the trigger (12% pre-trigger, 88% post-trigger).
- **Center**—This selection saves 50% pre-trigger and 50% post-trigger data.
- **Post**—This selection saves signal activity that occurred before the trigger (88% pre-trigger, 12% post-trigger).

These pre-defined ratios apply to both non-segmented buffers and segmented buffers.

If you use the custom-state based triggering flow, you can specify a custom trigger position. The `segment_trigger` and trigger actions accept a post-fill count argument. The post-fill count specifies the number of samples to capture before stopping data acquisition for the non-segmented buffer or a data segment when using the `trigger` and `segment_trigger` commands, respectively. When the captured data is displayed in the SignalTap II data window, the trigger position appears as the number of post-count samples from the end of the acquisition segment or buffer. Refer to Equation 14–1:

Equation 14–1.

$$\text{Sample Number of Trigger Position} = (N - \text{Post-Fill Count})$$

In this case, N is the sample depth of either the acquisition segment or non-segmented buffer.

For segmented buffers, the acquisition segments that have a post-count argument defined use the post-count setting. Segments that do not have a post-count setting default to the trigger position ratios defined in the **Setup** tab.

For more details about the custom State-based triggering flow, refer to "Custom State-Based Triggering" on page 14–38 and "Custom State-Based Triggering" on page 14–38.

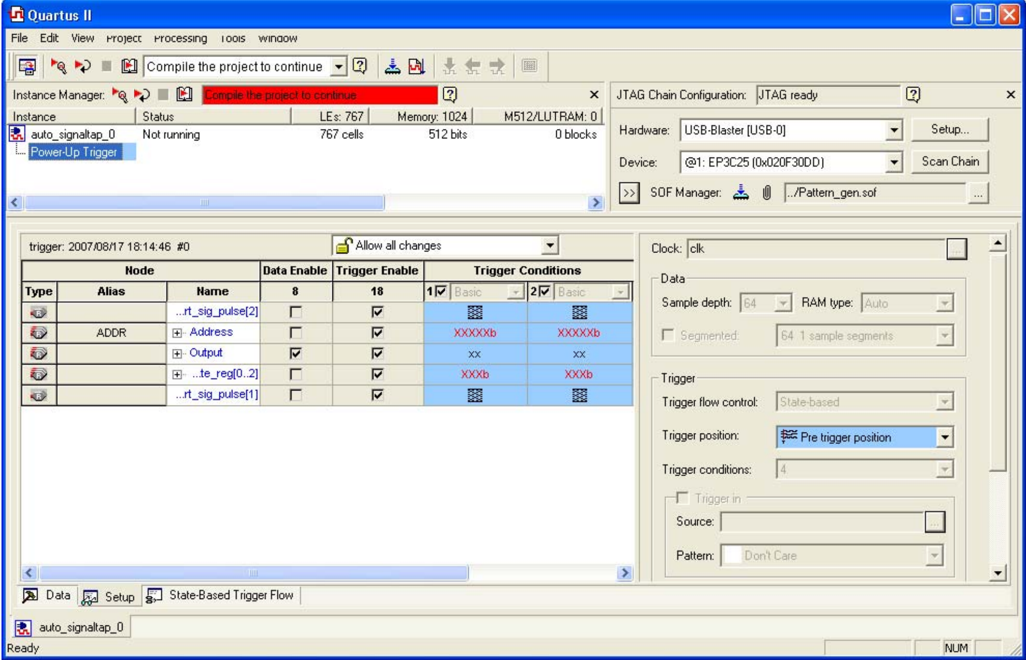
Creating a Power-Up Trigger

Typically, the SignalTap II Embedded Logic Analyzer is used to trigger on events that occur during normal device operation. You start an analysis manually once the target device is fully powered on and the device's JTAG connection is available. However, there may be cases when you would like to capture trigger events that occur during device initialization, immediately after the FPGA is powered on or reset. With the SignalTap II Power-Up Trigger feature, you arm the SignalTap II Embedded Logic Analyzer and capture data immediately after device programming.

Enabling a Power-Up Trigger

You can add a different Power-Up Trigger to each logic analyzer instance in the SignalTap II Instance Manager. To enable the Power-Up Trigger for a logic analyzer instance, right-click the instance and click **Enable Power-Up Trigger**, or select the instance, and on the **Edit** menu, click **Enable Power-Up Trigger**. To disable a Power-Up Trigger, click **Disable Power-Up Trigger** in the same locations. Power-Up Trigger is shown as a child instance below the name of the selected instance with the default trigger conditions set in the node list. Figure 14–39 shows the SignalTap II Editor when Power-Up Trigger is enabled.

Figure 14–39. SignalTap II Editor with Power-Up Trigger Enabled



Managing and Configuring Power-Up and Runtime Trigger Conditions

When the Power-Up Trigger is enabled for a logic analyzer instance, you can create basic and advanced trigger conditions for it in the same way you do with the regular trigger, also called the Run-Time Trigger. Power-Up Trigger conditions that you can adjust are color coded light blue, while Run-Time Trigger conditions remain white. Since each instance now has two sets of trigger conditions—the Power-Up Trigger and the Run-Time Trigger—you can differentiate between the two with color coding. To switch between the trigger conditions of the Power-Up Trigger and the Run-Time Trigger, double-click the instance name or the Power-Up Trigger name in the Instance Manager.

You cannot make changes to Power-Up Trigger conditions that would normally require a full recompile with Runtime Trigger conditions, such as adding signals, deleting signals, or changing between basic and advanced triggers. For these changes to be applied to the Power-Up Trigger conditions, first make the changes using the Runtime Trigger conditions.

Any change made to the Power-Up Trigger conditions requires that the SignalTap II Embedded Logic Analyzer be recompiled, even if a similar change to the Runtime Trigger conditions does not require a recompilation.

While creating or making changes to the trigger conditions for the Run-Time Trigger or the Power-Up Trigger, you may want to copy these conditions to the other trigger. This enables you to look for the same trigger during both power-up and runtime. To do this, right-click the instance name or the Power-Up Trigger name in the Instance Manager and click **Duplicate Trigger**, or select the instance name or the Power-Up Trigger name and on the **Edit** menu, click **Duplicate Trigger**.

For information about running the SignalTap II Embedded Logic Analyzer instance with a Power-Up Trigger enabled, refer to "Running with a Power-Up Trigger" on page 14–62.

Using External Triggers

You can create a trigger input that allows you to trigger the SignalTap II Embedded Logic Analyzer from an external source. The external trigger input behaves like trigger condition 1. It is evaluated and must be `TRUE` before any other configured trigger conditions are evaluated. The analyzer can also supply a signal to trigger external devices or other SignalTap II instances. These features allow you to synchronize external logic analysis equipment with the internal logic analyzer. Power-Up Triggers can use the external triggers feature, but they must use the same source or target signal as their associated Run-Time Trigger.

Trigger In

To use Trigger In, perform the following steps:

1. In the SignalTap II Editor, click the **Setup** tab.
2. If a Power-Up Trigger is enabled, ensure you are viewing the Runtime Trigger conditions.
3. In the **Signal Configuration** pane, turn on **Trigger In**.
4. In the **Pattern** list, select the condition you want to act as your trigger event. You can set this separately for Runtime or Power-Up Trigger.
5. Click Browse next to the **Source** field in the **Trigger In** pane (Figure 14–41 on page 14–53). The **Node Finder** dialog box appears.
6. In the **Node Finder** dialog box, select the signal (either an input pin or an internal signal) that you want to drive the Trigger In source and click **OK**.

   If you type a new signal name in the **Source** field, you create a new node that you can assign to an input pin in the Pin Planner or Assignment editor. If you leave the **Source** field blank, a default name is entered in the form `auto_stp_trigger_in_`<*SignalTap instance number*>.

Trigger Out

To use Trigger Out, perform the following steps:

1. In the SignalTap II Editor, click the **Setup** tab.
2. If a Power-Up trigger is enabled, ensure you are viewing the Runtime Trigger conditions.
3. To signify that the trigger event is occurring, in the **Signal Configuration** pane, turn on **Trigger Out** (refer to Figure 14–40 on page 14–52).
4. In the **Level** list, select the condition you want. You can set this separately for a Run-Time or a Power-Up Trigger.

5. Type a new signal name in the **Target** field. A new node name is created that you must assign to an output pin in the Pin Planner or Assignment Editor.

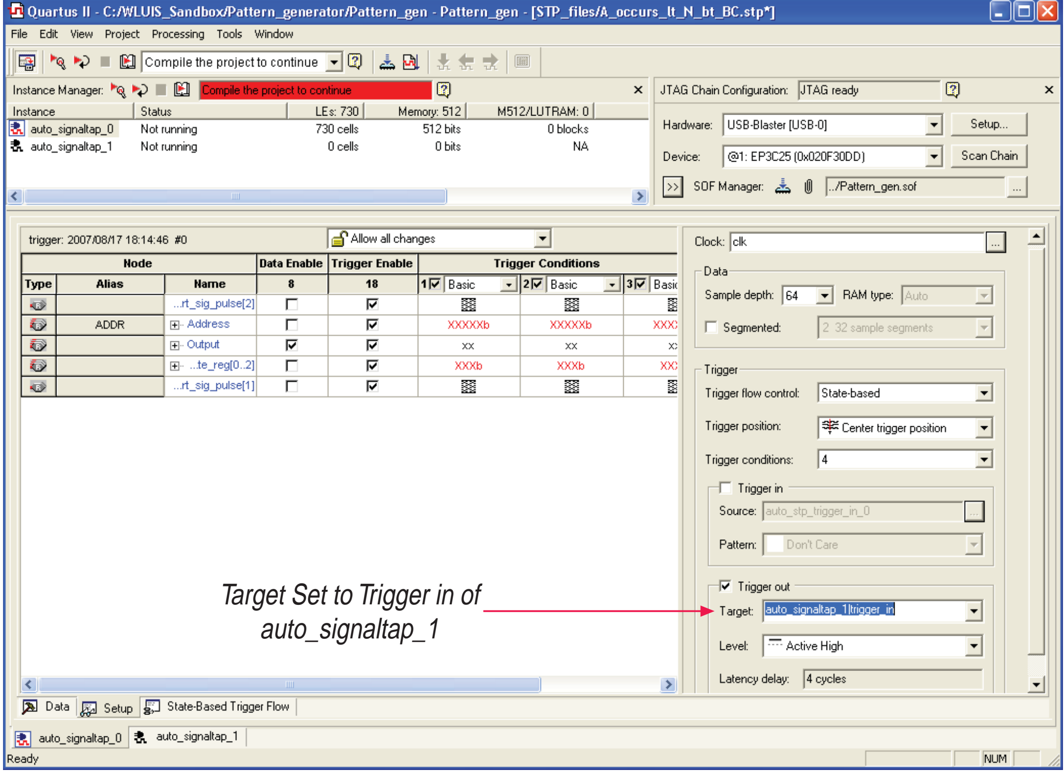
   If you leave the **Target** field blank, a default name is entered in the form `auto_stp_trigger_out_`<*SignalTap instance number*>. When the logic analyzer triggers, a signal at the level you indicated is output on the pin you assigned to the new node.

Using the Trigger Out of One Analyzer as the Trigger In of Another Analyzer

An advanced feature of the SignalTap II Embedded Logic Analyzer is the ability to use the Trigger Out of one analyzer as the Trigger In to another analyzer. This feature allows you to synchronize and debug events that occur across multiple clock domains.

To perform this operation, first enable the **Trigger Out** of the source logic analyzer instance. On the Trigger Out **Target** list, select the targeted logic analyzer instance. For example, if the instance named `auto_signaltap_0` should trigger `auto_signaltap_1`, select `auto_signaltap_1|trigger_in` from the list (Figure 14–40).

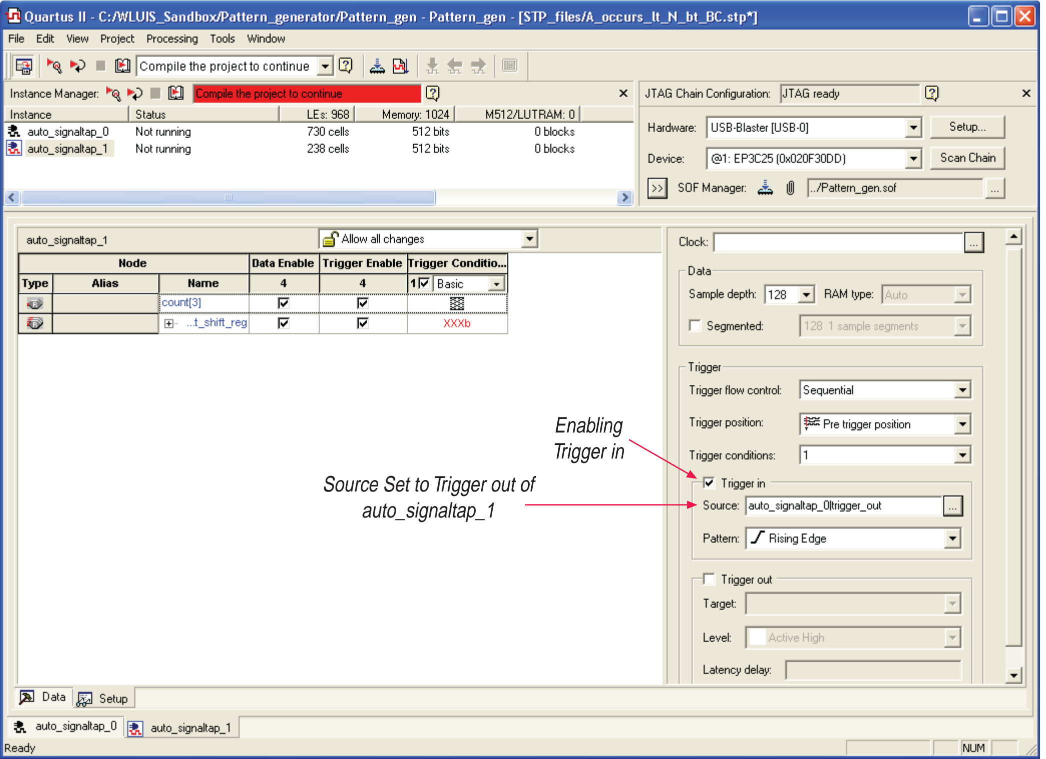
Figure 14–40. Configuring the Trigger Out Signal



■ This automatically enables the Trigger In of the targeted logic analyzer instance and fills in the Trigger In **Source** field with the Trigger Out signal from the source logic analyzer instance. In this example, `auto_signaltap_0` is targeting `auto_signaltap_1`. The Trigger In Source field of `auto_signaltap_1` is automatically filled in with `auto_signaltap_0`|trigger_out (Figure 14–41).

© March 2009 Altera Corporation

Figure 14–41. Configuring the Trigger In Signal



Compile the Design

When you add an **.stp** file to your project, the SignalTap II Embedded Logic Analyzer becomes part of your design. You must compile your project to incorporate the SignalTap II logic and enable the JTAG connection that is used to control the logic analyzer. When you are debugging with a traditional external logic analyzer, it is often necessary to make changes to the signals monitored as well as the trigger conditions. Because these adjustments often translate into recompilation time when using the SignalTap II Embedded Logic Analyzer, use the SignalTap II Embedded Logic Analyzer feature along with incremental compilation in the Quartus II software to reduce time spent recompiling.

Faster Compilations with Quartus II Incremental Compilation

To use incremental compilation with the SignalTap II Embedded Logic Analyzer, perform the following steps:

1. Enable **Full Incremental Compilation** for your design.
2. Assign design partitions.
3. Set partitions to the proper preservation levels.
4. Enable **SignalTap** for your design.
5. Add signals to **SignalTap** using the appropriate netlist filter in the node finder (either SignalTap II: pre-synthesis or SignalTap II: post-fitting).

When you compile your design with an **.stp** file, the `sld_signaltap` and `sld_hub` entities are automatically added to the compilation hierarchy. These two entities are the main components of the SignalTap II Embedded Logic Analyzer, providing the trigger logic and JTAG interface required for operation.

Incremental compilation enables you to preserve the synthesis and fitting results of your original design and add the SignalTap II Embedded Logic Analyzer to your design without recompiling your original source code. This feature is also useful when you want to modify the configuration of the **.stp** file. For example, you can modify the buffer sample depth or memory type without performing a full compilation after the change is made. Only the SignalTap II Embedded Logic Analyzer, configured as its own design partition, must be recompiled to reflect the changes.

To use incremental compilation, first enable **Full Incremental Compilation** for your design if it is not already enabled, assign design partitions if necessary, and set the design partitions to the correct preservation levels. Incremental compilation is the default setting for new projects in the Quartus II software, so you can establish design partitions immediately in a new project. However, it is not necessary to create any design partitions to use the SignalTap II incremental compilation feature. When your design is set up to use full incremental compilation, the SignalTap II Embedded Logic Analyzer acts as its own separate design partition. You can begin taking advantage of incremental compilation by using the **SignalTap II: post-fitting filter** in the Node Finder to add signals for logic analysis.

Enabling Incremental Compilation for Your Design

To enable incremental compilation if it is not already enabled, perform the following steps:

1. On the Assignments menu, click the **Design Partitions** window.
2. In the **Incremental Compilation** list, select **Full Incremental Compilation**.
3. Create user-defined partitions if desired and set the Netlist Type to **Post-fit** for all partitions.

The netlist type for the top-level partition defaults to **source**. To take advantage of incremental compilation, set the Netlist types for the partitions you wish to tap as **Post-fit**.

4. On the Processing menu, click **Start Compilation**, or, on the toolbar, click **Start Compilation**.

Your project is fully compiled the first time, establishing the design partitions you have created. When enabled for your design, the SignalTap II Embedded Logic Analyzer is always a separate partition. After the first compilation, you can use the SignalTap II Embedded Logic Analyzer to analyze signals from the post-fit netlist. If your partitions are set correctly, subsequent compilations due to SignalTap II settings are able to take advantage of the shorter compilation times.

For more information about configuring and performing incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Using Incremental Compilation with the SignalTap II Embedded Logic Analyzer

The SignalTap II Embedded Logic Analyzer is automatically configured to work with the incremental compilation flow. For all signals that you want to connect to the SignalTap II Embedded Logic Analyzer from the post-fit netlist, set the netlist type of the partition containing the desired signals to **Post-Fit** or **Post-Fit (Strict)** with a Fitter Preservation Level of **Placement and Routing** using the Design Partitions window. Use the **SignalTap II: post-fitting filter** in the **Node Finder** to add the signals of interest to your SignalTap II configuration file. If you want to add signals from the pre-synthesis netlist, set the netlist type to **Source File** and use the **SignalTap II: pre-synthesis filter** in the **Node Finder**. Do not use the netlist type **Post-Synthesis** with the SignalTap II Embedded Logic Analyzer.

Be sure to conform to the following guidelines when using post-fit and pre-synthesis nodes:

- Read all incremental compilation guidelines to ensure the proper partition of a project.
- To speed compile time, use only post-fit nodes for partitions set to preservation-level post-fit.
- Do not mix pre-synthesis and post-fit nodes in any partition. If you must tap pre-synthesis nodes for a particular partition, make all tapped nodes in that partition pre-synthesis nodes and change the netlist type to **source** in the design partitions window.

Node names may be different between a pre-synthesis netlist and a post-fit netlist. In general, registers and user input signals share common names between the two netlists. During compilation, certain optimizations change the names of combinational signals in your RTL. If the type of node name chosen does not match the netlist type, the compiler may not be able to find the signal to connect to your SignalTap II Embedded Logic Analyzer instance for analysis. The compiler issues a critical warning to alert you of this scenario. The signal that is not connected is tied to ground in the **SignalTap II data** tab.

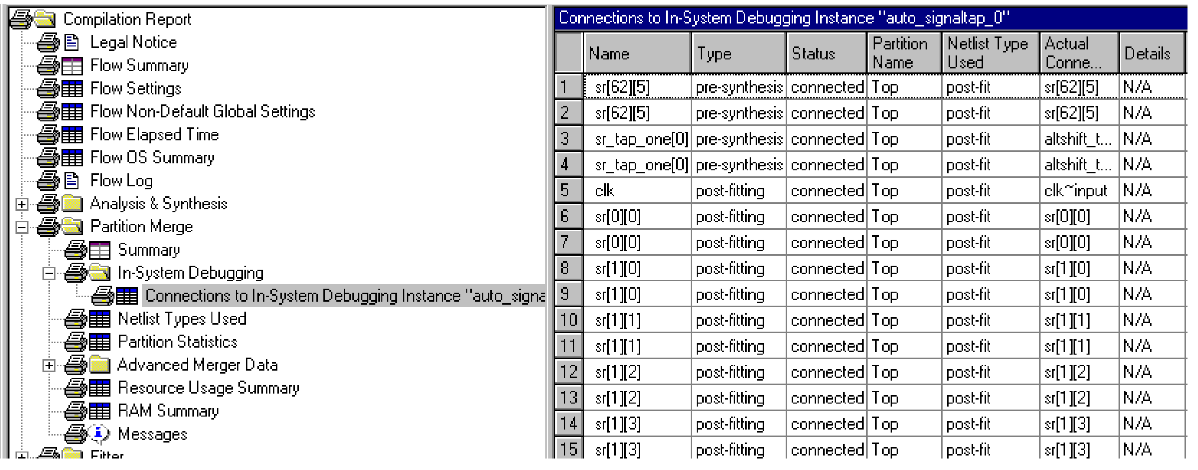
If you do use incremental compile flow with the SignalTap II Embedded Logic Analyzer and source file changes are necessary, be aware that you may have to remove compiler-generated post-fit net names. Source code changes force the affected partition to go through resynthesis. During synthesis, the compiler cannot find compiler-generated net names from a previous compilation.

Altera recommends using only registered and user-input signals as debugging taps in your **.stp** file whenever possible.

Both registered and user-supplied input signals share common node names in the pre-synthesis and post-fit netlist. As a result, using only registered and user-supplied input signals in your **.stp** file limits the changes you need to make to your SignalTap configuration.

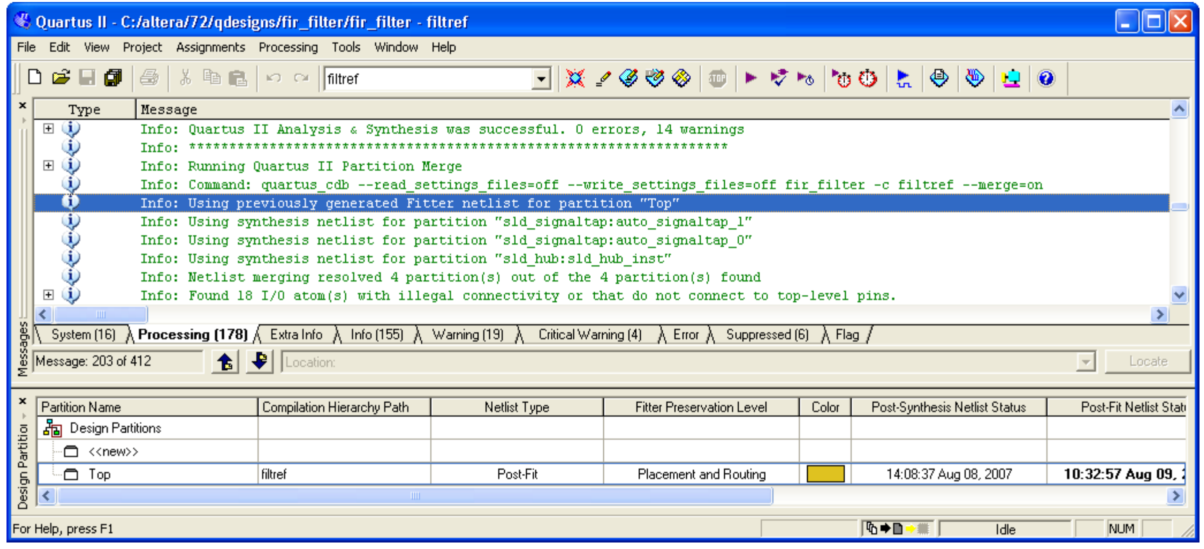
You can check the nodes that are connected to each SignalTap II instance using the In-System Debugging compilation reports. These reports list each node name you selected to connect to a SignalTap II instance, the netlist type used for the particular connection, and the actual node name used after compilation. If incremental compile is turned off, the In-System Debugging reports are located in the Analysis & Synthesis folder. If incremental compile is turned on, this report is located in the Partition Merge folder. Figure 14–42 shows an example of an In-System Debugging compilation report for a design using incremental compilation.

Figure 14–42. Compilation Report Showing Connectivity to SignalTap II Instance



To verify that your original design was not modified, examine the messages in the **Partition Merge** section of the Compilation Report. Figure 14–43 shows an example of the messages displayed.

Figure 14–43. Compilation Report Messages



Unless you make changes to your design partitions that require recompilation, only the SignalTap II design partition is recompiled. If you make subsequent changes to only the **.stp** file, only the SignalTap II design partition must be recompiled, reducing your recompilation time.

Preventing Changes Requiring Recompilation

You can configure the **.stp** file to prevent changes that normally require recompilation. To do this, select a lock mode from above the node list in the **Setup** tab. To lock your configuration, choose to allow only trigger condition changes, regardless of whether you use incremental compilation.

For more information about the use of lock modes, refer to the Quartus II Help.

Timing Preservation with the SignalTap II Embedded Logic Analyzer

In addition to verifying functionality, timing closure is one of the most crucial processes in successfully completing a design. When you compile a project with a SignalTap II Embedded Logic Analyzer without the use of incremental compilation, you add IP to your existing design. Therefore, you can affect the existing placement, routing, and timing of your design. To minimize the effect that the SignalTap II Embedded Logic Analyzer has on your design, Altera recommends that you use incremental compilation for your project. Incremental compilation is the default setting in new designs and can be easily enabled and configured in existing designs. With the SignalTap II Embedded Logic Analyzer in its own design partition, it has little to no affect on your design.

In addition to using the incremental compilation flow for your design, you can use the following techniques to help maintain timing:

- Avoid adding critical path signals to your **.stp** file.
- Minimize the number of combinational signals you add to your **.stp** file and add registers whenever possible.
- Specify an f_{MAX} constraint for each clock in your design.

For an example of timing preservation with the SignalTap II Embedded Logic Analyzer, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Performance and Resource Considerations

There is an inherent trade-off between runtime flexibility of the SignalTap II Embedded Logic Analyzer, timing performance of the SignalTap II Embedded Logic Analyzer, and resource usage. The SignalTap II Embedded Logic Analyzer allows you to select the runtime configurable parameters to balance the need for runtime flexibility, speed, and area. The default values have been chosen to provide maximum flexibility so you can reach debugging closure as quickly as possible; however, you can adjust these settings to determine whether there is a more optimal configuration for your design.

The suggestions in this section provide some tips to provide extra timing slack if you have determined that the SignalTap II logic is in your critical path, or to alleviate the resource requirements that the SignalTap II Embedded Logic Analyzer consumes if your design is resource-constrained.

If SignalTap II logic is part of your critical path, the following suggestions can help to speed up the performance of the SignalTap II Embedded Logic Analyzer:

- **Disable runtime configurable options**—Certain resources are allocated to accommodate for runtime flexibility. If you are using either advanced triggers or State-based triggering flow, disable runtime configurable parameters for a boost in f_{MAX} of the SignalTap II logic. If you are using State-based triggering flow, try disabling the **Goto state destination** option and performing a recompilation before disabling the other runtime configurable options. The **Goto state destination** option has the greatest impact on f_{MAX}, as compared to the other runtime configurable options.
- **Minimize the number of signals that have Trigger Enable selected**—All of the signals that you add to the **.stp** file have Trigger Enable turned on. Turn off Trigger Enable for signals that you do not plan to use as triggers.
- **Turn on Physical Synthesis for register retiming**—If you have a large number of triggering signals enabled (greater than the number of inputs that would fit in a LAB) that fan-in to logic gate-based triggering condition, such as a basic trigger condition or a logical reduction operator in the advanced trigger tab, turn on the **Perform register retiming** option. This can help balance combinational logic across LABs.

If your design is resource constrained, the following suggestions can help to reduce the amount of logic or memory used by the SignalTap II Embedded Logic Analyzer:

- **Disable runtime configurable options**—Disabling runtime configurability for advanced trigger conditions or runtime configurable options in the State-based triggering flow results in using fewer LEs.
- **Minimize the number of segments in the acquisition buffer**—You can reduce the number of logic resources used for the SignalTap II Embedded Logic Analyzer by limiting the number of segments in your sampling buffer to only those required.
- **Disable the Data Enable for signals that are used for triggering only**—By default, both the **data enable** and **trigger enable** options are selected for all signals. Turning off the data enable option for signals used as trigger inputs only saves on memory resources used by the SignalTap II Embedded Logic Analyzer.

Because performance results are design-dependent, try these options in different combinations until you achieve the desired balance between functionality, performance, and utilization.

For more information about area and timing optimization, refer the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Program the Target Device or Devices

After your project, including the SignalTap II Embedded Logic Analyzer, is compiled, configure the FPGA target device. When you are using the SignalTap II Embedded Logic Analyzer for debugging, configure the device from the **.stp** file instead of the Quartus II Programmer. Because you configure from the **.stp** file, you can open more than one **.stp** file and program multiple devices to debug multiple designs simultaneously.

The settings in an **.stp** file must be compatible with the programming **.sof** file used to program the device. An **.stp** file is considered compatible with an **.sof** file when the settings for the logic analyzer, such as the size of the capture buffer and the signals selected for monitoring or triggering, match the way the target device will be programmed. If the files are not compatible, you can still program the device, but you cannot run or control the logic analyzer from the SignalTap II Editor.

To ensure programming compatibility, make sure to program your device with the latest **.sof** file created from the most recent compilation.

Before starting a debugging session, do not make any changes to the **.stp** file settings that would require the project to be recompiled. You can check the SignalTap II status display at the top of the Instance Manager to verify whether a change you made requires the design to be recompiled, producing a new **.sof** file. This gives you the opportunity to undo the change, so that a recompilation is not necessary. To prevent any such changes, enable lock mode in the **.stp** file.

Although the Quartus II project is not required, it is recommended. The project database contains information about the integrity of the current SignalTap II session. Without the project database, there is no way to verify that the current **.stp** file matches the **.sof** file that is downloaded to the device. If you have an **.stp** file that does not match the **.sof** file, you will see incorrect data captured in the SignalTap II Embedded Logic Analyzer.

Programming a Single Device

To configure a single device for use with the SignalTap II Embedded Logic Analyzer, perform the following steps:

1. In the **JTAG Chain Configuration** pane in the SignalTap II Editor, select the connection you use to communicate with the device from the **Hardware** list. If you need to add your communication cable to the list, click **Setup** to configure your connection.
2. In the **JTAG Chain Configuration** pane, click **Browse** and select the **.sof** file that includes the compatible SignalTap II Embedded Logic Analyzer.
3. Click **Scan Chain**. The Scan Chain operation enumerates all of the JTAG devices within your JTAG chain.
4. In the **Device** list, select the device to which you want to download the design. The device list shows an ordered list of all devices in the JTAG chain.

   All of the devices are numbered sequentially according to their position in the JTAG chain, prefixed with the “@”. For example: `@1 : EP3C25 (0x020F30DD)` lists a Cyclone III device as the first device in the chain with the JTAG ID code of `0x020F30DD`.

5. Click the **Program Device** icon.

Programming Multiple Devices to Debug Multiple Designs

You can simultaneously debug multiple designs using one instance of the Quartus II software by performing the following steps:

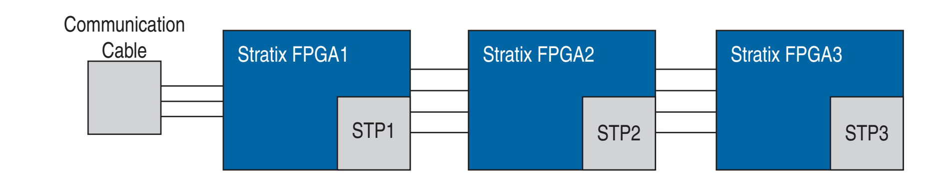
1. Create, configure, and compile each project that includes an **.stp** file.
2. Open each **.stp** file.

   ☞ You do not have to open a Quartus II project to open an **.stp** file.

3. Use the **JTAG Chain Configuration** pane controls to select the target device in each **.stp** file.
4. Program each FPGA.
5. Run each analyzer independently.

Figure 14–44 shows a JTAG chain and its associated **.stp** files.

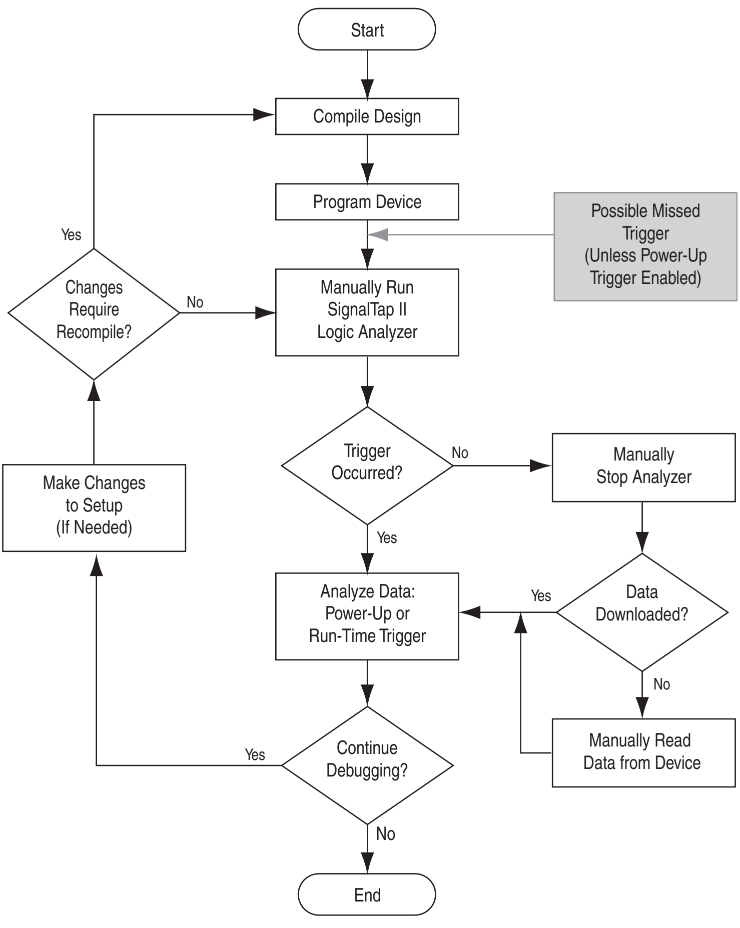
Figure 14–44. JTAG Chain



Run the SignalTap II Embedded Logic Analyzer

After the device is configured with your design that includes the SignalTap II Embedded Logic Analyzer, perform debugging operations in a manner similar to the use of an external logic analyzer. You "arm" the logic analyzer by starting an analysis. When your trigger event occurs, the captured data is stored in the memory buffer on the device and then transferred to the **.stp** file over the JTAG connection. You can also perform the equivalent of a "force trigger" that lets you view the captured data currently in the buffer without a trigger event occurring. Figure 14–45 illustrates a flow that shows how you operate the SignalTap II Embedded Logic Analyzer. The flowchart indicates where Power-Up and Runtime Trigger events occur and when captured data from these events is available for analysis.

Figure 14–45. Power-Up and Runtime Trigger Events Flowchart



The SignalTap II toolbar in the Instance Manager has four options for running the analyzer:

- **Run Analysis**—The SignalTap II Embedded Logic Analyzer runs until the trigger event occurs. When the trigger event occurs, monitoring and data capture stops when the acquisition buffer is full.
- **AutoRun Analysis**—The SignalTap II Embedded Logic Analyzer continuously captures data until the **Stop Analysis** button is clicked, ignoring all trigger event conditions.
- **Stop Analysis**—SignalTap II analysis stops. The acquired data does not appear automatically if the trigger event has not occurred.
- **Read Data**—Captured data is displayed. This button is useful to view the acquired data, even if the trigger has not occurred.

Running with a Power-Up Trigger

If you have enabled and set up a Power-Up Trigger for an instance of the SignalTap II Embedded Logic Analyzer, the captured data may already be available for viewing if the trigger event occurred after device configuration. To download the captured data or to check if the Power-Up Trigger is still running, in the Instance Manager, click **Run Analysis**. If the Power-Up Trigger occurred, the logic analyzer immediately stops, and the captured data is downloaded from the device. The data can now be viewed on the **Data** tab of the SignalTap II Editor. If the Power-Up Trigger did not occur, no captured data is downloaded, and the logic analyzer continues to run. You can wait for the Power-Up Trigger event to occur, or, to stop the logic analyzer, click **Stop Analysis**.

Running with Runtime Triggers

You can arm and run the SignalTap II Embedded Logic Analyzer manually after device configuration to capture data samples based on the Runtime Trigger. You can do this immediately if there is no Power-Up Trigger enabled. If a Power-Up Trigger is enabled, you can do this after the Power-Up Trigger data is downloaded from the device or once the logic analyzer is stopped because the Power-Up Trigger event did not occur. Click **Run Analysis** in the SignalTap II Editor to start monitoring for the trigger event. You can start multiple SignalTap II instances at the same time by selecting all of the required instances before you click **Run Analysis** on the toolbar.

Unless the logic analyzer is stopped manually, data capture begins when the trigger event evaluates to `TRUE`. When this happens, the captured data is downloaded from the buffer. You can view the data in the **Data** tab of the SignalTap II Editor.

Performing a Force Trigger

Sometimes when you use an external logic analyzer or oscilloscope, you want to see the current state of signals without setting up or waiting for a trigger event to occur. This is referred to as a “force trigger” operation, because you are forcing the test equipment to capture data without regard to any set trigger conditions. With the SignalTap II Embedded Logic Analyzer, you can choose to run the analyzer and capture data immediately or run the analyzer and capture data when you want.

To run the analyzer and immediately capture data, disable the trigger conditions by turning off each **Trigger Condition** column in the node list. This operation does not require a recompilation. In the Instance Manager, click **Run Analysis**. The SignalTap II Embedded Logic Analyzer immediately triggers, captures, and downloads the data to the **Data** tab of the SignalTap II Editor. If the data does not download automatically, click **Read Data** in the Instance Manager.

If you want to choose when to capture data manually, it is not required that you disable the trigger conditions. To start the logic analyzer, click **Autorun Analysis**; to capture data, click **Stop Analysis**. If the data does not download to the **Data** tab of the SignalTap II Editor automatically, click **Read Data**.

You can also use In-System Sources and Probes in conjunction with the SignalTap II Embedded Logic Analyzer to force trigger conditions. The In-System Sources and Probes feature allows you to drive and sample values on to selected nets over the JTAG chain. For more information, refer to the *Design Debugging Using In-System Sources and Probes* chapter in volume 3 of the *Quartus II Handbook*.

Runtime Reconfigurable Options

Certain settings in the **.stp** file are changeable without incurring a recompilation when you are using Runtime Trigger mode. All Runtime Reconfigurable features are described in Table 14–10.

Table 14–10. Runtime Reconfigurable Features

Runtime Reconfigurable Setting	Description
Basic Trigger Conditions and Basic Storage Qualifier Conditions	All signals that have the trigger check box enabled can be changed to any basic trigger condition value without recompiling.
Advanced Trigger Conditions and Advanced Storage Qualifier Conditions	Many operators include runtime configurable settings. For example, all comparison operators are runtime-configurable. Configurable settings are shown with a white background in the block representation.This runtime reconfigurable option is enabled through the **Object Properties** dialog box.
Switching between a storage-qualified and a continuous acquisition	Within any storage-qualified mode, you can switch between to continuous capture mode easily without recompiling the design. You enable this feature by selecting the check box for **disable storage qualifier**.
State-based trigger flow parameters	Refer to Table 14–5 for a list of Reconfigurable State-based trigger flow options.

Runtime Reconfigurable options can potentially save time during the debugging cycle by allowing you to cover a wider possible scenario of events without the need to recompile the design. The trade-off is that there may be a slight impact to the performance and logic utilization of the SignalTap II IP core. Runtime re-configurability for Advanced Trigger Conditions and the State-based trigger flow parameters can be turned off, boosting performance and decreasing area utilization.

You can configure the **.stp** file to prevent changes that normally require recompilation. To do this, in the **Setup** tab, select **Allow Trigger Condition changes only** above the node list.

Example 14–4 illustrates a potential use case for Runtime Reconfigurable features. This example provides a storage qualified enabled State-based trigger flow description and shows how you can modify the size of a capture window at runtime without a recompile. This example gives you equivalent functionality to a segmented buffer with a single trigger condition where the segment sizes are runtime reconfigurable.

Example 14–4. Trigger Flow Description Providing Runtime Reconfigurable "Segments"

```
state ST1:
if ( condition1 && (c1 <= m) )    // each "segment"  triggers on condition
                                  //1
begin                             // m  = number of total "segments"
   start_store;
   increment c1;
   goto ST2:
End

else (c1 > m )                  //This else condition handles  the last
                                //segment.
begin
    start_store
    Trigger (n-1)
end

state ST2:
if ( c2 >= n)                  //n = number of samples to capture in each
                               //segment.
begin
    reset c2;
    stop_store;
    goto ST1;
end

else (c2 < n)
begin
    increment c2;
    goto ST2;
end
```

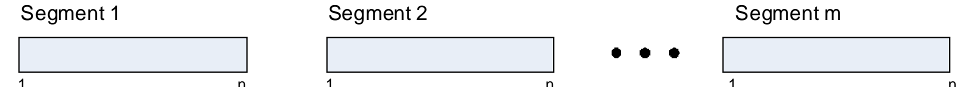
Note to Example 14–4:

(1) *m* x *n* must equal the sample depth to efficiently use the space in the sample buffer.

Figure 14–46 depicts a segmented buffer described by the trigger flow in Example 14–4.

During runtime, the values *m* and *n* are runtime reconfigurable. By changing the *m* and *n* values in the preceding trigger flow description, you can dynamically adjust the segment boundaries without incurring a recompile.

Figure 14–46. Segmented Buffer Created with Storage Qualifier and State-Based Trigger *(Note 1)*



Note to Figure 14–46:

(1) Total sample depth is fixed, where *m* x *n* must equal sample depth.

You can add states into the trigger flow description and selectively mask out specific states and enable other ones at runtime with status flags.

Example 14–5 shows a modified description of Example 14–4 with an additional state inserted. This extra state is used to specify a different trigger condition that does not use the storage qualifier feature. Status flags are inserted into the conditional statements to control the execution of the trigger flow.

Example 14–5. Modified Trigger Flow Description of Example 14–4 with Status Flags to Selectively Enable States

```
state ST1 :

if (condition2  && f1)                    //additional state added for a non-segmented
                                          //acquisition   Set f1 to enable state
begin
     start_store;
     trigger
end

else if (! f1)
     goto ST2;


state ST2:
if ( (condition1 && (c1 <= m)   && f2)    // f2  status flag used to mask state.  Set f2
                                          //to enable.
begin
   start_store;
   increment c1;
   goto ST3:
end

else (c1 > m )
   start_store
   Trigger (n-1)
end


state ST3:
if ( c2 >= n)
begin
   reset c2;
   stop_store;
   goto ST1;
end

else (c2 < n)
begin
   increment c2;
   goto ST2;
end
```

SignalTap II Status Messages

Table 14–11 describes the text messages that might appear in the SignalTap II Status Indicator in the Instance Manager before, during, and after a data acquisition. Use these messages to know the state of the logic analyzer or what operation it is performing.

Table 14–11. Text Messages in the SignalTap II Status Indicator

Message	Message Description
`Not running`	The SignalTap II Logic Analyzer is not running. There is no connection to a device or the device is not configured.
`(Power-Up Trigger) Waiting for clock` *(1)*	The SignalTap II Logic Analyzer is performing a Runtime or Power-Up Trigger acquisition and is waiting for the clock signal to transition.
`Acquiring (Power-Up) pre-trigger data` *(1)*	The trigger condition has not been evaluated yet. A full buffer of data is collected if using the non-segmented buffer acquisition mode and storage qualifier type is continuous.
`Trigger In conditions met`	Trigger In condition has occurred. The SignalTap II Embedded Logic Analyzer is waiting for the condition of the first trigger condition to occur. This can appear if Trigger In is specified.
`Waiting for (Power-up) trigger` *(1)*	The SignalTap II Logic Analyzer is now waiting for the trigger event to occur.
`Trigger level <x> met`	The condition of trigger condition *x* has occurred. The SignalTap II Embedded Logic Analyzer is waiting for the condition specified in condition *x + 1* to occur.
`Acquiring (power-up) post-trigger data` *(1)*	The entire trigger event has occurred. The SignalTap II Embedded Logic Analyzer is acquiring the post-trigger data. The amount of post-trigger data collected is user-defined between 12%, 50%, and 88% when the non-segmented buffer acquisition mode is selected.
`Offload acquired (Power-Up) data` *(1)*	Data is being transmitted to the Quartus II software through the JTAG chain.
`Ready to acquire`	The SignalTap II Embedded Logic Analyzer is waiting for the user to arm the analyzer.

Note to Table 14–11:

(1) This message can appear for both Runtime and Power-Up Trigger events. When referring to a Power-Up Trigger, the text in parentheses is added.

In segmented acquisition mode, pre-trigger and post-trigger do not apply.

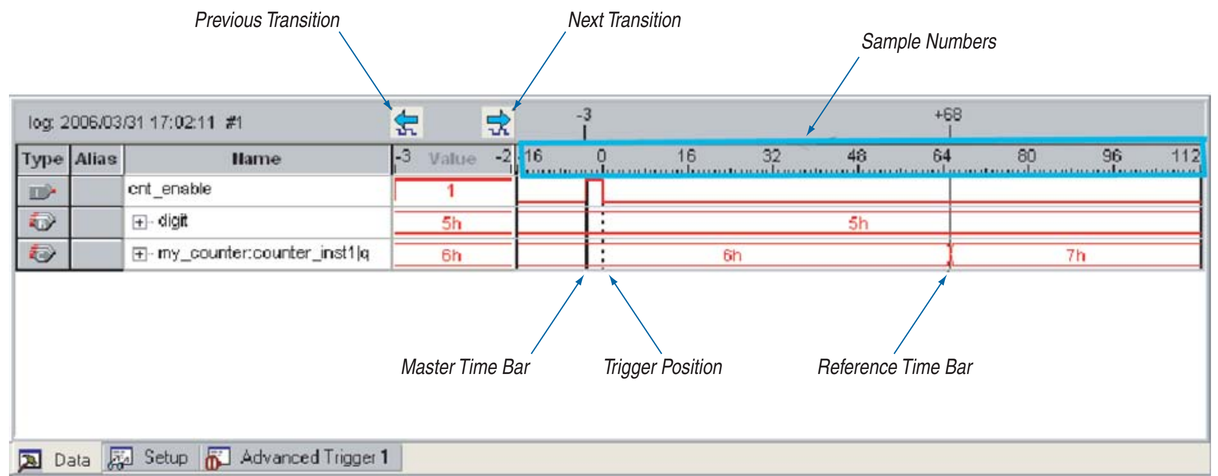
View, Analyze, and Use Captured Data

Once a trigger event has occurred or you capture data manually, you can use the SignalTap II interface to examine the data, and use your findings to help debug your design.

Viewing Captured Data

You can view captured SignalTap II data in the **Data** tab of the **.stp** file (Figure 14–47). Each row of the **Data** tab displays the captured data for one signal or bus. Buses can be expanded to show the data for each individual signal on the bus. Click on the data waveforms to zoom in on the captured data samples; right-click to zoom out.

Figure 14–47. Captured SignalTap II Data



When viewing captured data, it is often useful to know the time interval between two events. Time bars enable you to see the number of clock cycles between two samples of captured data in your system. There are two types of time bars:

- **Master Time Bar**—The master time bar's label displays the absolute time of its location in bold. The master time bar is a thick black line in the **Data** tab. The captured data has only one master time bar.
- **Reference Time Bar**—The reference time bar's label displays the time relative to the master time bar. You can create an unlimited number of reference time bars.

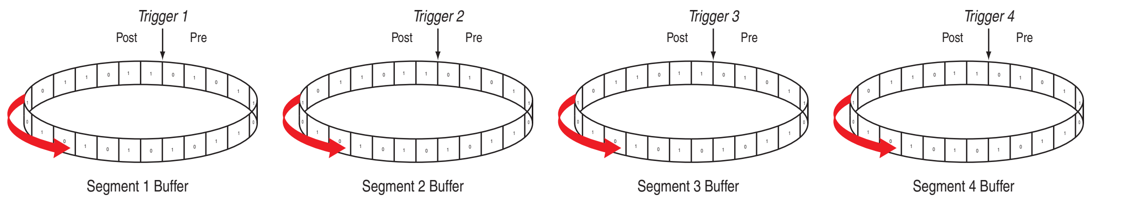
To help you find a transition of signals relative to the master time bar location, use either the **Next Transition** or the **Previous Transition** button. This aligns the master time bar with the next or previous transition of a selected signal or group of selected signals. This feature is very useful when the sample depth is very large and the rate at which signals toggle is very low.

To find bus values within the waveform quickly, use the **Find bus value** option. After you select the bus, right-click and select **Find bus value**. A dialog box appears to enter search parameters.

Capturing Data Using Segmented Buffers

Segmented Acquisition buffers allow you to perform multiple captures with a separate trigger condition for each acquisition segment. This feature allows you to capture a recurring event or sequence of events that span over a long period time efficiently. Each acquisition segment acts as a non-segmented buffer, continuously capturing data when it is activated. When you run an analysis with the **segmented buffer** option enabled, the SignalTap II Embedded Logic Analyzer performs back-to-back data captures for each acquisition segment within your data buffer. The trigger flow, or the type and order in which the trigger conditions evaluate for each buffer, is defined by either the Sequential trigger flow control or the Custom State-based trigger flow control. Figure 14–48 shows a segmented acquisition buffer with four segments represented as four separate non-segmented buffers.

Figure 14–48. Segmented Acquisition Buffer

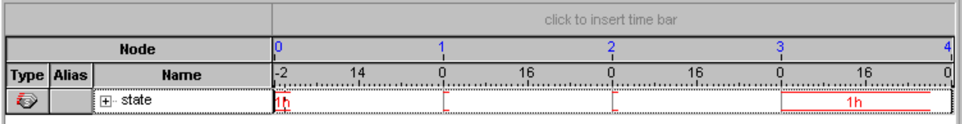


The SignalTap II Embedded Logic Analyzer finishes an acquisition with a segment, and advances to the next segment to start a new acquisition. Depending when on a trigger condition occurs, it may affect the way the data capture appears in the waveform viewer. Figure 14–48 illustrates the method data is captured. The Trigger markers in Figure 14–48—Trigger 1, Trigger 2, Trigger 3 and Trigger 4—refer to the evaluation of the `segment_trigger` and `trigger` commands in the Custom State-based trigger flow. If you are using a sequential flow, the Trigger markers refer to trigger conditions specified within the **Setup** tab.

If the Segment 1 Buffer is the active segment and Trigger 1 occurs, the SignalTap II Embedded Logic Analyzer starts evaluating Trigger 2 immediately. Data Acquisition for Segment 2 buffer starts when either Segment Buffer 1 finishes its post-fill count, or when Trigger 2 evaluates as `TRUE`, whichever condition occurs first. Thus, trigger conditions associated with the next buffer in the data capture sequence can preempt the post-fill count of the current active buffer. This is necessary so the SignalTap II Embedded Logic Analyzer can capture accurately all of the trigger conditions that have occurred. Samples that have not been used appear as a blank space in the waveform viewer.

Figure 14–49 shows an example of a capture using sequential flow control with the trigger condition for each segment set to “don’t care”. Each segment before the last captures only one sample, because the next trigger condition immediately preempts capture of the current buffer. The trigger position for all segments is set to pre-trigger (10% of the data is before the trigger condition and 90% of the data is after the trigger position). Because the last segment starts immediately with the trigger condition, the segment contains only post-trigger data. The three empty samples in the last segment is the space left over from the pre-trigger samples that the SignalTap II Embedded Logic Analyzer allocated to the buffer.

Figure 14–49. Segmented Capture with Preemption of Acquisition Segments *(Note 1)*



Note to Figure 14–49:

(1) A segmented acquisition buffer using the sequential trigger flow with a trigger condition set to “don’t care”. All segments, with the exception of the last segment, capture only one sample because the next trigger condition preempts the current buffer from filling to completion.

For the sequential trigger flow, the **Trigger Position** option applies to every segment in the buffer. For maximum flexibility on how the trigger position is defined, use the custom state-based trigger flow. By adjusting the trigger position that is specific to your debugging scenario, you can help maximize the use of the allocated buffer space.

Creating Mnemonics for Bit Patterns

The mnemonic table feature allows you to assign a meaningful name to a set of bit patterns, such as a bus. To create a mnemonic table, right-click in the **Setup** or **Data** tab of an **.stp** file and click Mnemonic Table Setup. You create a mnemonic table by entering sets of bit patterns and specifying a label to represent each pattern. Once you have created a mnemonic table, assign it to a group of signals. To assign a mnemonic table, right-click on the group, click **Bus Display Format** and select the desired mnemonic table.

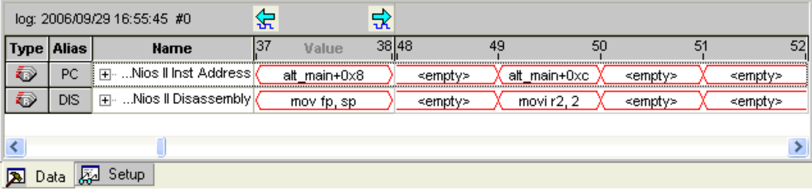
The labels you create in a table are used in different ways on the **Setup** and **Data** tabs. On the **Setup** tab, you can create basic triggers with meaningful names by right-clicking an entry in any **Trigger Conditions** column and selecting a label from the table you assigned to the signal group. On the **Data** tab, if any captured data matches a bit pattern contained in an assigned mnemonic table, the signal group data is replaced with the appropriate label, making it easy to see when expected data patterns occur.

Automatic Mnemonics with a Plug-In

When you use a plug-in to add signals to an **.stp** file, mnemonic tables for the added signals are automatically created and assigned to the signals defined in the plug-in. If you ever need to enable these mnemonic tables manually, right-click on the name of the signal or signal group. On the **Bus Display Format** submenu, click the name of the mnemonic table that matches the plug-in.

As an example, the Nios II plug-in makes it easy to monitor your design's signal activity as code is executed. If you have set up the logic analyzer to trigger on a function name in your Nios II code based on data from an **.elf** file, you can see the function name in the Instance Address signal group at the trigger sample, along with the corresponding disassembled code in the Disassembly signal group, as shown in Figure 14–50. Captured data samples around the trigger are referenced as offset addresses from the trigger function name.

Figure 14–50. Data Tab when the Nios II Plug-In is Used



Locating a Node in the Design

When you find the source of an error in your design using the SignalTap II Embedded Logic Analyzer, you can use the node locate feature to locate that signal in many of the tools found in the Quartus II software, as well as in your design files. This lets you find the source of the problem quickly so you can modify your design to correct the flaw. To locate a signal from the SignalTap II Embedded Logic Analyzer in one of the Quartus II software tools or your design files, right-click on the signal in the **.stp** file, and click **Locate in <*tool name*>**.

You can locate a signal from the node list in any of the following locations:

- Assignment Editor
- Pin Planner
- Timing Closure Floorplan
- Chip Planner
- Resource Property Editor
- Technology Map Viewer
- RTL Viewer
- Design File

For more information about using these tools, refer to each of the corresponding chapters in the *Quartus II Handbook*.

Saving Captured Data

The data log shows the history of captured data and the triggers used to capture the data. The analyzer acquires data, stores it in a log, and displays it as waveforms. When the logic analyzer is in auto-run mode and a trigger event occurs more than once, captured data for each time the trigger occurred is stored as a separate entry in the data log. This allows you to go back and review the captured data for each trigger event. The default name for a log is based on the time when the data was acquired. Altera recommends that you rename the data log with a more meaningful name.

The logs are organized in a hierarchical manner; similar logs of captured data are grouped together in trigger sets. If the **Data Log** pane is closed, on the View menu, select **Data Log** to reopen it. To enable data logging, turn on **Enable data log** in the **Data Log** (Figure 14–25). To recall a data log for a given trigger set and make it active, double-click the name of the data log in the list.

The Data Log feature is useful for organizing different sets of trigger conditions and different sets of signal configurations. For more information, refer to "Managing Multiple SignalTap II Files and Configurations" on page 14–32.

Converting Captured Data to Other File Formats

You can export captured data in the following file formats, some of which can be used with other EDA simulation tools:

- Comma Separated Values File (**.csv**)
- Table File (**.tbl**)
- Value Change Dump File (**.vcd**)
- Vector Waveform File (**.vwf**)
- Graphics format files (**.jpg**, **.bmp**)

To export the SignalTap II Embedded Logic Analyzer's captured data, on the File menu, click **Export** and specify the **File Name**, **Export Format**, and **Clock Period**.

Creating a SignalTap II List File

Captured data can also be viewed in an **.stp** list file. An **.stp** list file is a text file that lists all the data captured by the logic analyzer for a trigger event. Each row of the list file corresponds to one captured sample in the buffer. Columns correspond to the value of each of the captured signals or signal groups for that sample. If a mnemonic table was created for the captured data, the numerical values in the list are replaced with a matching entry from the table. This is especially useful with the use of a plug-in that includes instruction code disassembly. You can immediately see the order in which the instruction code was executed during the same time period of the trigger event. To create an **.stp** list file, on the File menu, select **Create/Update** and click **Create SignalTap II List File**.

Other Features

The SignalTap II Embedded Logic Analyzer has other features that do not necessarily belong to a particular task in the task flow.

Using the SignalTap II MATLAB MEX Function to Capture Data

If you use MATLAB for DSP design, you can call the MATLAB MEX function `alt_signaltap_run`, built into the Quartus II software, to acquire data from the SignalTap II Embedded Logic Analyzer directly into a matrix in the MATLAB environment. If you use the MEX function repeatedly in a loop, you can perform as many acquisitions as you can when using SignalTap II in the Quartus II software environment in the same amount of time.

The SignalTap II MATLAB MEX function is available only in the Windows version of the Quartus II software. It is compatible with MATLAB Release 14 Original Release Version 7 and later.

To set up the Quartus II software and the MATLAB environment to perform SignalTap II acquisitions, perform the following steps:

1. In the Quartus II software, create an **.stp** file (refer to "Creating and Enabling a SignalTap II File" on page 14–7).
2. In the node list in the **Data** tab of the SignalTap II Editor, organize the signals and groups of signals into the order in which you want them to appear in the MATLAB matrix. Each column of the imported matrix represents a single SignalTap II acquisition sample, while each row represents a signal or group of signals in the order they are organized in the **Data** tab.

   Signal groups acquired from the SignalTap II Embedded Logic Analyzer and transferred into the MATLAB environment with the MEX function are limited to a width of 32 signals. If you want to use the MEX function with a bus or signal group that contains more than 32 signals, split the group into smaller groups that do not exceed the 32-signal limit.

3. Save the **.stp** file and compile your design. Program your device and run the SignalTap II Embedded Logic Analyzer to ensure your trigger conditions and signal acquisition are working correctly.

4. In the MATLAB environment, add the Quartus II binary directory to your path with the following command:

```
addpath <Quartus install directory>\win ↵
```

You can view the help file for the MEX function by entering the following command in MATLAB without any operators:

```
alt_signaltap_run ↵
```

Use the MEX function in the MATLAB environment to open the JTAG connection to the device and run the SignalTap II Embedded Logic Analyzer to acquire data. When you finish acquiring data, close the connection.

To open the JTAG connection and begin acquiring captured data directly into a MATLAB matrix called `stp`, use the following command:

```
stp = alt_signaltap_run \
('<stp filename>'[,('signed'|'unsigned')[,'<instance names>'[, \
'<signalset name>'[,'<trigger name>']]]]); ↵
```

When capturing data, <*stp filename*> is the name of the **.stp** file you want to use. This is required for using the MEX function. The other MEX function options are defined in Table 14–12.

Table 14–12. SignalTap II MATLAB MEX Function Options

Option	Usage	Description
`signed` `unsigned`	`'signed'` `'unsigned'`	The **signed** option turns signal group data into 32-bit two's-complement signed integers. The MSB of the group as defined in the SignalTap II **Data** tab is the sign bit. The **unsigned** option keeps the data as an unsigned integer. The default is **signed**.
<*instance name*>	`'auto_signaltap_0'`	Specify a SignalTap II instance if more than one instance is defined. The default is the first instance in the **.stp** file, `auto_signaltap_0`.
<*signal set name*> <*trigger name*>	`'my_signalset'` `'my_trigger'`	Specify the signal set and trigger from the SignalTap II data log if multiple configurations are present in the **.stp** file. The default is the active signal set and trigger in the file.

You can enable or disable verbose mode to see the status of the logic analyzer while it is acquiring data. To enable or disable verbose mode, use the following commands:

```
alt_signaltap_run('VERBOSE_ON'); ↵
alt_signaltap_run('VERBOSE_OFF'); ↵
```

When you finish acquiring data, close the JTAG connection. Use the following command to close the connection:

```
alt_signaltap_run('END_CONNECTION'); ↵
```

For more information about the use of MEX functions in MATLAB, refer to the MATLAB Help.

Using SignalTap II in a Lab Environment

You can install a stand-alone version of the SignalTap II Embedded Logic Analyzer. This version is particularly useful in a lab environment in which you do not have a workstation that meets the requirements for a complete Quartus II installation, or if you do not have a license for a full installation of the Quartus II software. The stand-alone version of the SignalTap II Embedded Logic Analyzer is included with the Quartus II stand-alone Programmer and is available from the Downloads page of the Altera website (www.altera.com).

Remote Debugging Using the SignalTap II Embedded Logic Analyzer

You can use the SignalTap II Embedded Logic Analyzer to debug a design that is running on a device attached to a PC in a remote location.

To perform a remote debugging session, you must have the following setup:

- The Quartus II software installed on the local PC
- Stand-alone SignalTap II Embedded Logic Analyzer or the full version of the Quartus II software installed on the remote PC
- Programming hardware connected to the device on the PCB at the remote location
- TCP/IP protocol connection

Equipment Setup

On the PC in the remote location, install the stand-alone version of the SignalTap II Embedded Logic Analyzer or the full version of the Quartus II software. This remote computer must have Altera programming hardware connected, such as the EthernetBlaster or USB-Blaster.

On the local PC, install the full version of the Quartus II software. This local PC must be connected to the remote PC across a LAN with the TCP/IP protocol.

Software Setup on the Remote PC

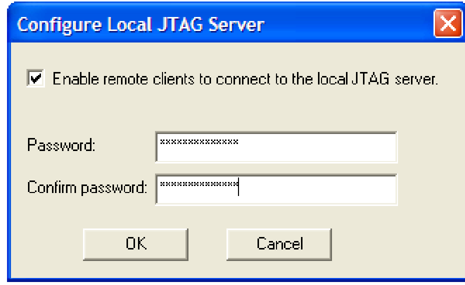
To set up the software on the remote PC, perform the following steps:

1. In the Quartus II programmer, click **Hardware Setup**.
2. Click the **JTAG Settings** tab (Figure 14–51).

Figure 14–51. Configure JTAG on Remote PC

3. Click **Configure local JTAG Server**.
4. In the **Configure Local JTAG Server** dialog box (Figure 14–52), turn on **Enable remote clients to connect to the local JTAG server** and in the password box, type your password. In the **Confirm Password** box, type your password again and click **OK**.

Figure 14–52. Configure Local JTAG Server on Remote

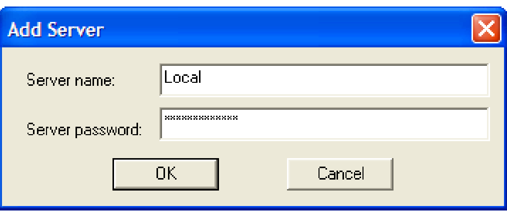


Software Setup on the Local PC

To set up your software on your local PC, perform the following steps:

1. Launch the Quartus II programmer.
2. Click **Hardware Setup**.
3. On the **JTAG** settings tab, click **Add server**.
4. In the **Add Server** dialog box (Figure 14–53), type the network name or IP address of the server you want to use and the password for the JTAG server that you created on the remote PC.

Figure 14–53. Add Server Dialog Box



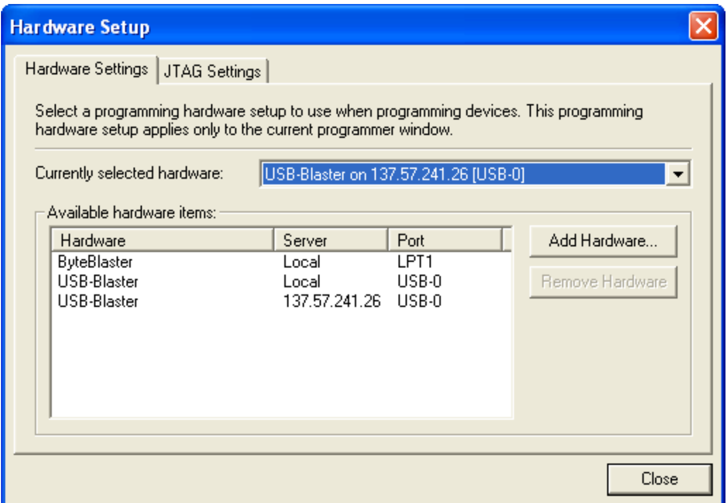
5. Click **OK**.

SignalTap II Setup on the Local PC

To connect to the hardware on the remote PC, perform the following steps:

1. Click the **Hardware Settings tab** and select the hardware on the remote PC (Figure 14–54).

Figure 14–54. Selecting Hardware on Remote PC



2. Click **Close**.

You can now control the logic analyzer on the device attached to the remote PC as if it was connected directly to the local PC.

Using the SignalTap II Embedded Logic Analyzer in Devices with Configuration Bitstream Security

Certain device families support bitstream decryption during configuration using an on-device AES decryption engine. You can still use the SignalTap II Embedded Logic Analyzer to analyze functional data within the FPGA. However, note that JTAG configuration is not possible after the security key has been programmed into the device.

Altera recommends that you use an unencrypted bitstream during the prototype and debugging phases of the design. Using an unencrypted bitstream allows you to generate new programming files and reconfigure the device over the JTAG connection during the debugging cycle.

If you must use the SignalTap II Embedded Logic Analyzer with an encrypted bitstream, first configure the device with an encrypted configuration file using Passive Serial (PS), Fast Passive Parallel (FPP), or Active Serial (AS) configuration modes. The design must contain at least one instance of the SignalTap II Embedded Logic Analyzer. After the FPGA is configured with a SignalTap II Embedded Logic Analyzer instance in the design, when you open the SignalTap II Embedded Logic Analyzer window/GUI in the Quartus II software, you then scan the chain and it will be ready to acquire data over JTAG.

Backward Compatibility with Previous Versions of Quartus II Software

You can open an old STP file in a current version of the Quartus II software. However, opening an STP file modifies it so that it cannot be opened in a previous version of the Quartus II software.

If you have a Quartus project file from a previous version of the software, you may have to update the STP configuration file if you wish to recompile the project. You can update the configuration file by invoking the SignalTap II GUI. If any updates to your configuration are necessary, a prompt will appear asking if you would like to update the **.stp** file to match the current version of the Quartus II software.

SignalTap II Scripting Support

You can run procedures and make settings described in this chapter in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF format.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

SignalTap II Command-Line Options

To compile your design with the SignalTap II Embedded Logic Analyzer using the command prompt, use the `quartus_stp` command. Table 14–13 shows the options that help you better understand how to use the `quartus_stp` executable.

Table 14–13. SignalTap II Command-Line Options

Option	Usage	Description
`stp_file`	`quartus_stp --stp_file <stp_filename>`	Assigns the specified **.stp** file to the `USE_SIGNALTAP_FILE` in the **.qsf** file.
`enable`	`quartus_stp --enable`	Creates assignments to the specified **.stp** file in the **.qsf** file and changes `ENABLE_SIGNALTAP` to `ON`. The SignalTap II Embedded Logic Analyzer is included in your design the next time the project is compiled. If no **.stp** file is specified in the **.qsf** file, the `--stp_file` option must be used. If the `--enable` option is omitted, the current value of `ENABLE_SIGNALTAP` in the **.qsf** file is used.
`disable`	`quartus_stp --disable`	Removes the **.stp** file reference from the **.qsf** file and changes `ENABLE_SIGNALTAP` to `OFF`. The SignalTap II Embedded Logic Analyzer is removed from the design database the next time you compile your design. If the `--disable` option is omitted, the current value of `ENABLE_SIGNALTAP` in the **.qsf** file is used.
`create_signaltap_hdl_file`	`quartus_stp --create_signaltap_hdl_file`	Creates an **.stp** file representing the SignalTap II instance in the design generated by the SignalTap II Embedded Logic Analyzer megafunction created with the MegaWizard Plug-In Manager. The file is based on the last compilation. You must use the `--stp_file` option to create an **.stp** file properly. Analogous to the **Create SignalTap II File from Design Instance(s)** command in the Quartus II software.

Example 14–6 illustrates how to compile a design with the SignalTap II Embedded Logic Analyzer at the command line.

Example 14–6.

```
quartus_stp filtref --stp_file stp1.stp --enable ↵
quartus_map filtref --source=filtref.bdf --family=CYCLONE ↵
quartus_fit filtref --part=EP1C12Q240C6 --fmax=80MHz --tsu=8ns ↵
quartus_tan filtref ↵
quartus_asm filtref ↵
```

The `quartus_stp --stp_file stp1.stp --enable` command creates the QSF variable and instructs the Quartus II software to compile the **stp1.stp** file with your design. The `--enable` option must be applied for the SignalTap II Embedded Logic Analyzer to compile properly into your design.

Example 14–7 shows how to create a new **.stp** file after building the SignalTap II Embedded Logic Analyzer instance with the MegaWizard Plug-In Manager.

Example 14–7.

```
quartus_stp filtref --create_signaltap_hdl_file --stp_file stp1.stp ↵
```

For information about the other command line executables and options, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

SignalTap II Tcl Commands

The **quartus_stp** executable supports a Tcl interface that allows you to capture data without running the Quartus II GUI. You cannot execute SignalTap II Tcl commands from within the Tcl console in the GUI. They must be run from the command line with the **quartus_stp** executable. To run a Tcl file that has SignalTap II Tcl commands, use the following command:

```
quartus_stp -t <Tcl file> ↵
```

Table 14–14 shows the Tcl commands that you can use with SignalTap II Embedded Logic Analyzer.

Table 14–14. SignalTap II Tcl Commands

Command	Argument	Description
`open_session`	`-name <stp_filename>`	Opens the specified **.stp** file. All captured data is stored in this file.
`run`	`-instance <instance_name>` `-signal_set <signal_set>` *(optional)* `-trigger <trigger_name>` *(optional)* `-data_log <data_log_name>` *(optional)* `-timeout <seconds>` *(optional)*	Starts the analyzer. This command must be followed by all the required arguments to properly start the analyzer. You can optionally specify the name of the data log you want to create. If the Trigger condition is not met, you can specify a timeout value to stop the analyzer.
`run_multiple_start`	None	Defines the start of a set of `run` commands. Use this command when multiple instances of data acquisition are started simultaneously. Add this command before the set of run commands that specify data acquisition. You must use this command with the `run_multiple_end` command. If the `run_multiple_end` command is not included, the `run` commands do not execute.
`run_multiple_end`	None	Defines the end of a set of `run` commands. Use this command when multiple instances of data acquisition are started simultaneously. Add this command after the set of `run_commands`.
`stop`	None	Stops data acquisition.
`close_session`	None	Closes the currently open **.stp** file. You cannot run the analyzer after the **.stp** file is closed.

For more information about SignalTap II Tcl commands, refer to the Quartus II Help.

Example 14–8 is an excerpt from a script that is used to continuously capture data. Once the trigger condition is met, the data is captured and stored in the data log.

Example 14–8.

```
#opens signaltap session
open_session -name stp1.stp
#start acquisition of instance auto_signaltap_0 and
#auto_signaltap_1 at the same time
#calling run_multiple_end will start all instances
#run after run_multiple_start call
run_multiple_start
run -instance auto_signaltap_0 -signal_set signal_set_1 -trigger /
trigger_1 -data_log log_1 -timeout 5
run -instance auto_signaltap_1 -signal_set signal_set_1 -trigger /
trigger_1 -data_log log_1 -timeout 5
run_multiple_end
#close signaltap session
close_session
```

When the script is completed, open the **.stp** file that you used to capture data to examine the contents of the Data Log.

Design Example: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems

The system in this example contains many components, including a Nios processor, a direct memory access (DMA) controller, on-chip memory, and an interface to external SDRAM memory. In this example, the Nios processor executes a simple C program from on-chip memory and waits for a button push. After a button is pushed, the processor initiates a DMA transfer, which you analyze using the SignalTap II Embedded Logic Analyzer.

For more information about this example and using the SignalTap II Embedded Logic Analyzer with SOPC builder systems, refer to *AN 323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder Systems* and *AN 446: Debugging Nios II Systems with the SignalTap II Logic Analyzer*.

Custom Triggering Flow Application Examples

The custom triggering flow in the SignalTap II Embedded Logic Analyzer is most useful for organizing a number of triggering conditions and for precise control over the acquisition buffer. This section provides two application examples for defining a custom triggering flow within the SignalTap II Embedded Logic Analyzer. Both examples can be easily copied and pasted directly into the state machine description box by using the state display mode **All states in one window**.

For additional triggering flow design examples, refer to the Quartus II On-Chip Debugging Design Examples page for on-chip debugging.

Design Example 1: Specifying a Custom Trigger Position

Actions to the acquisition buffer can accept an optional post-count argument. This post-count argument enables you to define a custom triggering position for each segment in the acquisition buffer. Example 14–9 shows an example that applies a trigger position to all segments in the acquisition buffer. The example describes a triggering flow for an acquisition buffer split into four segments. If each acquisition segment is 64 samples in depth, the trigger position for each buffer will be at sample #34. The acquisition stops after all four segments are filled once.

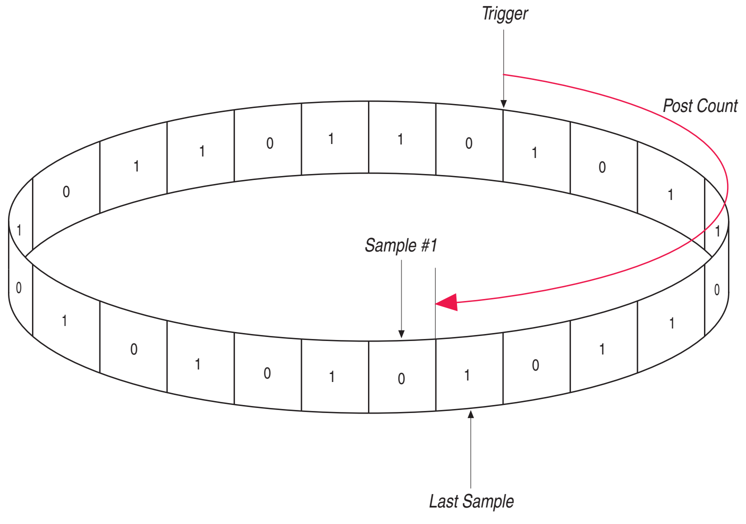
Example 14–9.

```
if (c1 == 3 && condition1)
   trigger 30;
else if ( condition1 )
begin
   segment_trigger 30;
   increment c1;

end
```

Each segment acts as a non-segmented buffer that continuously updates the memory contents with the signal values. The last acquisition before stopping the buffer is displayed on the **Data** tab as the last sample number in the affected segment. The trigger position in the affected segment is then defined by *N – post count fill*, where *N* is the number of samples per segment. Figure 14–55 illustrates the triggering position.

Figure 14–55. Specifying a Custom Trigger Position



Design Example 2: Trigger When triggercond1 Occurs Ten Times between triggercond2 and triggercond3

The custom trigger flow description is often useful to count a sequence of events before triggering the acquisition buffer. Example 14–10 shows such a sample flow. This example uses three basic triggering conditions configured in the SignalTap II **Setup** tab.

This example triggers the acquisition buffer when `condition1` occurs after `condition3` and occurs ten times prior to `condition3`. If `condition3` occurs prior to ten repetitions of `condition1`, the state machine transitions to a permanent wait state.

Example 14–10.

```
state ST1:

if ( condition2  )
begin
   reset c1;
   goto ST2;
end


State ST2 :
if ( condition1 )
    increment c1;

else if (condition3 && c1 < 10)
    goto ST3;

else if ( condition3 && c1 >= 10)
    trigger;

ST3:
goto ST3;
```

Conclusion

As the FPGA industry continues to make technological advancements, outdated methodologies need to be replaced with new technologies that maximize productivity. The SignalTap II Embedded Logic Analyzer gives you the same benefits as a traditional logic analyzer, without the many shortcomings of a piece of dedicated test equipment. This version of the SignalTap II Embedded Logic Analyzer provides many new and innovative features that allow you to capture and analyze internal signals in your FPGA, allowing you to find the source of a design flaw in the shortest amount of time.

Referenced Documents

This chapter references the following documents:

- *AN 323: Using SignalTap II Embedded Logic Analyzers in SOPC Builder System*
- *AN 446: Debugging Nios II Systems with the SignalTap II Embedded Logic Analyzer*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Design Debugging Using In-System Sources and Probes* chapter in volume 3 of the *Quartus II Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*
- *In-System Debugging Using External Logic Analyzers* chapter in volume 3 of the *Quartus II Handbook*

- *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Reference Manual*
- *Quick Design Debugging Using SignalProbe* chapter in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 14–15 shows the revision history for this chapter.

Table 14–15. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Updated Table 14–1 ■ Updated "Using Incremental Compilation with the SignalTap II Embedded Logic Analyzer" on page 14–55 ■ Added new Figure 14–42 ■ Made minor editorial updates	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	Updated for the Quartus II software version 8.1 release: ■ Added new section "Using the Storage Qualifier Feature" on page 14–25 ■ Added description of `start_store` and `stop_store` commands in section "Trigger Condition Flow Control" on page 14–36 ■ Added new section "Runtime Reconfigurable Options" on page 14–63	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	Updated for the Quartus II software version 8.0: ■ Added "Debugging Finite State machines" on page 14-24 ■ Documented various GUI usability enhancements, including improvements to the resource estimator, the bus find feature, and the dynamic display updates to the counter and flag resources in the State-based trigger flow control tab ■ Added "Capturing Data Using Segmented Buffers" on page 14–16 ■ Added hyperlinks to referenced documents throughout the chapter ■ Minor editorial updates	Updated for the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

 © March 2009 Altera Corporation

15. In-System Debugging Using External Logic Analyzers

QII53016-9.0.0

Introduction

The Quartus II Logic Analyzer Interface (LAI) allows you to examine the behavior of internal signals using an external logic analyzer and using a minimal number of FPGA I/O pins, while your design is running at full speed on your FPGA.

This chapter's use of the term "logic analyzer" includes both logic analyzers and oscilloscopes equipped with digital channels, commonly referred to as mixed signal analyzers or MSOs.

The LAI connects a large set of internal device signals to a small number of output pins. You can connect these output pins to an external logic analyzer for debugging purposes. In the Quartus II LAI, the internal signals are grouped together, distributed to a user-configurable multiplexer, and then output to available I/O pins on your FPGA. Instead of having a one-to-one relationship between internal signals to output pins, the Quartus II LAI enables you map many internal signals to a smaller number of output pins. The exact number of internal signals that you can map to an output pin varies based on the multiplexer settings in the Quartus II LAI.

This chapter details the following topics:

- "Choosing a Logic Analyzer"
- "Debugging Your Design Using the Logic Analyzer Interface" on page 15–3
- "Advanced Features" on page 15–11

Choosing a Logic Analyzer

The Quartus II software offers the following two general purpose on-chip debugging tools for debugging a large set of RTL signals from your design:

- The embedded SignalTap® II logic analyzer
- An external logic analyzer, which connects to internal signals in your FPGA, by using the Quartus II LAI

Table 15–1 describes the advantages to each debugging technologies.

Table 15–1. Comparing the SignalTap II Embedded Logic Analyzer with the Logic Analyzer Interface

Feature and Description	Logic Analyzer Interface	SignalTap II Embedded Logic Analyzer
Sample Depth You have access to a wider sample depth with an external logic analyzer. In the SignalTap II Embedded Logic Analyzer, the maximum sample depth is set to 128 Kb, which is a device constraint. However, with an external logic analyzer, there are no device constraints, providing you a wider sample depth.	✓	—
Debugging Timing Issues Using an external logic analyzer provides you with access to a “timing” mode, which enables you to debug combined streams of data.	✓	—
Performance You frequently have limited routing resources available to place-and-route when you use SignalTap II with your design. An external logic analyzer adds minimal logic, which removes resource limits on place-and-route.	✓	—
Triggering Capability SignalTap II offers triggering capabilities that is comparable with external logic analyzers.	✓	✓
Use of Output Pins Using the SignalTap II Logic Analyzer, no additional output pins are required. Using an external logic analyzer requires the use of additional output pins.	—	✓
Acquisition Speed With the SignalTap II Logic Analyzer, you can acquire data at speeds of over 200 MHz. You can achieve the same acquisition speeds with an external logic analyzer; however, you must consider signal integrity issues.	—	✓

The Quartus II software offers a portfolio of on-chip debugging tools. For an overview and comparison of all tools available in the Quartus II software on-chip debugging tool suite, refer to *Section V. In-System Debugging* in volume 3 of the *Quartus II Handbook*.

Required Components

You must have the following components to perform analysis using the Quartus II LAI:

- The Quartus II software starting with version 5.1 and later
- The device under test
- An external logic analyzer
- An Altera® communications cable
- A cable to connect the FPGA to the external logic analyzer

Figure 15–1 shows the LAI and the hardware setup.

Figure 15–1. Logic Analyzer Interface and Hardware Setup

Notes to Figure 15–1:

(1) Configuration and control of the LAI using a computer loaded with the Quartus II software via the JTAG port.
(2) Configuration and control of the LAI using a third-party vendor logic analyzer via the JTAG port. Support varies by vendor.

FPGA Device Support

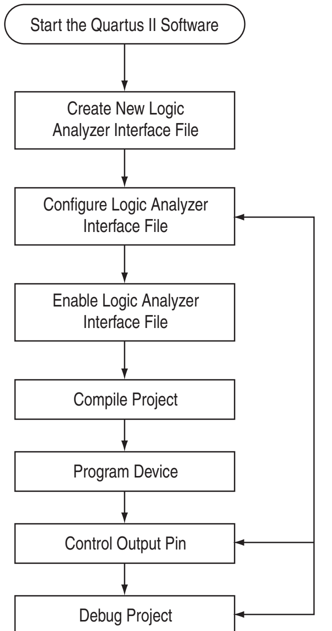
You can use the Quartus II Logic Analyzer Interface (LAI) with the following FPGA device families:

- Arria® GX
- Stratix® series
- Cyclone® series
- MAX® II
- APEX™ 20K
- APEX II

Debugging Your Design Using the Logic Analyzer Interface

Figure 15–2 shows the steps you must follow to debug your design with the Quartus II LAI.

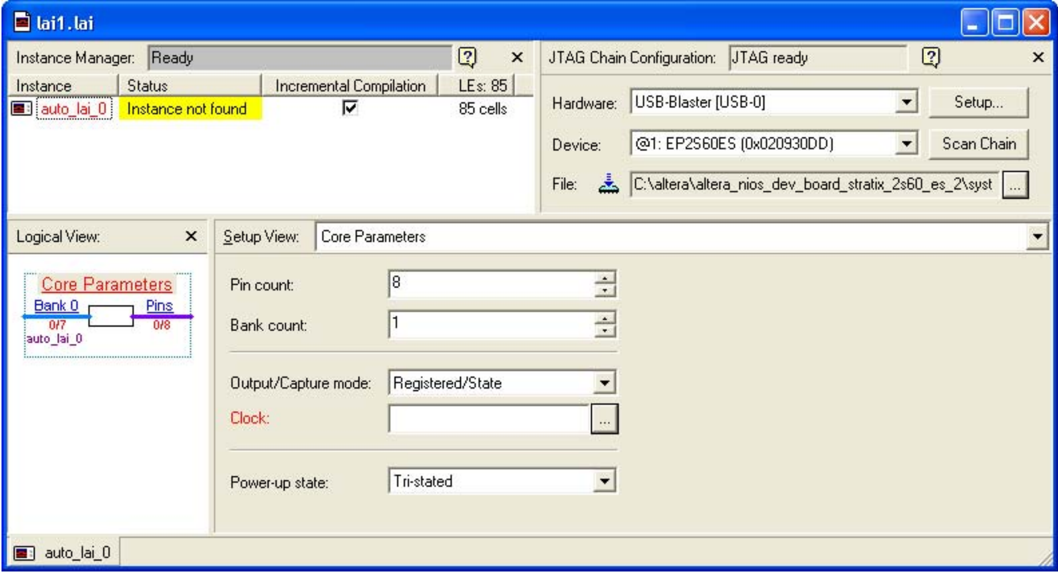
Figure 15–2. LAI Process Flow



Creating an LAI File

The Logic Analyzer Interface (**.lai**) file defines the interface that builds a connection between internal FPGA signals and your external logic analyzer. Figure 15–3 shows an example of an **.lai** editor.

Figure 15–3. The LAI Editor



To define the Quartus II LAI, you can create a new **.lai** file or use an existing **.lai** file.

Creating a New Logic Analyzer Interface File

To create a new **.lai** file, perform the following steps:

1. In the Quartus II software, on the File menu, click **New**. The **New** dialog box opens.
2. Click the **Other Files** tab.
3. Select **Logic Analyzer Interface File**.
4. Click **OK**. The LAI editor opens. The file name is assigned by the Quartus II software (refer to Figure 15–3 on page 15–4). When you save the file, you will be prompted for a file name. Refer to "Saving the External Analyzer Interface File" on page 15–5.

Opening an Existing External Analyzer Interface File

To open an existing **.lai** file, on the Tools menu, click **Logic Analyzer Interface Editor**. If no **.lai** file is enabled for the current project, the editor automatically creates a new **.lai** file. If an **.lai** file is currently enabled for the project, that file opens when you select the **Logic Analyzer Interface Editor**.

Alternatively, on the File menu, click **Open**, and select the **.lai** file you want to open.

Saving the External Analyzer Interface File

To save your **.lai** file, perform the following steps:

1. In the Quartus II software, on the File menu, click **Save As**. The **Save As** dialog box opens.
2. In the **File name** box, enter the desired file name.
3. Click **Save**.

Configuring the Logic Analyzer Interface File Core Parameters

After you have created the **.lai** file, you must configure the **.lai** file core parameters.

To configure the **.lai** file core parameters, from the **Setup View** list, select **Core Parameters**. Refer to Figure 15–4.

Figure 15–4. Logic Analyzer Interface File Core Parameters

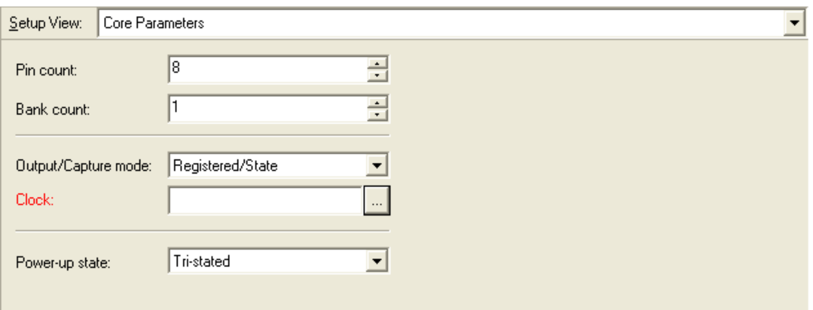


Table 15–2 lists the **.lai** file core parameters.

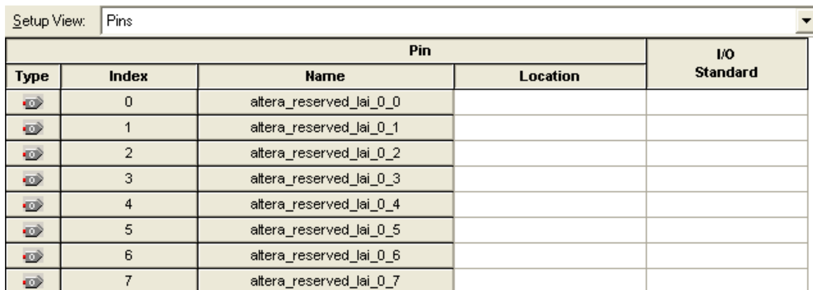
Table 15–2. Logic Analyzer Interface File Core Parameters

Parameter	Description
Pin Count	The Pin Count parameter signifies the number of pins you want dedicated to your LAI. The pins must be connected to a debug header on your board. Within the FPGA, each pin is mapped to a user-configurable number of internal signals. The Pin Count parameter can range from 1 to 255 pins.
Bank Count	The Bank Count parameter signifies the number of internal signals that you want to map to each pin. For example, a Bank Count of 8 implies that you will connect eight internal signals to each pin. The Bank Count parameter can range from 1 to 255 banks.
Output/Capture Mode	The Output/Capture Mode parameter signifies the type of acquisition you perform. There are two options that you can select: **Combinational/Timing**—This acquisition uses your external logic analyzer's internal clock to determine when to sample data. Because Combinational/Timing acquisition samples data asynchronously to your FPGA, you must determine the sample frequency you should use to debug and verify your system. This mode is effective if you want to measure timing information, such as channel-to-channel skew. For more information about the sampling frequency and the speeds at which it can run, refer to the data sheet for your external logic analyzer. **Registered/State**—This acquisition uses a signal from your system under test to determine when to sample. Because Registered/State acquisition samples data synchronously with your FPGA, it provides you with a functional view of your FPGA while it is running. This mode is effective when you verify the functionality of your design.
Clock	The clock parameter is available only when Output/Capture Mode is set to Registered State. You must specify the sample clock in the Core Parameters view. The sample clock can be any signal in your design. However, for best results, Altera recommends that you use a clock with an operating frequency fast enough to sample the data you would like to acquire.
Power-Up State	The Power-Up State parameter specifies the power-up state of the pins you have designated for use with the LAI. You have the option of selecting tri-stated for all pins, or selecting a particular bank that you have enabled.

Mapping the Logic Analyzer Interface File Pins to Available I/O Pins

To configure the **.lai** file I/O pins parameters, select Pins from the Setup View list (Figure 15–5).

Figure 15–5. Logic Analyzer Interface File Pins Parameters



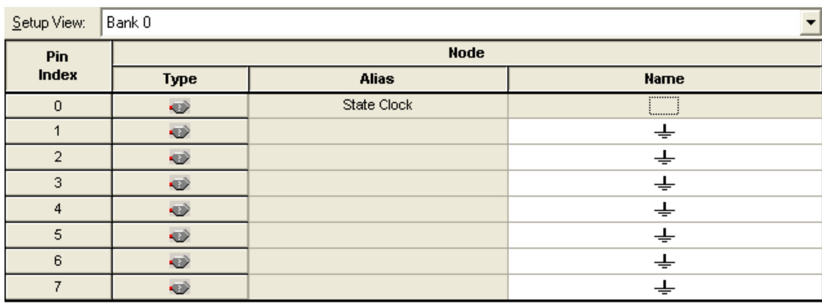
To assign pin locations for the LAI, double-click the **Location** column next to the reserved pins in the **Name** column. This opens the Pin Planner.

For information about how to use the Pin Planner, refer to the *Pin Planner* section in the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Mapping Internal Signals to the Logic Analyzer Interface Banks

After you have specified the number of banks to use in the **Core Parameters** settings page, you must assign internal signals for each bank in the LAI. Click the **Setup View** arrow and select **Bank n** or **All Banks** (Figure 15–6).

Figure 15–6. Logic Analyzer Interface Bank Parameters

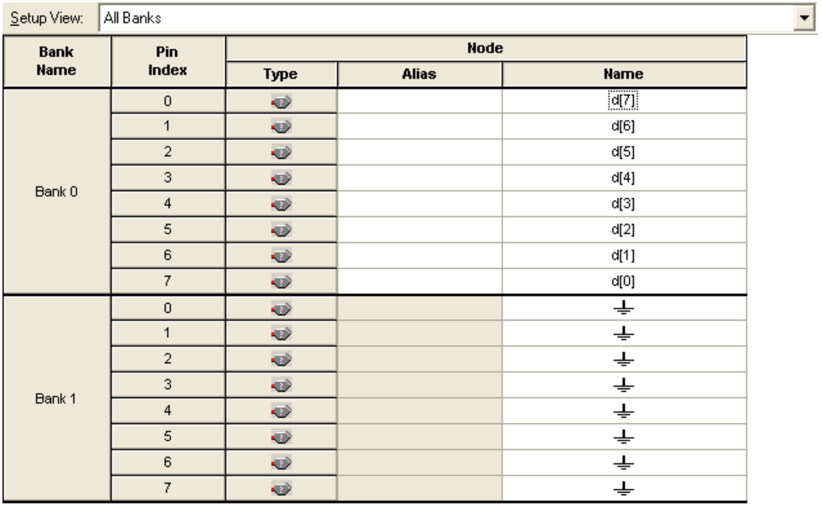


Setup View: Bank 0

Pin Index	Node		
	Type	Alias	Name
0		State Clock	
1			
2			
3			
4			
5			
6			
7			

To view all of your bank connections, click **Setup View** and select **All Banks** (Figure 15–7).

Figure 15–7. Logic Analyzer Interface All Bank Parameters



Setup View: All Banks

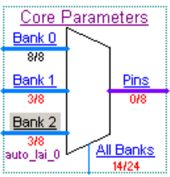
Bank Name	Pin Index	Node		
		Type	Alias	Name
Bank 0	0			d[7]
	1			d[6]
	2			d[5]
	3			d[4]
	4			d[3]
	5			d[2]
	6			d[1]
	7			d[0]
Bank 1	0			
	1			
	2			
	3			
	4			
	5			
	6			
	7			

Using the Node Finder

Before making bank assignments, on the View menu, point to **Utility Windows** and click Node Finder. Find the signals that you want to acquire, then drag and drop the signals from the **Node Finder** dialog box into the bank **Setup View**. When adding signals, use **SignalTap II: pre-synthesis** for non-incrementally routed instances and **SignalTap II: post-fitting** for incrementally routed instances.

As you continue to make assignments in the bank **Setup View**, the schematic of your LAI in the **Logical View** of your **.lai** file begins to reflect your assignments (Figure 15–8).

Figure 15–8. A Logical View of the Logic Analyzer Interface Schematic



Continue making assignments for each bank in the **Setup View** until you have added all of the internal signals for which you wish to acquire data.

You can right-click to switch between the LAI schematic and the LAI **Setup** view.

Enabling the Logic Analyzer Interface Before Compiling Your Quartus II Project

Compile your project after you have completed the following steps:

- Configure your LAI parameters
- Map the LAI pins to available I/O pins
- Map the internal signals to the LAI banks

Compiling Your Quartus II Project

Before compiling your project, you must enable the LAI.

1. On the Assignments menu, click **Settings**. The **Settings** dialog box opens.
2. Under Category, click **Logic Analyzer Interface**. The **Logic Analyzer Interface** displays.
3. Turn on **Enable Logic Analyzer Interface**.
4. Click **Logic Analyzer Interface file name** and specify the full path name to your **.lai** file.

After you have specified the name of your **.lai** file, you must compile your project. To compile your project, on the Processing menu, click **Start Compilation**.

To ensure that the LAI is properly compiled with your project, expand the entity hierarchy in the Project Navigator. (To display the Project Navigator, on the View menu, point to **Utility Windows** and click **Project Navigator**.) If the LAI is compiled with your design, the `sld_hub` and `sld_multitap` entities are shown in the project navigator (Figure 15–9).

Figure 15–9. Project Navigator

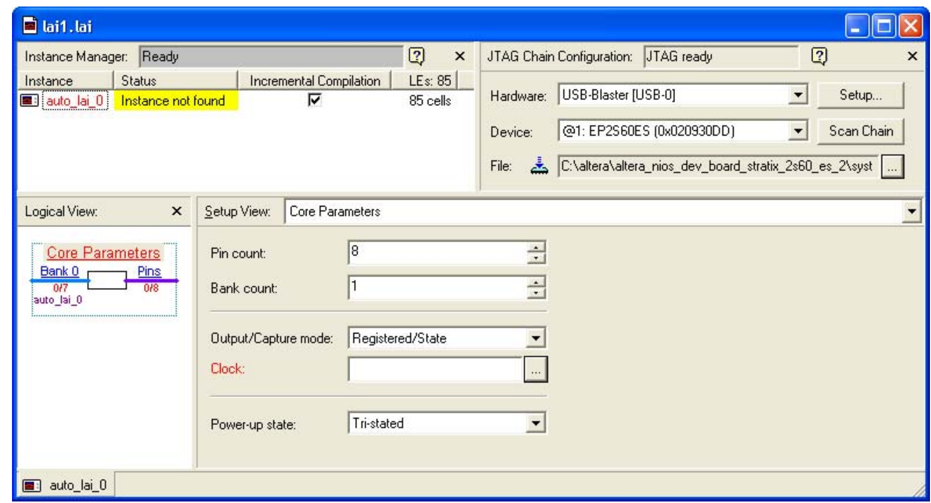
Entity	Logic Cells	LC Registers
Stratix: EP1S10B672C7		
test	136 (1)	81
sld_multitap:auto_lai_0	35 (11)	15
sld_hub:sld_hub_inst	100 (25)	65

Programming Your FPGA Using the Logic Analyzer Interface

After compilation completes, you must configure your FPGA before using the LAI. To configure a device for use with the LAI, perform the following steps:

1. Open the **.lai** file Editor (Figure 15–10).
2. Under **JTAG Chain Configuration**, click **Hardware** and select your hardware communications device. You may have to click **Settings** to configure your hardware.
3. Click **Device** and select the FPGA device to which you want to download the design (it may be automatically detected). You may have to click **Scan Chain** to configure your device.
4. Click **File** and select the SRAM Object File (**.sof**) that includes the **.lai** file (it may be automatically detected).
5. If desired, turn on **Incremental Compilation**.
6. Save the **.lai** file.
7. Click the **Program Device** icon to program the device.

Figure 15–10. The JTAG Section of the Logic Analyzer Interface File



Using the Logic Analyzer Interface with Multiple Devices

You can use the LAI with multiple devices in your JTAG chain. Your JTAG chain can also consist of devices that do not support the LAI or non-Altera, JTAG-compliant devices. To use the LAI in more than one FPGA, create an LAI and configure an **.lai** file for each FPGA that you want to analyze. To perform multi-FPGA analysis, perform the following steps:

1. Open the Quartus II software.
2. Create, configure, and compile an **.lai** file for each design.
3. Open one **.lai** file at a time.

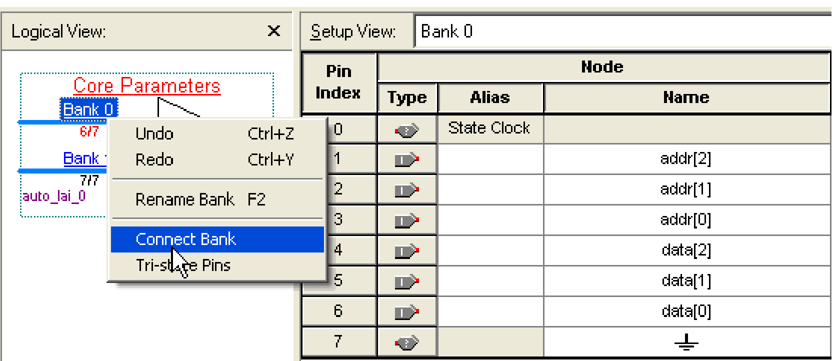
   You do not have to open a Quartus II project to open an **.lai** file.

4. Follow Steps 2 through 6 under "Programming Your FPGA Using the Logic Analyzer Interface" on page 15–9.
5. Click the **Program Device** icon to program the device.
6. Control each **.lai** file independently.

Configuring Banks in the Logic Analyzer Interface File

When you have programmed your FPGA, you can control which bank is mapped to the reserved **.lai** file output pins. To control which bank is mapped, in the schematic in the logical view, right-click the bank and click **Connect Bank** (Figure 15–11).

Figure 15–11. Configuring Banks



Acquiring Data on Your Logic Analyzer

To acquire data on your logic analyzer, you must establish a connection between your device and the external logic analyzer.

For more information about this process and for guidelines on how to establish connections between debugging headers and logic analyzers, refer to the documentation for your logic analyzer.

Advanced Features

This section describes the following advanced features:

- Using the Logic Analyzer Interface with Incremental Compilation
- Creating Multiple Logic Analyzer Interface Instances in One FPGA

Using the Logic Analyzer Interface with Incremental Compilation

Using the LAI with Incremental Compilation enables you to preserve the synthesis and fitting of your original design, and add the LAI to your design without recompiling your original source code.

To use the LAI with Incremental Compilation, perform the following steps:

1. Start the Quartus II software.
2. Enable Design Partitions. To enable Partitions, perform the following steps:
   a. On the Assignments menu, click **Design Partitions**.
   b. In the **Incremental Compilation** list, select **Full Incremental Compilation**.
   c. Create Design Partitions for the entities in your design, and set the Netlist Type to **Post-fit**.
   d. On the Processing menu, click **Start Compilation**.
3. Enable LAI Incremental Compilation by performing these steps:
   a. In your **.lai** file, under **Instance Manager**, click **Incremental Compilation**.

When you enable **Incremental Compilation**, all existing presynthesis signals are converted into post-fitting signals. Only post-fitting signals can be used with the LAI with Incremental Compilation.

   b. Add Post-Fitting nodes to your **.lai** file.
   c. On the Processing menu, click **Start Compilation**.

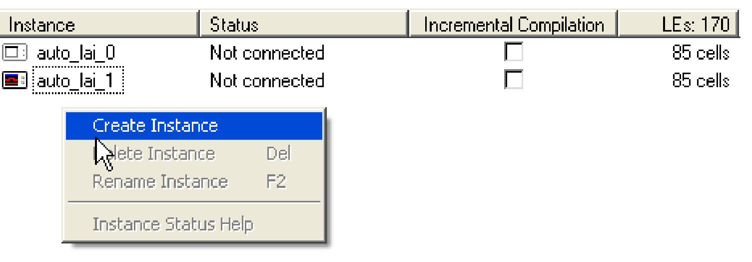
Creating Multiple Logic Analyzer Interface Instances in One FPGA

The LAI includes support for multiple interfaces in one FPGA. This feature is particularly useful when you want to build LAI configurations that contain different settings. For example, you can build one LAI instance to perform Registered/State analysis and build another instance that performs Combinational/Timing analysis on the same set of signals.

Another example would be performing Registered/State analysis on portions of your design that are in different clock domains.

To create multiple LAIs, on the Edit menu, click **Create Instance**. Alternatively, you can right-click the **Instance Manager** window, and click **Create Instance** (Figure 15–12).

Figure 15–12. Creating Multiple Logic Analyzer Interface Instances in One FPGA



Conclusion

As the FPGA industry continues to make technological advancements, outdated debugging methodologies must be replaced with new technologies that maximize productivity. The LAI feature enables you to connect many internal signals within your FPGA to an external logic analyzer with the use of a small number of I/O pins. This new technology in the Quartus II software enables you to use feature-rich external logic analyzers to debug your FPGA design, ultimately enabling you to delver your product in the shortest amount of time.

Referenced Documents

This chapter references the following documents:

- *Section V. In-System Debugging* in volume 3 of the *Quartus II Handbook*
- *I/O Management* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 15–3 shows the revision history for this chapter.

Table 15–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Minor editorial updates. Removed Figures 15–4, 15–5, and 15–11 from 8.1 version.	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Updated device support list on page 15–3 ■ Added links to referenced documents throughout the chapter ■ Added “Referenced Documents” ■ Added reference to *Section V. In-System Debugging* ■ Minor editorial updates	Updated for the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

16. In-System Updating of Memory and Constants

QII53012-9.0.0

Introduction

The In-System Memory Content Editor allows you to view and update memories and constants using the JTAG port connection. This chapter explains how to use the Quartus II In-System Memory Content Editor as part of your FPGA design and verification flow.

FPGA designs are growing larger in density and are becoming more complex. Designers and verification engineers require more access to the design that is programmed in the device to quickly identify, test, and resolve issues. The in-system updating of memory and constants capability of the Quartus® II software provides read and write access to in-system FPGA memories and constants through the Joint Test Action Group (JTAG) interface, making it easier to test changes to memory contents in the FPGA while the FPGA is functioning in the end system.

This chapter contains the following sections:

- "Device Megafunction Support" on page 16–2
- "Using In-System Updating of Memory and Constants with Your Design" on page 16–2
- "Creating In-System Modifiable Memories and Constants" on page 16–3
- "Running the In-System Memory Content Editor" on page 16–3

Overview

The ability to read and update memories and constants in a programmed device provides more insight into and control over your design. The Quartus II In-System Memory Content Editor gives you access to device memories and constants. When used in conjunction with the SignalTap® II Embedded Logic Analyzer, this feature provides the visibility required to debug your design in the hardware lab.

For more information about the SignalTap II Embedded Logic Analyzer, refer to the *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

The ability to read data from memories and constants allows you to quickly identify the source of problems. In addition, the write capabilities allow you to bypass functional issues by writing expected data. For example, if a parity bit in your memory is incorrect, you can use the In-System Content Editor to write the correct parity bit values into your RAM, allowing your system to continue functioning. You can also intentionally write incorrect parity bit values into your RAM to check your design's error handling functionality.

The Quartus II software offers a portfolio of on-chip debugging tools. For an overview and comparison of all tools available in the Quartus II software on-chip debugging tool suite, refer to *Section V. In-System Debugging* in volume 3 of the *Quartus II Handbook*.

Device Megafunction Support

The following tables list the devices and types of memories and constants that are currently supported by the Quartus II software. Table 16–1 lists the types of memory supported by the MegaWizard™ Plug-In Manager and the In-System Memory Content Editor.

Table 16–1. MegaWizard Plug-In Manager Support

Installed Plug-Ins Category	Megafunction Name
Gates	LPM_CONSTANT
Memory Compiler	RAM: 1-PORT, ROM: 1-PORT
Storage	ALTSYNCRAM, LPM_RAM_DQ, LPM_ROM

Table 16–2 lists support for in-system updating of memory and constants for the Stratix® series, Arria® GX, Cyclone® series, APEX™ II, and APEX 20K device families.

Table 16–2. Supported Megafunctions

Megafunction	Arria GX / Stratix Series			Cyclone Series	APEX II	APEX 20K
	M512 Blocks	M4K Blocks	MegaRAM Blocks			
LPM_CONSTANT	Read/ Write	Read/ Write	Read/ Write	Read/ Write	Read/ Write	Read/ Write
LPM_ROM	Write	Read/ Write	N/A	Read/ Write	Read/ Write	Write
LPM_RAM_DQ	N/A	Read/ Write	Read/ Write	Read/ Write	Read/ Write	N/A *(1)*
ALTSYNCRAM (ROM)	Write	Read/ Write	N/A	Read/ Write	N/A	N/A
ALTSYNCRAM (Single-Port RAM Mode)	N/A	Read/ Write	Read/ Write	Read/ Write	N/A	N/A

Note to Table 16–2:

(1) Only write-only mode is applicable for this single-port RAM. In read-only mode, use LPM_ROM instead of LPM_RAM_DQ.

Using In-System Updating of Memory and Constants with Your Design

Using the In-System Updating of Memory and Constants feature requires the following steps:

1. Identify the memories and constants that you want to access.
2. Edit the memories and constants to be run-time modifiable.
3. Perform a full compilation.
4. Program your device.
5. Launch the In-System Memory Content Editor.

Creating In-System Modifiable Memories and Constants

When you specify that a memory or constant is run-time modifiable, the Quartus II software changes the default implementation. A single-port RAM is converted to dual-port RAM, and a constant is implemented in registers instead of look-up tables (LUTs). These changes enable run-time modification without changing the functionality of your design. For a list of run-time modifiable megafunctions, refer to Table 16–1.

To enable your memory or constant to be modifiable, perform the following steps:

1. On the Tools menu, click **MegaWizard Plug-In Manager**.
2. If you are creating a new megafunction, select **Create a new custom megafunction variation**. If you have an existing megafunction, select **Edit an existing custom megafunction variation**.
3. Make the necessary changes to the megafunction based on the characteristics required by your design, turn on **Allow In-System Memory Content Editor to capture and update content independently of the system clock**, and type a value in the **Instance ID** text box. These parameters can be found on the last page of the wizard for megafunctions that support in-system updating.

   The Instance ID is a four-character string used to distinguish the megafunction from other in-system memories and constants.

4. Click **Finish**.
5. On the Processing menu, click **Start Compilation**.

If you instantiate a memory or constant megafunction directly using ports and parameters in VHDL or Verilog HDL, add or modify the `lpm_hint` parameter as follows:

In VHDL code, add the following:

```
lpm_hint => "ENABLE_RUNTIME_MOD = YES,
  INSTANCE_NAME = <instantiation name>";
```

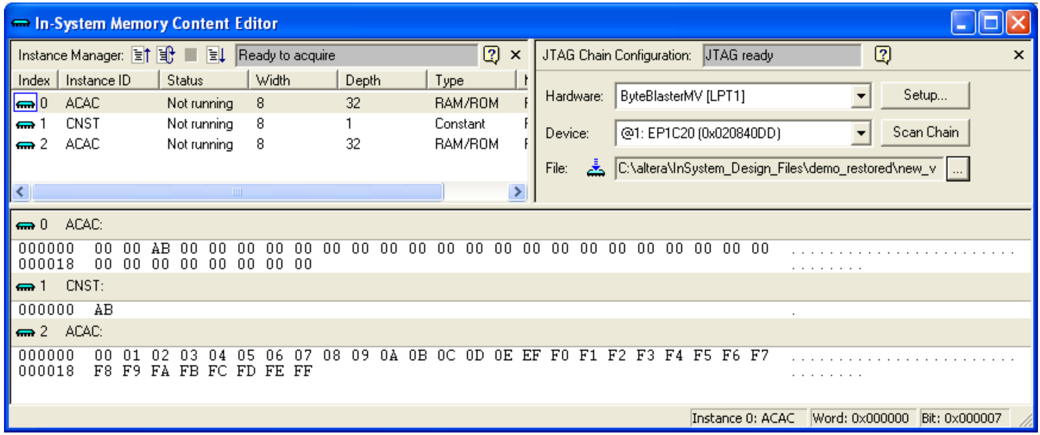
In Verilog HDL code, add the following:

```
defparam <megafunction instance name>.lpm_hint =
  "ENABLE_RUNTIME_MOD = YES,
  INSTANCE_NAME = <instantiation name>";
```

Running the In-System Memory Content Editor

The In-System Memory Content Editor is separated into the Instance Manager, JTAG Chain Configuration, and the Hex Editor (Figure 16–1).

Figure 16–1. In-System Memory Content Editor



The Instance Manager displays all available run-time modifiable memories and constants in your FPGA device. The JTAG Chain Configuration section allows you to program your FPGA and select the Altera® device in the chain to update.

Using the In-System Memory Content Editor does not require that you open a project. The In-System Memory Content Editor retrieves all instances of run-time configurable memories and constants by scanning the JTAG chain and sending a query to the specific device selected in the JTAG Chain Configuration section.

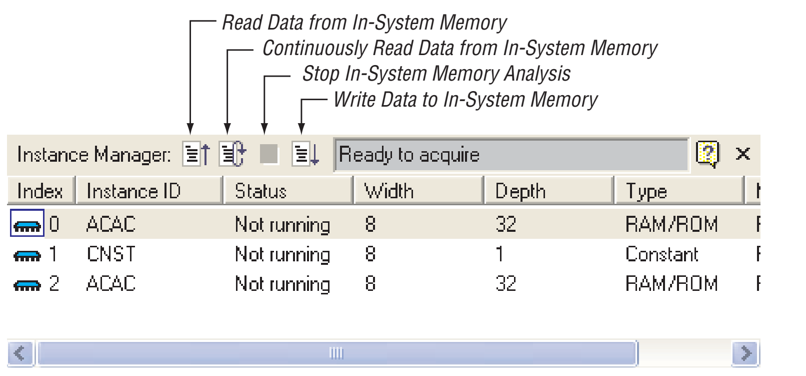
Each In-System Memory Content Editor can access the in-system memories and constants in a single device. If you have more than one device containing in-system configurable memories or constants in a JTAG chain, you can launch multiple In-System Memory Content Editors within the Quartus II software to access the memories and constants in each of the devices.

Instance Manager

Scan the JTAG chain to update the Instance Manager with a list of all run-time modifiable memories and constants in the design. The Instance Manager displays the Index, Instance, Status, Width, Depth, Type, and Mode of each element in the list.

You can read and write to in-system memory using the Instance Manager, as shown in Figure 16–2.

Figure 16–2. Instance Manager Controls



© March 2009 Altera Corporation

The following buttons are provided in the Instance Manager:

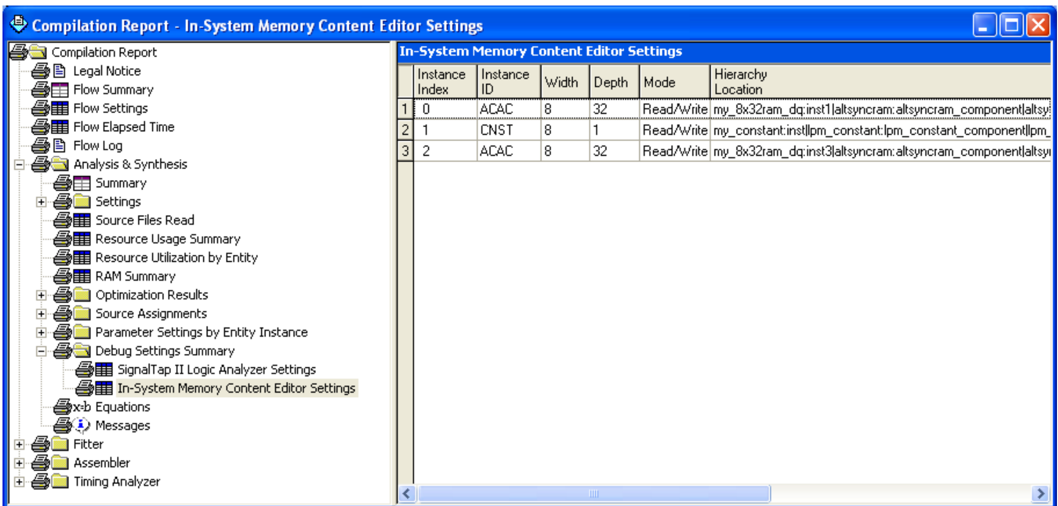
- **Read data from In-System Memory**—Reads the data from the device independently of the system clock and displays it in the Hex Editor
- **Continuously Read Data from In-System Memory**—Continuously reads the data asynchronously from the device and displays it in the Hex Editor
- **Stop In-System Memory Analysis**—Stops the current read or write operation
- **Write Data to In-System Memory**—Asynchronously writes data present in the Hex Editor to the device

In addition to the buttons available in the Instance Manager, you can also read and write data by selecting the command from the Processing menu, or the right button pop-up menu in the Instance Manager or Hex Editor.

The status of each instance is also displayed beside each entry in the Instance Manager. The status indicates if the instance is **Not running**, **Offloading data**, or **Updating data**. The health monitor provides useful information about the status of the editor.

The Quartus II software assigns a different index number to each in-system memory and constant to distinguish between multiple instances of the same memory or constant function. View the **In-System Memory Content Editor Settings** section of the compilation report to match an index with the corresponding instance ID (Figure 16–3).

Figure 16–3. Compilation Report In-System Memory Content Editor Settings Section



Editing Data Displayed in the Hex Editor

You can edit the data read from your in-system memories and constants displayed in the Hex Editor by typing values directly into the editor or by importing memory files.

To modify the data displayed in the Hex Editor, click a location in the editor and type or paste in the new data. The new data appears in blue, indicating modified data that has not been written into the FPGA. On the Edit menu, choose **Value**, and click **Fill with 0's**, **Fill with 1's**, **Fill with Random Values**, or **Custom Fills to** update a block of data by selecting a block of data.

Importing and Exporting Memory Files

The In-System Memory Content editor allows you specify a file to import and export data values to and from memories that have the In-System Updating feature enabled. Importing from a data files enables you to quickly load an entire memory image. Exporting to a data file enables you to save the contents of the memory for future use and analysis.

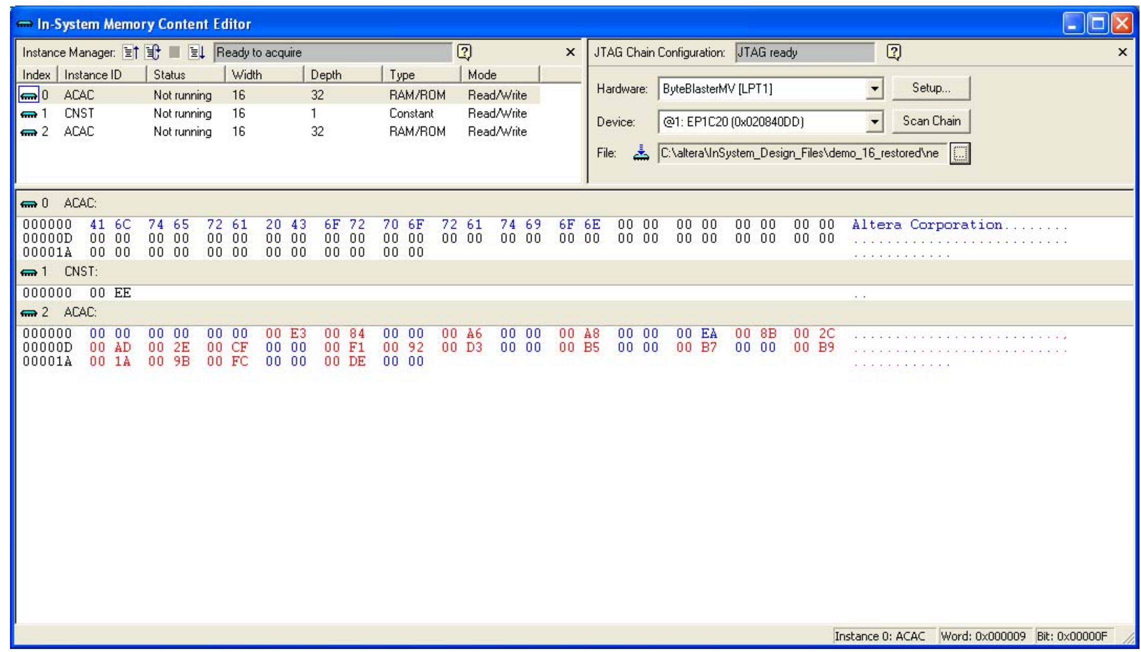
To import a file to memory using the In-System Memory Content Editor, select the memory or constant that you want to target from the instance manager. From the Edit menu, click **Import Data from File**, and specify the data file that you want to load to the targeted memory or constant. You can only import a memory file that is in either a Hexadecimal (Intel-Format) file (**.hex**) or memory initialization file (**.mif**) format.

Similarly, to export the contents of memory to a file using the In-System Memory Content Editor, select the memory or constant that you want to target from the instance manager. From the Edit menu, click **Export Data from File**, and specify the file name to which you want to save the data. You can export data to a **.hex**, **.mif**, Verilog Value Change Dump file (**.vcd**), or RAM Initialization file (**.rif**) format.

Viewing Memories and Constants in the Hex Editor

For each instance of an in-system memory or constant, the Hex Editor displays data in hexadecimal representation and ASCII characters (if the word size is a multiple of 8 bits). The arrangement of the hexadecimal numbers depends on the dimensions of the memory. For example, if the word width is 16 bits, the Hex Editor displays data in columns of words that contain columns of bytes (Figure 16–4).

Figure 16–4. Editing 16-Bit Memory Words Using the Hex Editor



Unprintable ASCII characters are represented by a period (.). The color of the data changes as you perform reads and writes. Data displayed in black indicates the data in the Hex Editor was the same as the data read from the device. If the data in the Hex Editor changes color to red, the data previously shown in the Hex Editor was different from the data read from the device.

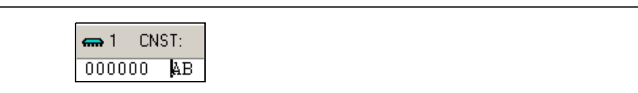
As you analyze the data, you can use the cursor and the status bar to quickly identify the exact location in memory. The status bar is located at the bottom of the In-System Memory Content Editor and displays the selected instance name, word position, and bit offset (Figure 16–5).

Figure 16–5. Status Bar in the In-System Memory Content Editor

The bit offset is the bit position of the cursor within the word. In the following example, a word is set to be 8-bits wide.

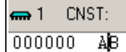
With the cursor in the position shown in Figure 16–6, the word location is 0x0000 and the bit position is 0x0007.

Figure 16–6. Hex Editor Cursor Positioned at Bit 0×0007



With the cursor in the position shown in Figure 16–7, the word location remains 0x0000 but the bit position is 0x0003.

Figure 16–7. Hex Editor Cursor Positioned at Bit 0×0003



Scripting Support

The In-System Memory Content Editor supports reading and writing of memory contents via a Tcl script or Tcl commands entered at a command prompt. For detailed information about scripting command options, refer to the Quartus II command-line and Tcl API Help browser.

To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

The commonly used commands for the In-System Memory Content Editor are as follows:

- Reading from memory:

```
read_content_from_memory
[-content_in_hex]
-instance_index <instance index>
-start_address <starting address>
-word_count <word count>
```

- Writing to memory:

```
write_content_to_memory
```

- Save memory contents to file:

```
save_content_from_memory_to_file
```

- Update memory contents from File:

```
update_content_to_memory_from_file
```

For descriptions about the command options and scripting examples, refer to the Tcl API Help Browser and the *Quartus II Scripting Reference Manual*.

Programming the Device Using the In-System Memory Content Editor

If you make changes to your design, you can program the device from within the In-System Memory Content Editor. To program the device, follow these steps:

1. On the Tools menu, click **In-System Memory Content Editor.**
2. In the **JTAG Chain Configuration** panel of the In-System Memory Content Editor, select the SRAM object file (**.sof**) that includes the modifiable memories and constants.

3. Click **Scan Chain**.
4. In the **Device** list, select the device you want to program.
5. Click **Program Device**.

Example: Using the In-System Memory Content Editor with the SignalTap II Embedded Logic Analyzer

The following scenario describes how you can use the In-System Updating of Memory and Constants feature with the SignalTap II Embedded Logic Analyzer to efficiently debug your design in-system. Although both the In-System Content Editor and the SignalTap II Embedded Logic Analyzer use the JTAG communication interface, you are able to use them simultaneously.

After completing your FPGA design, you find that the characteristics of your FIR filter design are not as expected.

1. To locate the source of the problem, change all your FIR filter coefficients to be in-system modifiable and instantiate the SignalTap II Embedded Logic Analyzer.
2. Using the SignalTap II Embedded Logic Analyzer to tap and trigger on internal design nodes, you find the FIR filter to be functioning outside of the expected cut-off frequency.
3. Using the **In-System Memory Content Editor**, you check the correctness of the FIR filter coefficients. Upon reading each coefficient, you discover that one of the coefficients is incorrect.
4. Because your coefficients are in-system modifiable, you update the coefficients with the correct data using the **In-System Memory Content Editor**.

In this scenario, you are able to quickly locate the source of the problem using both the In-System Memory Content Editor and the SignalTap II Embedded Logic Analyzer. You are also able to verify the functionality of your device by changing the coefficient values before modifying the design source files.

An extension to this example would be to modify the coefficients with the In-System Memory Content Editor to vary the characteristics of the FIR filter (for example, filter attenuation, transition bandwidth, cut-off frequency, and windowing function).

Conclusion

The In-System Updating of Memory and Constants feature provides access into a device for efficient debug in a hardware lab. You can use In-System Updating of Memory and Constants with the SignalTap II Embedded Logic Analyzer to maximize the visibility into an Altera FPGA. By increasing visibility and access to internal logic of the device, you can identify and resolve problems with your design more easily.

Referenced Documents

This chapter references the following documents:

- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Design Debugging Using the SignalTap II Embedded Logic Analyzer* chapter in volume 3 of the *Quartus II Handbook*

- *Quartus II Scripting Reference Manual*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 16–3 shows the revision history of this chapter.

Table 16–3. Document Revision History

Date and Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change to content.	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Added reference to Section V. In-System Debugging in volume 3 of the Quartus II Handbook on page 16-1. ■ Removed references to the Mercury device, as it is now considered to be a “Mature” device ■ Added links to referenced documents throughout document ■ Minor editorial updates	Updated for the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

17. Design Debugging Using In-System Sources and Probes

QII53021-9.0.0

Introduction

Traditional debugging techniques often involve using an external pattern generator to exercise the logic and a logic analyzer to study the output waveforms during run-time. The SignalTap® II Logic Analyzer and SignalProbe allow you to read or "tap" internal logic signals during run-time as a way to debug your logic on-chip. While this is useful, you can enhance the debugging cycle efficiency with the ability to drive any internal signal manually within your design. By doing this you can perform the following activities:

- Force the occurrence of trigger conditions setup in the SignalTap II Logic Analyzer
- Create simple test vectors to exercise your design without using external test equipment
- Dynamically control run-time control signals with the JTAG chain

With the introduction of the In-System Sources and Probes feature in the Quartus® II software beginning with version 7.1, Altera extends the portfolio of verification tools. The In-System Sources and Probes feature allows you to easily control any internal signal, providing you with a completely dynamic debugging environment. Coupled with either the SignalTap II Logic Analyzer or SignalProbe, the In-System Sources and Probes feature gives you a powerful debugging environment in which to generate stimuli and solicit responses from your logic design.

The Virtual JTAG Megafunction and the In-System Memory Content Editor also give you the capability to drive virtual inputs into your design. The Quartus II software offers a portfolio of on-chip debugging tools. For an overview and comparison of all the tools available in the Quartus II software on-chip debugging tool suite, refer to *Section V. In-System Debugging* in volume 3 of the *Quartus II Handbook*.

This chapter addresses the following topics:

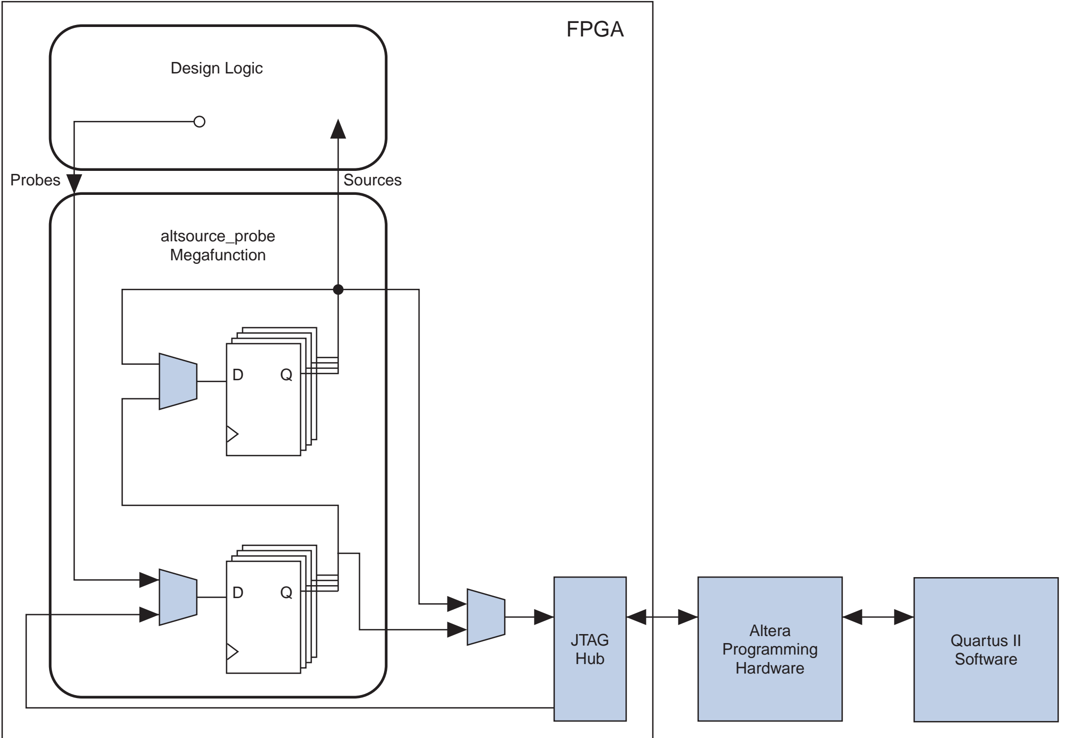
- "Design Flow Using In-System Sources and Probes" on page 17–3
- "Running the In-System Sources and Probes Editor" on page 17–7
- "Tcl Support" on page 17–11
- "Design Example: Dynamic PLL Reconfiguration" on page 17–14

Overview

The In-System Sources and Probes feature consists of the altsource_probe megafunction and an interface to control the altsource_probe megafunction instances during run-time. Each altsource_probe megafunction instance provides you with source output ports and probe input ports, where source ports drive selected signals and probe ports sample selected signals. Upon compilation, the altsource_probe

megafunction sets up a register chain to either drive or sample the selected nodes in your logic design. During runtime, the In-System Sources and Probes interface uses a JTAG connection to shift data to and from the altsource_probe megafunction instances. Figure 17–1 shows a block diagram of the components that make up the In-System Sources and Probes feature.

Figure 17–1. In-System Sources and Probes Block Diagram



The altsource_probe megafunction hides the detailed transactions between the JTAG Hub and the registers instrumented in your design to give you a basic building block for stimulating and probing your design. Moreover, the In-System Sources and Probes feature provides single-cycle samples and single-cycle writes to the selected logic nodes. This provides an easy way to input simple virtual stimuli and an easy way to capture the current value on instrumented nodes. Because In-System Sources and Probes gives you access to logic nodes within your design, this feature can be used during the debugging process to toggle the inputs of low-level components. If used in conjunction with the SignalTap II Logic Analyzer, you can force trigger conditions to help isolate your problem and shorten your debugging process.

Additionally, the ease of use of the In-System Sources and Probes feature makes it ideal for implementing control signals as virtual stimuli. This feature can be especially helpful for prototyping your design. Examples of such applications could include the ability to do the following:

- Create virtual push buttons
- Create a virtual front panel to interface with your design
- Mimic external sensor data
- Monitor and change run-time constants on the fly

In-System Sources and Probes supports Tcl commands to interface with all your altsource_probe instances to increase the level of automation.

Hardware and Software Requirements

The following components are required to use In-System Sources and Probes:

- Quartus II design software

or

- Quartus II Web Edition (with TalkBack feature enabled)
- Download Cable (USB-Blaster™ download cable or ByteBlaster™ cable)
- Altera® development kit or user design board with JTAG connection to device under test

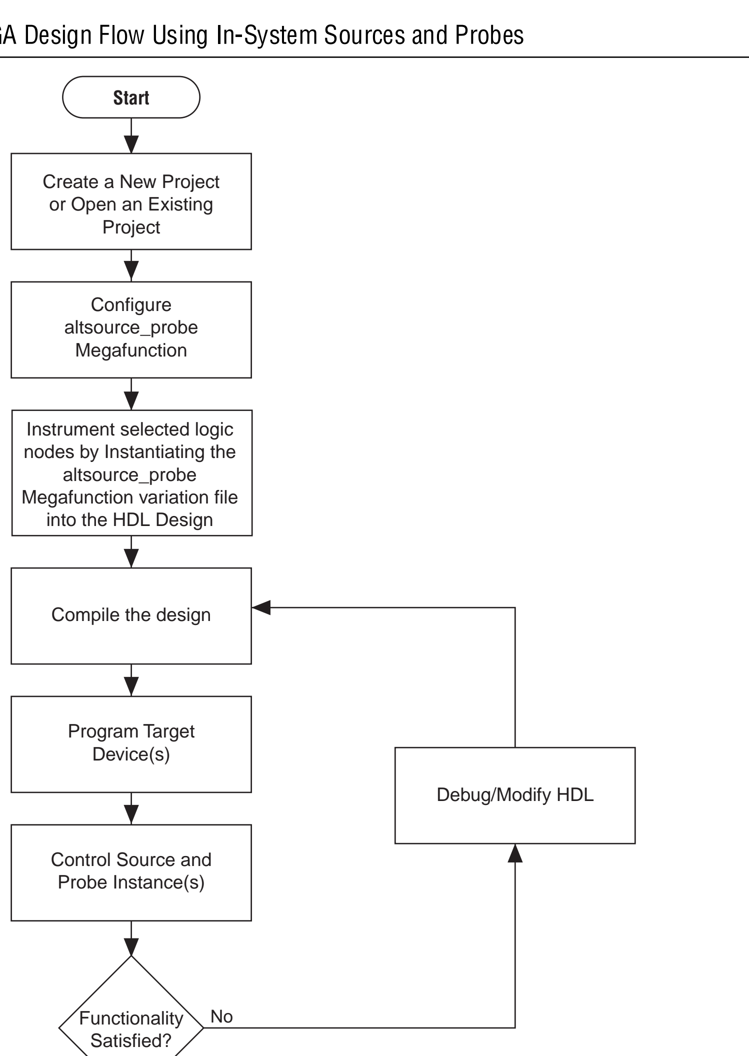
The In-System Sources and Probes feature supports the following device families:

- Arria® GX
- Stratix® series
- HardCopy® II
- HardCopy Stratix
- Cyclone® series
- MAX® II
- APEX™ II
- APEX 20KE
- APEX 20KC
- APEX 20K

Design Flow Using In-System Sources and Probes

In-System Sources and Probes supports an RTL flow in which your design nodes are instrumented in your HDL code via instantiation of the altsource_probe megafunction. After your device is compiled with the design nodes that you want instrumented, you can control your altsource_probe instances via the Sources and Probes Editor GUI or via a Tcl interface. The complete design flow is shown in Figure 17–2.

Figure 17–2. FPGA Design Flow Using In-System Sources and Probes



Configuring the altsource_probe Megafunction

To add in-system sources and probes functionality to your design, you must first instantiate the altsource_probe megafunction variation file. The altsource_probe megafunction can be easily configured using the MegaWizard™ Plug-In Manager. Each source or probe port can be up to 256 bits wide. You can have up to 128 instances of the altsource_probe megafunction in your design. The following steps will guide you through the steps necessary to configure the altsource_probe megafunction:

1. On the Tools menu, click **MegaWizard Plug-In Manager**.
2. Select **Create a new custom megafunction variation**.
3. Click **Next**.

4. On Page 2a, make the following selections:
   a. In the Installed Plug-Ins list, expand the JTAG-accessible Extensions folder. In the JTAG-accessible Extensions list, select **In-System Sources and Probes**.
   b. Make sure that the currently selected device family matches the device you are targeting.
   c. Select an output file type and enter the desired name of the altsource_probe megafunction. You can choose AHDL (**.tdf**), VHDL (**.vhd**), or Verilog HDL (**.v**) as the output file type.
5. Click **Next**.
6. On Page 3, make the following selections:
   a. Make sure that the currently selected device family matches the device that you are targeting.
   b. Under **Do you want to specify an Instance Index?**, turn on **Yes**.
   c. Specify the Instance ID of this instance.
   d. Specify the width of the probe port. The width can be from 1 bit to 256 bits wide.
   e. Specify the width of the source port. The width can be from 1 bit to 256 bits wide.
7. On Page 3, you can click **Advanced Options** and specify other parameters. The following options are included:
   - **What is the initial value of the source port, in hexadecimal?** This option allows you to specify the initial value driven on the source port at run-time.
   - **Write data to the source port synchronously to the source clock.** This allows you to synchronize your source port write transactions with the clock domain of your choice.
   - **Create an enable signal for the registered source port.** When enabled, this creates a clock enable input for the synchronization registers. This option is enabled only when the **Write data to the source port synchronously to the source clock** option is enabled.

Table 17–1 summarizes the configurable fields for the altsource_probe megafunction.

The In-System Sources and Probes feature does not support simulation. The megafunction instantiation must be removed before creating a simulation netlist.

Table 17–1. MegaWizard Plug-In Manager —altsource_probe (page 3) Options (Part 1 of 2)

Options	Description
Currently selected device family	Specifies the device family.
Do you want to specify an Instance Index?	Specifies the numeric index of the megafunction instance during run-time (from 0 to 127).
The 'Instance ID' of this Instance (optional):	Specifies the four character ID tag of the megafunction in the instance manager window of the Sources and Probes Editor.

Table 17–1. MegaWizard Plug-In Manager —altsource_probe (page 3) Options (Part 2 of 2)

Options	Description
How wide should the probe port be?	Specifies the number of signals to be read by In-System Sources and Probes.
How wide should the source port be?	Specifies the number of signals to be driven by In-System Sources and Probes.
What is the initial value of the source port? (under **Advanced Options**)	Specifies the initial value driven on the source port at run time.
Write data to the source port synchronously to the source clock. Each bit in the source port will utilize two additional registers to achieve metastability (under **Advanced Options**)	When turned on, allows you to synchronize your source port write transactions with the clock domain of your choice.
Create an enable signal for the registered source port (configured under **Advanced Options**)	Turning on this option creates a clock enable input for the synchronization registers.

Instantiating the altsource_probe Megafunction

The MegaWizard Plug-in Manager produces the necessary variation file and the instantiation template based on your inputs to the MegaWizard. Use the template to instantiate the ALTSOURCE_PROBE megafunction variation file in your design. The port information is shown in Table 17–2.

Table 17–2. altsource_probe Megafunction Port Information

Port Name	Required?	Direction	Comments
`probe[]`	No	Input	The outputs from the user's design.
`source_clk`	No	Input	Source Data is written synchronously to this clock. This input is required if the Source Clock option is turned on in the Advanced Options box in the MegaWizard Plug-in Manager.
`source_ena`	No	Input	Clock enable signal for `source_clk`. This input is required if specified in the Advanced Options box in the MegaWizard Plug-in Manager.
`source[]`	No	Output	Used to drive inputs to user design.

You can include up to 128 instances of the altsource_probe megafunction in your design, provided that there are available logic resources in your device. Each instance of the altsource_probe megafunction uses a pair of registers per signal for the width of the widest port it contains. Additionally, there will be some fixed overhead logic to accommodate communication between the altsource_probe instances and the JTAG controller. An additional pair of registers per source port is added for synchronization if it is specified.

Compiling the Design

When you compile your design with the In-System Sources and Probes megafunction instantiated, an instance of the altsource_probe instance and sld_hub megafunctions are added to your compilation hierarchy automatically. These two instances allow communication between the JTAG controller and your instrumented logic.

To modify your In-System Sources and Probes connections, you can modify the number of connections to your design by editing the altsource_probe megafunction. To open the design instance you want to modify in the MegaWizard Plug-In Manager, double-click the desired instance in the Project Navigator. You can then modify the connections in the HDL source file. You must recompile your design when you are finished editing it.

Because the design cycle is iterative in nature, you can use the Quartus II incremental compilation feature to reduce compilation time. Incremental compilation allows you to organize your design into logical partitions. During recompilation of a design, incremental compilation preserves the compilation results and performance of unchanged partitions and reduces design iteration time by compiling only modified design partitions.

For more information about Incremental Compilation, refer to the *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

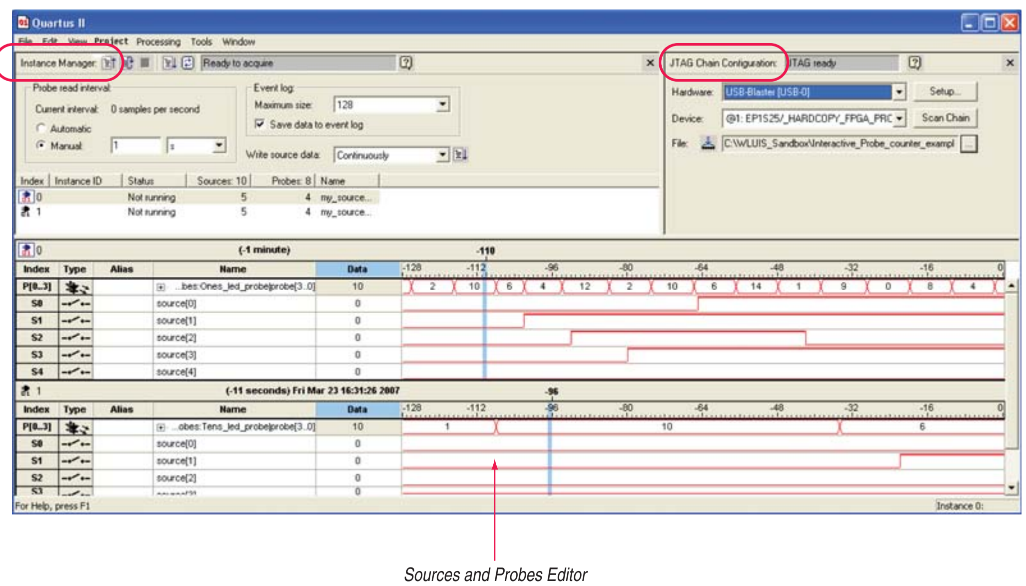
Running the In-System Sources and Probes Editor

The In-System Sources and Probes Editor is a GUI that gives you control over all of the altsource_probe megafunction instances within your design. It displays all available run-time controllable instances of the altsource_probe megafunction in your design, provides a push-button interface to drive all of your source nodes, and a logging feature to store your probe and source data.

To run the In-System Sources and Probes Editor, from the **Tools** menu, click **In-System Sources and Probes Editor.**

Figure 17–3 shows the Editor window.

Figure 17–3. In-System Sources and Probes Editor



The In-System Sources and Probes Editor is made up of three panes:

- **JTAG Chain Configuration**—Allows you to specify programming hardware, device, and file settings that the In-System Sources and Probes Editor uses to program and acquire data from a device.
- **Instance Manager**—Displays information about the instances generated when you compile a design, and allows you to control the data the In-System Sources and Probes Editor acquires.
- **Sources and Probes Editor**—Logs all the data read from the selected instance and allows you to modify source data to be written to your device.

Using the In-System Sources and Probes Editor does not require you to open a Quartus II project. The In-System Sources and Probes Editor retrieves all instances of the altsource_probe megafunction by scanning the JTAG chain and sending a query to the specific device selected in the JTAG Chain Configuration pane. Also, you can use a previously saved configuration to run the In-System Sources and Probes Editor.

Each In-System Sources and Probes Editor window can access the altsource_probe megafunction instances in a single device. If you have more than one device containing megafunction instances in a JTAG chain, you can launch multiple In-System Sources and Probes Editor windows to access the megafunction instances in each of the devices.

Programming Your Device Using the JTAG Chain Configuration Window

After compilation is complete, you must configure your FPGA before using In-System Sources and Probes. To configure a device for use with the In-System Sources and Probes, perform the following steps:

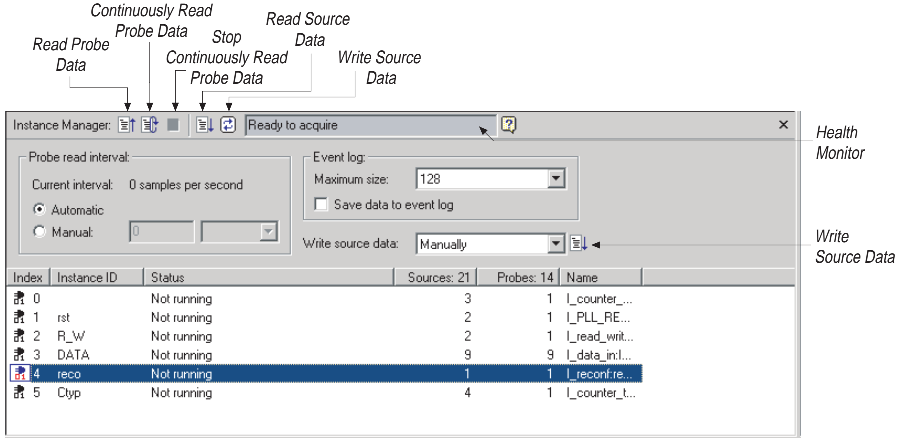
1. Open the In-System Sources and Probes Editor.
2. Under JTAG Chain Configuration, point to **Hardware** and select the desired hardware communications device. You may be prompted to configure your hardware; in this case, click **Setup**.
3. From the **Device** list, select the FPGA device to which you want to download the design (it may be automatically detected). You may have to click **Scan Chain** to detect your target device.
4. In the JTAG Configuration window, click **Browse** and select the SRAM Object File (**.sof**) that includes the In-System Sources and Probes instance or instances. (Note that it may be automatically detected).
5. Click **Program Device** (next to **File:**) to program the target device.

Instance Manager

The Instance Manager provides a list of all altsource_probe instances in the design and allows you to configure how data is acquired from or written to those instances.

The Instance Manager is shown in Figure 17–4.

Figure 17–4. Instance Manager



The following buttons and sub-panes are provided in the Instance Manager:

- **Read Probe Data**—Samples the probe data in the selected instance and displays it in the Sources and Probes Editor Window
- **Continuously Read Probe Data**—Continuously samples the probe data of the selected instance and displays it in the Sources and Probes Editor Window; you can modify the sample rate via the Probe read interval setting
- **Stop Continuously Reading Probe Data**—Cancels continuous sampling of probe of selected instance
- **Write Source Data** sub-pane —Writes data to all source nodes of the selected instance
- **Probe Read Interval** sub-pane—Displays the sample interval of all the In-system Sources and Probe instances in your design; you can modify the sample interval by clicking **Manual**
- **Event Log** sub-pane—controls the event log in the Sources and Probes Editor Window
- **Write Source Data** sub-pane—Allows you to manually or continuously write data to the system

The status of each instance is also displayed beside each entry in the Instance Manager. The status indicates if the instance is **Not running Offloading data**, **Updating data**, or if an **Unexpected JTAG communication error** occurs. This status indicator provides useful information about the sources and probes instances in your design.

Sources and Probes Editor Window

The Sources and Probes Editor window organizes and displays the data from all sources and probes in your design, organized according to the index number of the instance. The editor provides an easy way to manage your signals, allowing you to rename signals or to group them into buses. All data collected from source and probe nodes is recorded in the event log and displayed as a timing diagram.

Reading Probe Data

You can read data by selecting the desired altsource_probe instance in the Instance Manager and clicking **Read Probe Data**. This produces a single sample of the probe data and updates the data column of the selected index in the Sources and Probes Editor window. You can save the data to an event log by turning on the **Save data to event log** option in the Instance Manager.

If you want to sample data from your probe instance continuously, in the Instance Manager, click the instance you want to read, and then click **Continuously read probe data**. While reading, the status of the active instance will show **Unloading**. You can read continuously from multiple instances.

You can access read data by using the right-click menus in the Instance Manager.

To adjust the probe read interval, in the Instance Manager, turn on the **Manual** option in the **Probe read interval** sub-pane, and specify the desired sample rate in the text field next to the **Manual** option. The maximum sample rate depends on your computer setup. The actual sample rate is shown in the **Current interval** box. The event log window buffer size can be adjusted in the **Maximum Size** box.

Writing Data

To modify the source data to be written into the altsource_probe instance, click in the name field of the signal you want to change. For buses of signals, you can double-click on the data field and type in the value to be driven out to the altsource_probe instance. The In-System Sources and Probes Editor stores the modified source data values in a temporary buffer. Modified values that have not been written out to the altsource_probe instances appear in red. To update the altsource_probe instance, highlight the instance in the Instance Manager and click **Write source data**. The **Write source data** function is also available via the shortcut menus in the Instance Manager.

You can choose to have the values stored in the In-System Sources and Probes Editor continuously update the altsource_probe instances. By doing so, any modifications you make to the source data buffer are written immediately to the altsource_probe instances. To continuously update the altsource_probe instances, change the **Write source data** field from **Manually** to **Continuously**.

Data Organization

The main editor window allows you to group signals into buses, and allows you to modify the display options of the data buffer.

To create a group of signals, select the node names you want to group, right-click and select **Group**. You can modify the display format in the Bus Display Format and the Bus Bit order submenus.

The Sources and Probes Editor Window allows you to rename any signal. To rename a signal, double-click the name of the desired signal and type in the new name.

The event log contains a record of the most recent samples. The buffer size is adjustable, up to 128k samples. The time stamp for each sample is logged and is displayed above the event log of the instance being examined as you move your mouse pointer over the data samples.

You can save the changes that you have made and the recorded data into a Sources and Probes File (**.spf**). To save changes, on the File menu, click **Save**. The file contains all of the modifications you made to the signal groups, as well as the current data event log.

Tcl Support

To support automation, In-system Sources and Probes supports the procedures described in this chapter in the form of Tcl commands. The Tcl package for In-System Sources and Probes is included by default when you run **quartus_stp**.

The Tcl interface for In-System Sources and Probes provides a powerful platform to help you debug your design. It is especially helpful for debugging designs that require toggling multiple sets of control inputs. You can aggregate multiple commands using a Tcl script to define your own custom command set.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Reference Manual*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Table 17–3 shows the Tcl commands you can use with In-System Sources and Probes.

Table 17–3. In-System Sources and Probes Tcl Commands (Part 1 of 2)

Command	Argument	Description
start_insystem_source_probe	-device_name *<device name>* -hardware_name *<hardware name>*	Opens a handle to a device using the specified hardware. Call this command before starting any transactions.
get_insystem_source_probe_instance_info	-device_name *<device name>* -hardware_name *<hardware name>*	Returns a list of all altsource_probe instances in your design. Each record returned will be in the following format: {*<instance Index>*, *<source width>*, *<probe width>*, *<instance name>*}
read_probe_data	-instance_index *<instance_index>* -value_in_hex *(optional)*	Retrieves the current value of the probe. A string is returned specifying the status of each probe, with the MSB as the left-most bit.
read_source_data	-instance_index *<instance_index>* -value_in_hex *(optional)*	Retrieves the current value of the sources. A string is returned specifying the status of each source, with the MSB as the left-most bit.

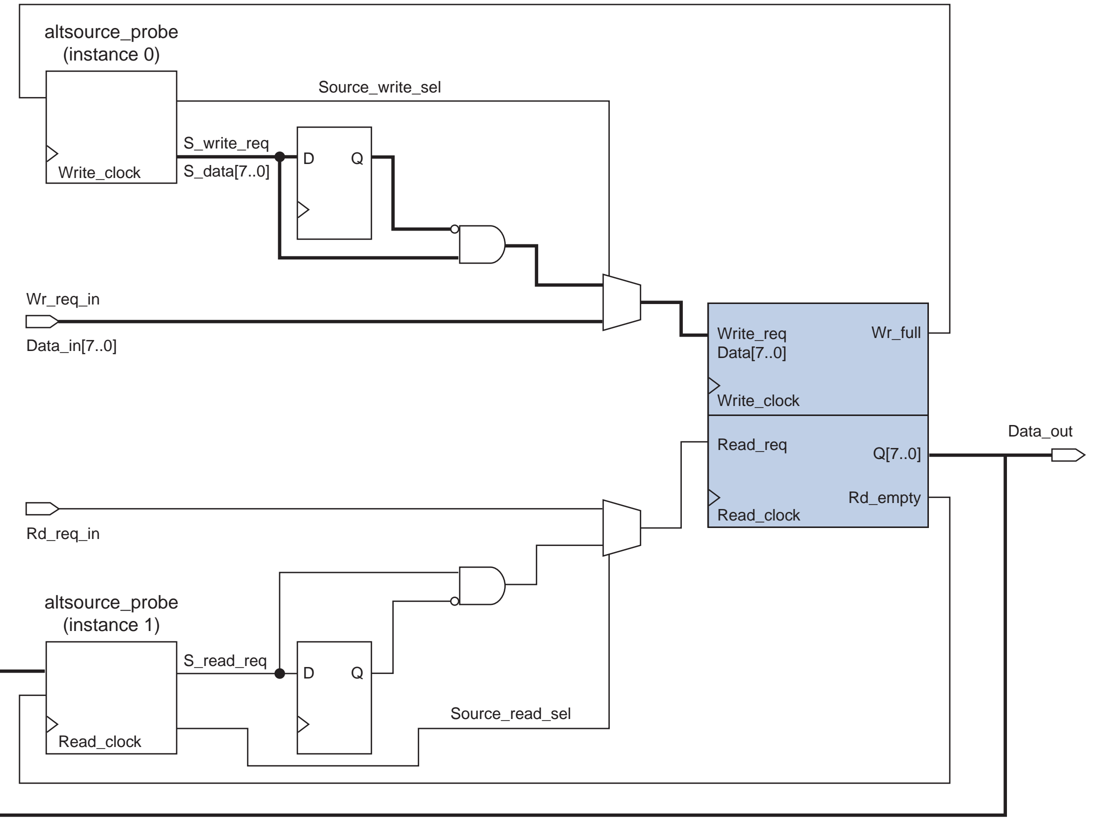
Table 17–3. In-System Sources and Probes Tcl Commands (Part 2 of 2)

Command	Argument	Description
write_source_data	`-instance_index` *`<instance_indx>`* `-value` *`<value>`* `-value_in_hex` *(optional)*	Sets the value of the sources. A binary string is sent to the source ports, with the MSB as the left-most bit.
end_interactive_probe	None	Releases the JTAG chain. Issue this command when all transactions are finished.

Example 17–1 shows an excerpt from a Tcl script with procedures that control the altsource_probe instances of the design as shown in Figure 17–5. The example design contains a DCFIFO with altsource_probe instances to read from and write to the DCFIFO. A set of control muxes are added into the design to control the flow of data to the DCFIFO between the input pins and the altsource_probe instances. A pulse generator is added to the read request and write request control lines to guarantee a single sample read or write. The altsource_probe instances, when used with the script in Example 17–1, provide visibility into the contents of the FIFO by performing single sample write and read operations and reporting the state of the full and empty status flags.

The Tcl script can be useful in debugging situations where you may want to either empty or preload the FIFO in your design. For example, you can use this feature to preload the FIFO to match a trigger condition you have set up within the SignalTap II Logic Analyzer.

Figure 17–5. A DCFIFO Example Design Controlled by the Tcl Script in Example 17–1



Example 17–1. Tcl Script Procedures for Reading and Writing to the DCFIFO in Figure 17–5 (Part 1 of 2)

```
## Setup USB hardware  - assumes only USB Blaster is installed and
## an FPGA is the only device in the JTAG chain

set usb [lindex [get_hardware_names] 0]
set device_name [lindex [get_device_names -hardware_name $usb] 0]
## write procedure :  argument value is integer

proc write {value} {

    global device_name usb
    variable full

    start_insystem_source_probe -device_name $device_name -hardware_name $usb

    #read full flag
    set full [read_probe_data -instance_index 0]


    if {$full == 1} {end_insystem_source_probe
    return "Write Buffer Full"
    }
```

Example 17–1. Tcl Script Procedures for Reading and Writing to the DCFIFO in Figure 17–5 (Part 2 of 2)

```
    ##toggle select line, drive value onto port, toggle enable
    ##bits 7:0 of instance 0 is S_data[7:0]; bit 8 = S_write_req;
    ##bit 9 = Source_write_sel

    ##int2bits is custom procedure that returns a bitstring from an integer
    ## argument


    write_source_data -instance_index 0 -value /[int2bits [expr 0x200 | $value]]
    write_source_data -instance_index 0 -value [int2bits [expr 0x300 | $value]]

    ##clear transaction

    write_source_data -instance_index 0 -value 0

    end_insystem_source_probe
}

proc read {} {

    global device_name usb
    variable empty
    start_insystem_source_probe -device_name $device_name -hardware_name $usb

    ##read empty flag : probe port[7:0] reads FIFO output; bit 8 reads empty_flag

    set empty [read_probe_data -instance_index 1]


    if {[regexp {1........} $empty]} { end_insystem_source_probe
    return "FIFO empty" }

    ## toggle select line for read transaction
    ## Source_read_sel = bit 0; s_read_reg = bit 1

    ## pulse read enable on DC FIFO
    write_source_data -instance_index 1 -value 0x1 -value_in_hex
    write_source_data -instance_index 1 -value 0x3 -value_in_hex

    set x [read_probe_data -instance_index 1 ]

    end_insystem_source_probe

    return $x
}
```

Design Example: Dynamic PLL Reconfiguration

The ease of use of the In-System Sources and Probes feature can be extremely helpful in creating a virtual front panel during the prototyping phase of your design. Relatively simple designs of high functionality can be created in a short amount of time. The following PLL reconfiguration example demonstrates how the In-System Sources and Probes feature is used to provide a GUI to dynamically reconfigure a Stratix PLL.

Stratix PLLs allows you to dynamically update PLL coefficients during run-time. Each enhanced PLL within the Stratix device contains a register chain that allows you to modify the pre-scale counters (*m* and *n* values), output divide counters, and delay counters. In addition, the altpll_reconfig megafunction provides an easy interface to access the register chain counters. The altpll_reconfig megafunction provides a cache containing all modifiable PLL parameters. After you have updated all of the PLL parameters in the cache, the alt_pll_reconfig megafunction drives the PLL register chain to update the PLL with the updated parameters. Figure 17–6 shows a Stratix-enhanced PLL with reconfigurable coefficients.

Stratix II and Stratix III devices also allow you to dynamically reconfigure PLL parameters. For more information about these families, refer to the appropriate data sheet. For more information about dynamic PLL reconfiguration, refer to *AN 282: Implementing PLL Reconfiguration in Stratix & Stratix GX Devices* or *AN 367: Implementing PLL Reconfiguration in Stratix II Devices*.

Figure 17–6. Stratix-Enhanced PLL with Reconfigurable Coefficients

The following design example uses an altsource_probe instance to update the PLL parameters in the altpll_reconfig megafunction cache. The altpll_reconfig megafunction connects to an enhanced PLL in a Stratix FPGA to drive the register chain containing the PLL reconfigurable coefficients. This design example uses a Tcl/Tk script to generate a GUI where you can enter in new *m* and *n* values for the enhanced PLL. The Tcl script extracts the m and n values from the GUI, shifts the

values out to the altsource_probe instances to update the values in the altpll_reconfig megafunction cache and asserts the reconfig signal on the altpll_reconfig megafunction. The reconfig signal on the altpll_reconfig megafunction starts the register chain transaction to update all PLL reconfigurable coefficients. A block diagram of a design example is shown in Figure 17–7. The Tk GUI is shown in Figure 17–8.

Figure 17–7. Block Diagram of Dynamic PLL Reconfiguration Design Example

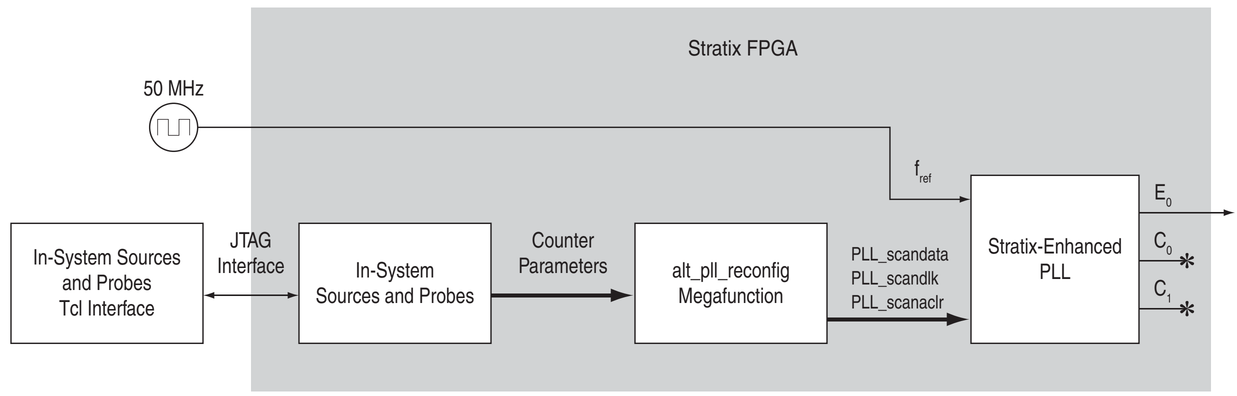
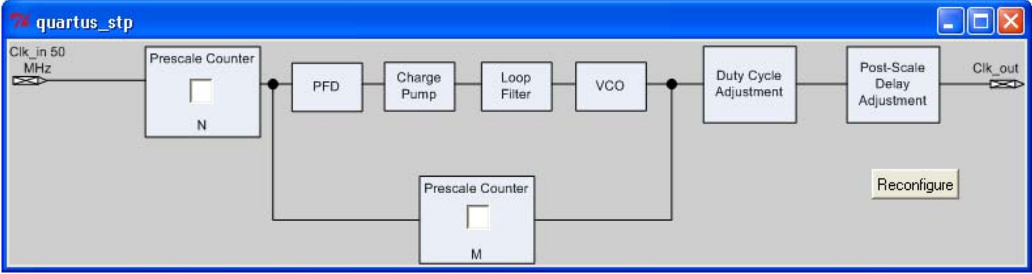


Figure 17–8. Interactive PLL Reconfiguration GUI Created with Tk and In-System Sources and Probes Tcl Package



This design example was created using a Nios® II Development Kit, Stratix Edition. The file **sourceprobe_DE_dynamic_pll.zip** contains all of the necessary files for running this design example:

- **Readme.txt**—A text file that describes the files contained in the Design Example and provides instructions on running the Tk GUI shown in Figure 17–8.
- **Interactive_Reconfig.qar**—The archived Quartus II project for this Design Example

You can download the **sourceprobe_DE_dynamic_pll.zip** file associated with this chapter.

Conclusion

In-System Sources and Probes can provide stimuli and get responses from the target design during run-time. With its simple and intuitive interface, you can provide virtual inputs into your design during run-time without using external equipment. When used in conjunction with SignalTap II, you can use In-System Sources and Probes to provide greater control of the signals in your design, and thus help shorten the verification cycle. Also, with its ability to create virtual inputs into your design, you can create simple, yet powerful applications to interact with your design.

Referenced Documents

This chapter references the following documents:

- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical & Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Settings File Reference Manual*
- *sld_virtual_jtag Megafunction User Guide*
- *Section V. In-System Debugging* in volume 3 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 17–4 shows the revision history for this chapter.

Table 17–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change to content.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Documented that this feature does not support simulation on page 17–5 ■ Updated Figure 17–8 for Interactive PLL reconfiguration manager ■ Added hyperlinks to referenced documents throughout the chapter ■ Minor editorial updates	Updated for the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

Section V. Formal Verification

The Quartus® II software easily interfaces with EDA formal design verification tools such as the Cadence Encounter Conformal and Synopsys Synplify software. In addition, the Quartus II software has built-in support for verifying the logical equivalence between the synthesized netlist from Synopsys Synplify and the post-fit Verilog Quartus Mapped (**.vqm**) files using Cadence Encounter Conformal software.

This section discusses formal verification, how to set-up the Quartus II software to generate the **.vqm** file and Cadence Encounter Conformal script, and how to compare designs using Cadence Encounter Conformal software.

This section includes the following chapter:

- Chapter 18, Cadence Encounter Conformal Support

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

18. Cadence Encounter Conformal Support

QII53011-9.0.0

Introduction

The Quartus® II software provides formal verification support for Altera® designs through interfaces with a formal verification EDA tool, the Cadence Encounter Conformal software.

Use the Encounter Conformal software to verify the functional equivalence of a post-synthesis Verilog Quartus Mapping (**.vqm**) netlist file from Synopsys Synplify Pro software, a post-fit Verilog Output File (**.vo**) from the Quartus II software, or both. You can also use the Encounter Conformal software to verify the functional equivalence of the register transfer level (RTL) source code and post-fit **.vo** file with the Quartus II software when using Quartus II integrated synthesis. These formal verification flows support designs for the Arria® GX, Stratix® IV, Stratix III, Stratix II, Stratix, Stratix II GX, Stratix GX, Cyclone® III, Cyclone II, Cyclone, and HardCopy® II device families.

The two types of formal verification are equivalence checking and model checking. This chapter discusses equivalence checking with the Cadence Encounter Conformal software.

This chapter contains the following sections:

- "Formal Verification Design Flow" on page 18–2
- "RTL Coding Guidelines for Quartus II Integrated Synthesis" on page 18–4
- "Black Boxes in the Encounter Conformal Flow" on page 18–8
- "Generating the Post-Fit Netlist Output File and the Encounter Conformal Setup Files" on page 18–10
- "Understanding the Formal Verification Scripts for Encounter Conformal" on page 18–15
- "Comparing Designs Using Encounter Conformal" on page 18–17
- "Known Issues and Limitations" on page 18–19
- "Black Box Models" on page 18–21
- "Conformal Dofile/Script Example" on page 18–23

Equivalence checking uses mathematical techniques to compare the logical equivalence of two versions of the same design rather than using test vectors to perform simulation. The two compared versions could be post-map design and post-fit design, or RTL design and post-fit design. Equivalence checking greatly shortens the verification cycle of the design.

Formal Verification Versus Simulation

Formal verification cannot be considered as a replacement to the vector-based simulation. Formal verification only complements the existing vector-based simulation techniques to speed up the verification cycle. Vector-based simulation techniques of gate level designs can take a considerable amount of time.

You can use Vector-based simulation techniques to perform the following functions:

- Verify design functionality
- Verify timing specifications
- Debug designs

Formal Verification: What You Need to Know

If you use formal verification techniques to verify logic equivalence of your design, you can save time by forgoing a comprehensive vector-based simulation of the gate level design. However, there might be an impact on area and performance during recompilation of your design with the Quartus II software if you choose to use formal verification flow for Cadence Conformal LEC software. The area and performance of your design might be affected by the following factors:

- Hierarchy preservation
- ROM implementation by logic elements (LEs)
- Disabled retiming is disabled

Refer to "Known Issues and Limitations" on page 18–19 before you consider using the formal verification flow in your design methodology.

Formal Verification Design Flow

Altera supports formal verification using the Encounter Conformal software for the following two synthesis tools:

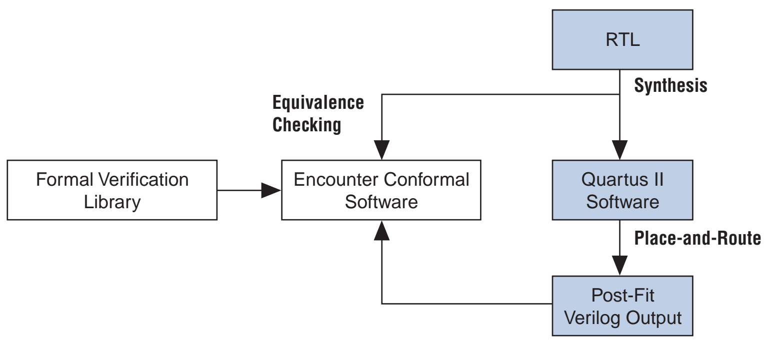
- "Quartus II Integrated Synthesis"
- "Synplify Pro" on page 18–3

The following sections describe the supported design flows for these synthesis tools.

Quartus II Integrated Synthesis

The design flow for formal verification using the Quartus II integrated synthesis is shown in Figure 18–1. This flow performs equivalency checking for the RTL source code and the post-fit netlist generated by the Quartus II software. The RTL source code can be in Verilog HDL or VHDL format. The post-fit netlist generated by the Quartus II software is always in Verilog HDL format.

Figure 18–1. Formal Verification Using Quartus II Integrated Synthesis and the Encounter Conformal Software



EDA Tool Support for Quartus II Integrated Synthesis

The formal verification flow using the Quartus II software and Cadence Encounter Conformal software supports the following software versions and operating systems:

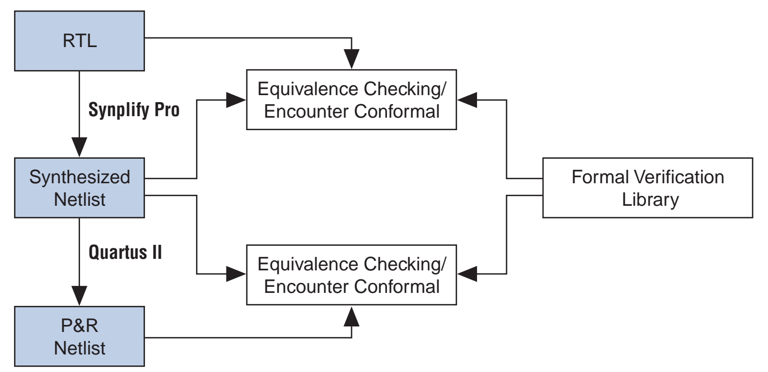
- Quartus II software beginning with version 4.2
- Cadence Encounter Conformal software beginning with 4.3.5A
- Solaris and Linux operating systems

Synplify Pro

The design flow for formal verification using Synplify Pro Synthesis performs equivalency checking for the post-synthesis netlist from Synplify Pro and the post fit netlist generated by Quartus II software, as shown in Figure 18–2.

For additional information about performing equivalency checking between RTL and post-synthesis netlist generated from Synplify Pro software, refer to the Synplify Pro documentation.

Figure 18–2. Formal Verification Flow Using Synplify Pro and the Encounter Conformal Software



EDA Tool Support for Synplify Pro

The formal verification flow using the Quartus II software, the Synopsys Synplify Pro, and the Cadence Encounter Conformal software supports the software versions and operating systems shown in Table 18–1.

Table 18–1. Compatible Software Versions

Quartus II Software Version	Cadence Conformal LEC Version	Synplify Pro Version
4.1	4.3.0.a	7.6.1
4.2	4.3.5.a	8.0
5.0	5.1	8.1
5.1	5.1	8.4
6.0	5.2	8.5
6.1	6.1	8.6.2
7.0	6.1	8.6.2
7.1	6.2	8.8.1
7.2	7.1	9.0
8.0	7.1	9.4
8.1	7.2	9.6.2
9.0	7.2	C2009.03

RTL Coding Guidelines for Quartus II Integrated Synthesis

The Cadence Encounter Conformal software can compare the RTL code against the post-fit netlist generated by the Quartus II software. The Encounter Conformal software and the Quartus II integrated synthesis parse and compile the RTL description in slightly different ways. The Quartus II software supports some RTL features that the Encounter Conformal software does not support and vice versa. The style of the RTL code is of particular concern because neither tool supports some constructs, leading to potential formal verification mismatches; for example, state machine extraction, wherein different encoding mechanisms can result in different structures. Therefore, for successful verification, both tools must interpret the RTL code in the same manner.

The following section provides information about recognizing and preventing problems that can arise in the formal verification flow.

For more details about RTL coding styles for Quartus II integrated synthesis, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Some of the coding guidelines apply to both the Quartus II integrated synthesis and Synplify Pro flow, as indicated in each of the guidelines in the following sections.

Synthesis Directives and Attributes

Synthesis directives, also known as pragmas, play an important role in successful verification of RTL against the post-fit **.vo** netlist file from the Quartus II software.

Pragmas and trigger keywords that are supported in Quartus II integrated synthesis and Encounter Conformal are also supported in the formal verification flow. The Quartus II integrated synthesis and Encounter Conformal both support the trigger keywords "synthesis" and "synopsys." When the Quartus II software does not recognize a keyword (such as "verplex"), the keyword is disabled in the formal verification scripts produced for use with the Cadence Conformal software. Therefore, it is important to use caution with unsupported pragmas because they can lead to verification mismatches.

For example, you can use the Quartus II integrated synthesis to synthesize RTL code with the synthesis directive `read_comments_as_HDL` (Example 18–1 and Example 18–2).

Example 18–1. Verilog HDL Example of Read Comments as HDL

```
// synthesis read_comments_as_HDL on
// my_rom lpm_rom (.address (address),
// .data (data));
// synthesis read_comments_as_HDL off
```

Example 18–2. VHDL Example of Read Comments as HDL

```
-- synthesis read_comments_as_HDL on
-- my_rom : entity lpm_rom
-- port map (
-- address => address,
-- data => data, );
-- synthesis read_comments_as_HDL off
```

The Encounter Conformal software does not support the synthesis directive `read_comments_as_HDL`, and the directive has no affect on the Encounter Conformal software.

Table 18–2 lists supported pragmas and trigger keywords for formal verification.

Table 18–2. Supported Pragmas and Trigger Keywords for Formal Verification

Pragmas *(1)*	Trigger Keywords
full_case	synthesis
parallel_case	synopsys
pragma	
synthesis_off	
synthesis_on	
translate_off	
translate_on	

Note to Table 18–2:

(1) Do not use Verilog 2001-style pragma declarations. The Quartus II software and the Encounter Conformal software support this style of pragma in different manners.

Stuck-at Registers

Quartus II integrated synthesis eliminates registers that have their output stuck at a constant value. Quartus II integrated synthesis gives a warning message and adds an entry to the corresponding report panel in the formal verification folder of the Analysis & Synthesis section of the Compilation Report. If Conformal LEC does not find the same optimizations, it can lead to unmapped points in the golden netlist. Example 18–3 illustrates the issue.

Example 18–3. Verilog HDL Example Showing Stuck at Registers

```
module stuck_at_example {clk, a,b,c,d,out};
input a,b,c,d,clk;
output out;
reg e,f,g;
   always @(posedge clk) begin
      e <= a and g;// e is stuck at 0
      g <= c and e;// g is stuck at 0
      f <= e | b;
   end
assign out = f and d;
endmodule
```

In this module description, registers e and g are tied to logic 0. In this example, the Quartus II software generates the following warning message:

```
Warning: Reduced register "g" with stuck data_in port to stuck value GND
Warning: Reduced register "e" with stuck data_in port to stuck value GND
```

Quartus II integrated synthesis then adds a command to the formal verification scripts telling Conformal LEC that a register is stuck at a constant value, as shown in Example 18–4.

Example 18–4. Conformal LEC Script Showing Commands for Instance Equivalence

```
// report floating signals
// Instance-constraints commands for constant-value registers removed
// during compilation
// add instance constraints 0 e -golden
// add instance constraints 0 g -golden
```

The command is commented in the formal verification script, forcing the Encounter Conformal software to treat the register as stuck at a constant value and potentially hiding a compilation error. You must verify that input to the e and g registers is constant in the design and uncomment the command to obtain accurate results.

Altera recommends recoding your design to eliminate “stuck-at” registers.

The stuck-at register information in this section also applies to the Synplify Pro flow.

ROM, LPM_DIVIDE, and Shift Register Inference

For the purpose of formal verification, the Quartus II integrated synthesis implements both ROM and shift registers in the form of LEs instead of using dedicated on-chip memory resources. Using LEs can be less area-efficient than inferring a megafunction that can be implemented in a RAM block. However, the Quartus II software generates a warning message indicating that the megafunction was not inferred. Quartus II

integrated synthesis also reports a suggested ROM or shift register instantiation that enables you to either use the MegaWizard™ Plug-In Manager to create the appropriate megafunction explicitly, or to isolate the corresponding logic in a separate entity that you can set as a black box. By setting black box properties on a particular module or entity, you are telling the formal verification tool not to look inside the module or entity for formal verification. If the black box properties are set on the corresponding megafunction before synthesis, you can verify the megafunction with the Encounter Conformal software. For details about setting black box properties on a particular module, refer to Table 18–3 on page 18–9.

If the design contains division functionality, the Quartus II software infers an LPM_DIVIDE megafunction, which is treated as a black box for the purpose of formal verification.

RAM Inference

When the Quartus II software infers the LPM ALTSYNCRAM megafunction from the RTL code, the Quartus II software generates the following warning message:

```
Created node "<mem_block_name>" as a RAM by generating altsyncram
megafunction to implement register logic with M512 or M4K memory block
or M-RAM. Expect to get an error or a mismatch for this block in the
formal verification tool.
```

This warning is generated because the memory block (`altsyncram`) is a new instance in the post-fit netlist that is handled as a black box by the formal verification tool. However, no such instance exists in the original RTL design, resulting in mismatch or error reporting in the formal verification tool.

Latch Inference

A latch is implemented in the Quartus II integrated synthesis using a combinational feedback loop. The Encounter Conformal software infers a latch primitive in the Encounter Conformal library (DLAT) to implement a latch. This results in having a DLAT on the golden side and a combinational loop with a cut point on the revised side, leading to verification mismatches. The Quartus II software issues a warning message whenever a latch is inferred, and the Quartus II software adds an entry to the report panel in the Formal Verification folder of the Analysis & Synthesis report. Altera recommends that you avoid latches in your design; however, if latches are necessary, Altera recommends using the corresponding LPM_LATCH megafunction.

For more information about the problems related to latches, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Combinational Loops

If the design consists of an intended combinational loop, you must define an appropriate cut point for both the RTL and the post-fit **.vo** netlist file. A warning that a combinational loop exists in the design is found in the Formal Verification subfolder of the Quartus II software Analysis & Synthesis report.

For more information on issues with combinational loops, refer to "Known Issues and Limitations" on page 18–19.

Finite State Machine Coding Styles

When a state machine is inferred by the Encounter Conformal software, it uses sequential encoding as the default encoding when no user encoding is present. The Quartus II software selects the encoding most suited for the inferred state machine if the **State Machine Processing** settings is set to the default value **Auto**. To do this, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. The **More Analysis & Synthesis Settings** dialog box appears.
4. Under **Option**, in the **Name** list, select **State Machine Processing**. In the **Setting** list, select **Auto**.
5. Click **OK**.
6. Click **OK**.

Use the coding style described in the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook* on RTL when writing finite state Machines (FSMs). This allows the Quartus II integrated synthesis and the Encounter Conformal software to infer a similar state machine for the same RTL code.

Black Boxes in the Encounter Conformal Flow

The Quartus II software usually generates a flattened netlist; however, you must treat some modules in the design as black boxes. The following is a list of some of these modules:

- LPMs and megafunctions without formal verification models
- Encrypted IP functions
- Entities not implemented in Verilog HDL or VHDL

To perform equivalence checking of a design between its version consisting of the modules listed above and its implemented version, the modules must be treated as black boxes by the Encounter Conformal software. To facilitate the formal verification flow, the Quartus II software reconstructs the hierarchy on the black boxes with a port interface that is identical to the module on the golden side of the design.

If your golden netlist (**.vqm** netlist file from Synplify Pro or RTL) includes any design entity not having a corresponding formal verification model, that entity is handled as a black box with its boundary interface preserved. There are three types of black boxes and required user actions, depending upon circumstances. Table 18–4 describes these three types of black boxes and the required user actions in detail.

Verilog Output netlist files written by the Quartus II software contain the black box hierarchy when you make the following assignments for a module:

- An EDA Formal Verification Hierarchy assignment with the value `BLACKBOX`
- A Preserve Hierarchical Boundary assignment with the value `Firm`

If these two assignments are not made for a module, the Quartus II software implements that module with logic cells. When this happens, the **.vo** netlist file no longer contains the black box hierarchy and does not preserve the port interface, resulting in a mismatch within the Encounter Conformal software.

Table 18–3. Black Boxes and Required User Action

Type of Black Box	Required User Action
Altera library of parameterized modules (LPMs) and megafunctions (refer to Table 18–5 on page 18–21 for a complete list).	No action required. The Quartus II software automatically creates a black box list of components and preserves the hierarchy.
Any parametrized entity other than those listed in Table 18–5 on page 18–21.	User must designate the wrapper that instantiates the parameterized entity as a black box.
Non-parameterized entities that the user wants designate as a black box.	User can designate the entity itself as a black box.

You can also use Tcl commands or Quartus II GUI to set the black box property on the entities, which the formal verification tool does not compare.

Tcl Command

Use the Tcl commands of Example 18–5 to preserve the boundary interface of a black box entity: `dram`.

Example 18–5. Tcl Command to Create a Black Box

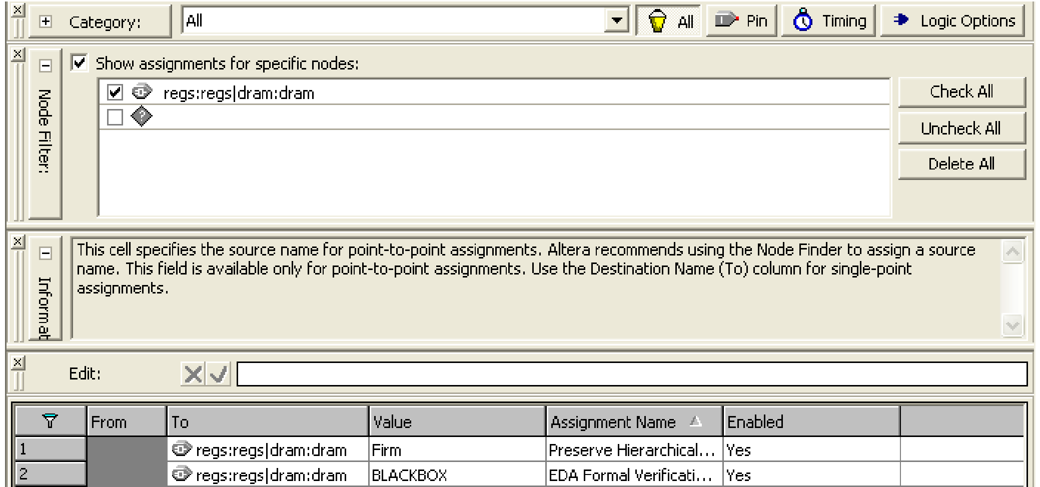
```
set_instance_assignment -name PRESERVE_HIERARCHICAL_BOUNDARY FIRM -to | -entity dram
set_instance_assignment -name EDA_FV_HIERARCHY BLACKBOX -to | -entity dram
```

GUI

To preserve the boundary interface of an entity using the GUI, perform the following steps:

1. Make an EDA Formal Verification Hierarchy assignment to the entity with the value `BLACKBOX`.
2. Make a Preserve Hierarchical Boundary assignment to the entity with the value `Firm` (Figure 18–3).

Figure 18–3. Setting the Black-Box Property on a Module



Generating the Post-Fit Netlist Output File and the Encounter Conformal Setup Files

The following steps describe how to set up the Quartus II software environment to generate the post-fit **.vo** netlist file and the Encounter Conformal script for use in formal verification. With the exception of step 2, the steps are identical for both of the Synthesis tools:

To create a new Quartus II project or open an existing project, perform the following steps:

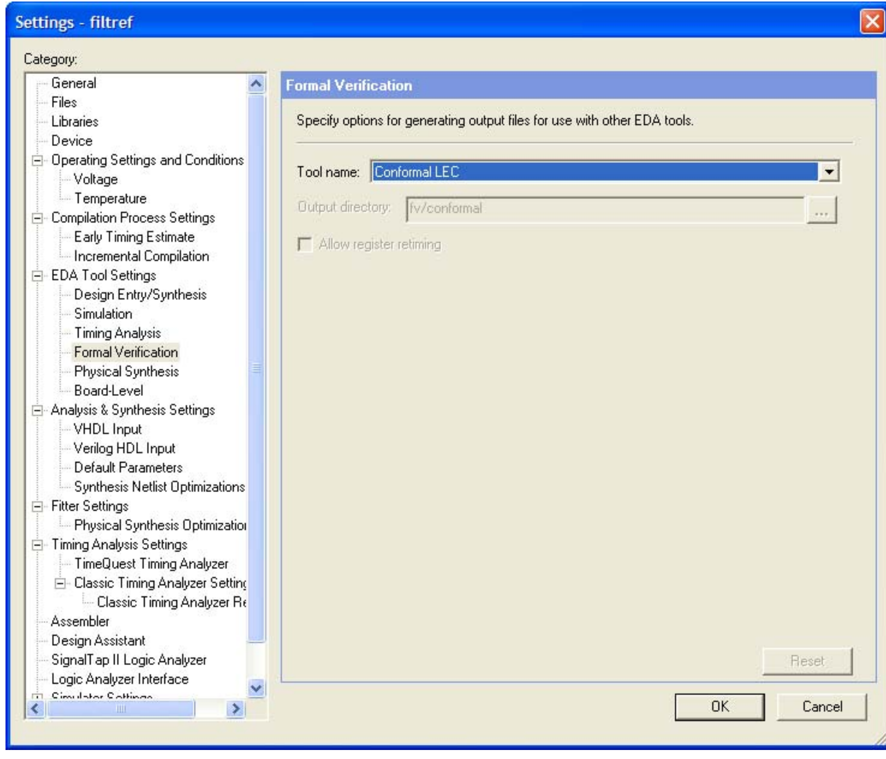
1. On the Assignments menu, click **EDA Tool Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click **EDA Tool Settings**.

   If you are using the Quartus II integrated synthesis, perform the following steps:

   a. In the **Category** list, under **EDA Tool Settings**, select **Design Entry/Synthesis**. Select **<None>** from the **Tool name** list.

   b. In the **Category** list, under **EDA Tool Settings**, select **Formal Verification**. Select **Conformal LEC** from the **Tool name** list (Figure 18–4).

Figure 18–4. Compilation Process Settings



If you are using Synplify Pro, perform the following steps:

a. In the **Category** list, under **EDA Tool Settings**, select **Design Entry/Synthesis**. Select **Synplify Pro** from the **Tool name** list.

b. In the **Category** list, under **EDA Tool Settings**, select **Formal Verification**. Select **Conformal LEC** from the Tool name list.

3. In the **Category** list, click the "+" icon to expand **Compilation Process Settings**, and select **Incremental Compilation**. The **Incremental Compilation** page appears.

4. Select **Full Incremental Compilation** to turn on Incremental Compilation.

or

Turn on Incremental Compilation by typing the following Tcl command in the Quartus II software Tcl console:

```
set_global_assignment -name INCREMENTAL_COMPILATION FULL_INCREMENTAL_COMPILATION
```

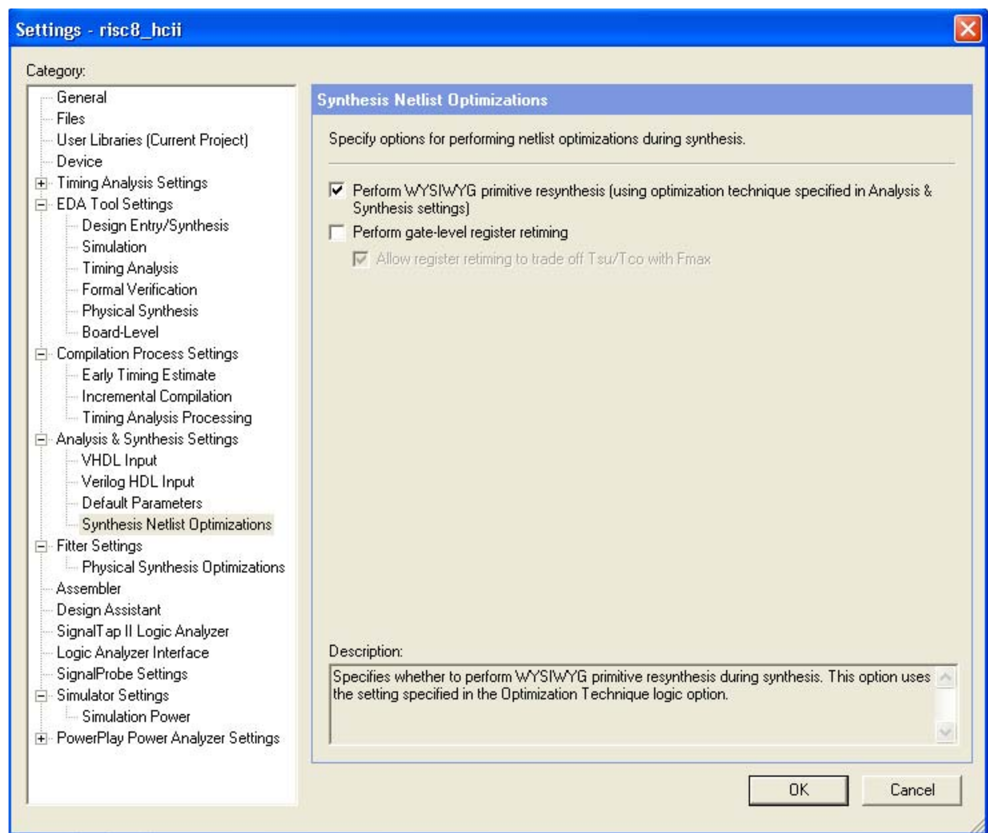
Altera requires that Incremental Compilation be turned on for Formal Verification, and that your design does not contain any user-created partitions. Starting with Quartus II software version 6.1 and later, the incremental compilation feature is on by default.

5. In the **Category** list, click the "+" icon to expand **Analysis and Synthesis Settings** and click **Synthesis Netlist Optimizations**. The **Synthesis Netlist Optimizations** page appears.

6. Turn off **Perform gate-level register retiming** (Figure 18–5).

If **Perform gate-level register retiming** is not turned off, the Encounter Conformal script can display a different set of compare points, making the resulting netlist difficult to compare against the reference netlist file.

Figure 18–5. Synthesis Netlist Optimizations

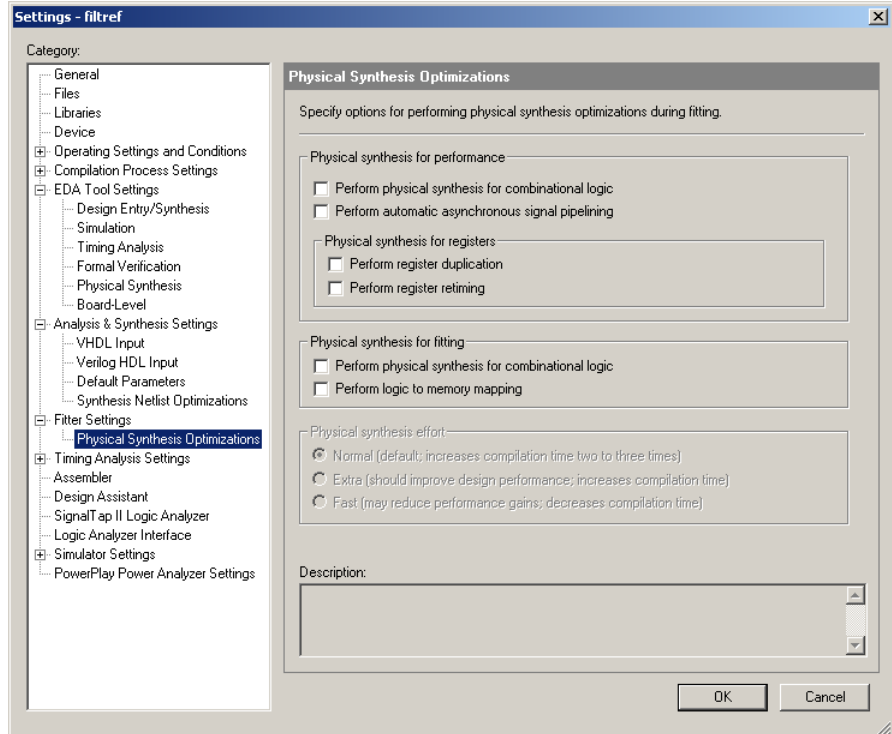


7. In the **Category** list, click the "+" icon to expand **Fitter Settings**, and select **Physical Synthesis Optimizations**. The **Physical Synthesis Optimizations** page appears.

   a. Under **Physical synthesis for registers**, turn off **Perform register retiming**.

   b. Under **Physical Synthesis for Fitting**, turn off both **Perform physical synthesis for combinational logic** and **Perform logic to memory mapping** to prevent logic from being mapped to RAMs (Figure 18–6).

Figure 18–6. Fitter Settings



Retiming a design, either during the synthesis step or during the fitting step, usually results in moving and merging registers along the critical path and is not well-supported by the equivalence checking tools. Because equivalence checkers compare the cone of logic terminating at registers, do not use retiming to move the registers during optimization in the Quartus II software.

If the **Perform gate-level register retiming** (Figure 18–5) and **Perform register retiming** options (Figure 18–6) are not turned off, the Encounter Conformal script displays a different set of compare points, making the resulting netlist difficult to compare against the reference netlist file. If you use retiming in your design during compilation, you cannot generate a netlist for formal verification.

For more information about physical synthesis, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

8. Perform a full compilation of the design. On the Processing menu, click **Start Compilation**, or click the **Start Compilation** icon on the Toolbar.

The Quartus II Software Generated Files, Formal Verification Scripts, and Directories

After successful compilation, the Quartus II software generates a list of files, formal verification scripts, and directories in the <*project_directory*>**/fv/conformal/** directory (Table 18–4).

Table 18–4. The Quartus II Software Compiler-Generated Files and Directories

File or Directory	Name	Details
.vo file	<*proj rev*>**.vo**	The Quartus II software-generated netlist for formal verification.
Script file	<*proj rev*>**.ctc**	The <*proj rev*>**.ctc** file references <*proj rev*>**.clg** and <*proj rev*>**.clr** that read the library files and black box descriptions. The <*proj rev*>**.ctc** file also references the <*proj rev*>**.cmc** file containing information about the mapped points. *(1)*
	<*proj rev*>**.cec**	The <*proj rev*>**.cec** file contains the information for instance equivalences.
	<*proj rev*>**.cep**	The <*proj rev*>**.cep** file contains the information for black box pin equivalences in the design.
	<*proj rev*>**.cmp**	The <*proj rev*>**.cmp** file contains the information for the black box pin mapping between the golden and revised sides. *(2)*
	<*proj rev*>**.cmc**	The <*proj rev*>**.cmc** file contains information about the additional points to be mapped in addition to the points selected by the tool.
	<*proj rev*>**_trivial.cmc**	This <*proj rev*>**_trivial.cmc** file contains mapping information for all the key points in the design. *(3)*
	<*proj rev*>**.clr**	The <*proj rev*>**.clr** file contains information about the macros and libraries for the revised design.
	<*proj rev*>**.clg**	The <*proj rev*>**.clg** file contains information about the macros and libraries for the golden design.
blackboxes directory	<*project directory*>**/fv/conformal/**<*project rev*>**_blackboxes**	This directory contains top-level module descriptions for all the user-defined black box entities and contains modules with definitions other than Verilog HDL or VHDL, for example, Block Design File (**.bdf**) in the design directory <*project directory*>**/fv/conformal/**<*project rev*>**_blackboxes**

Notes to Table 18–4:

(1) This file is used with the Encounter Conformal software.

(2) This file is called from the <*proj rev*>**.ctc** script file. By default, the line where this file is called is commented out. These files are only useful for HardCopy II device families.

(3) In some cases, Encounter Conformal software performs incorrect key point mapping, resulting in formal verification mismatches. To overcome the verification mismatches, the Quartus II software writes out the <*proj rev*>**_trivial.cmc** file that contains mapping information for all the key points in the design. Reading this file during the formal verification setup can result in increased run time. Therefore, the Quartus II software writes out the top-level script file <*proj rev*>**.ctc** with the command to read the <*proj rev*>**_trivial.cmc** file commented out. If the formal verification results are not acceptable, the user can uncomment the command and read the <*proj rev*>**_trivial.cmc** file. The command in the <*proj rev*>**.ctc** file is:

```
//Trivial mappings with same name registers
//read mapped points $PROJECT/fv/conformal/<proj rev>_trivial.cmc
```

The script file contains the setup and constraints information to use with the formal verification tool. The file <*entity*>**.v** in the **blackboxes** directory contains the module description of entities that are not defined in the formal verification library. The file also contains entities that you specify as black boxes. For example, if there is a reference to a black box for an instance of the ALTDPRAM megafunction in the design, the **blackboxes** directory does not contain a module description for the ALTDPRAM megafunction because it is defined in the **altdpram.v** file of the formal verification library. When a module does not have an RTL description, or the

description exists only in the formal verification library and you do not want to compare the module using formal verification, a file containing only the top-level module description with port declaration is written out to the **blackboxes** directory and read into the Encounter Conformal software. To learn more about black boxes, refer to "Black Boxes in the Encounter Conformal Flow" on page 18–8.

Understanding the Formal Verification Scripts for Encounter Conformal

The Quartus II software generates scripts to use with the Encounter Conformal Logic Equivalence Check (LEC) software. This section elaborates on the details of the Encounter Conformal commands used within the scripts to help you compare the revised netlist with the golden netlist. In most cases, you do not have to add any more Encounter Conformal constraints to verify your netlists.

A sample script generated by the Quartus II software is provided in "Conformal Dofile/Script Example" on page 18–23.

The Encounter Conformal Commands within the Quartus II Software-Generated Scripts

The value for the variable `QUARTUS` is the path to the Quartus II software installation directory:

```
setenv QUARTUS <Quartus Installation Directory>
```

The Quartus II software assigns the current working directory of the project to the `PROJECT` variable. Use this variable to change the project directory to the directory where the design files are installed when moving from a UNIX to a Windows environment, or vice versa:

```
setenv PROJECT <Quartus Project Directory>
```

The following command reads both the golden and the revised netlists, along with the appropriate library models:

```
read design <design files>
```

You must update the project location when the files are moved from the Windows environment to the UNIX environment.

The post place-and-route netlist from the Quartus II software might contain net and instance names that are slightly different from those of the golden netlist. By using the following command, the Quartus II software defines temporary substitute string patterns enabling the Encounter Conformal software to automatically map key points when the names are not the same:

```
add renaming rule <rule>
```

The Encounter Conformal LEC software employs three name-based methods to map key points to compare the revised netlist with the golden netlist. Scripts set the correct method to get the best results.

```
set mapping method <mapping_rule>
```

The Quartus II software performs several optimizations, including optimizing the registers whose input is driven by a constant. Under these circumstances, for the formal verification software to compare the netlists properly, the command `set flatten model` is used with the option `seq_constant`.

```
set flatten model <flattening_rule>
```

When you use the command `report black box`, verify that the following modules are listed as black boxes, along with any of the modules black boxed by the user, in both the golden and revised netlists:

- LPMs and megafunctions without the formal verification models
- Encrypted IP functions
- Entities not implemented in Verilog HDL or VHDL

Use the following command to set the same implementation on multipliers for both the golden and revised netlists:

```
set multiplier implementation <implementation_name>
```

If there are any combinational loops or instances of `LPM_LATCH`, the Quartus II software cuts the loop at the same point using the following command on both the golden and revised netlists:

```
add cut point
```

The Encounter Conformal software does not always automatically map all of the keypoints, or can incorrectly map some keypoints. To help the Encounter Conformal software successfully complete the mapping process, the Quartus II software records optimizations performed on the netlist as a series of `add mapped points` in the Encounter Conformal <*file_name*>**.cmc** script.

```
add mapped points <key_points>
```

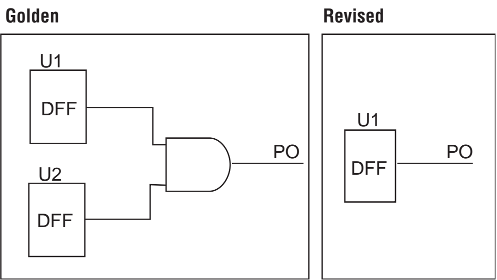
There are situations in which the inverter in front of the register is moved after the register. In this situation, the following command is used:

```
add mapped points <key_points> -invert
```

The following command reads in the mapped point information from the specified file:

```
read mapped points <file_name>.cmc
```

Figure 18–7. Instance Equivalence



During the process of optimization, the Quartus II software might merge two registers into one (Figure 18–7). The Quartus II software informs the formal verification tool that the U1 and U2 registers are equivalent to each other using the following command:

```
add instance equivalence <instance_pathname ..> [-Golden]
```

If the register duplication takes place, the following command is used:

```
add instance equivalence <instance_pathname ..> [-revised]
```

The following command is used when the inverter is moved beyond the register along with either register duplication or merging:

```
add instance equivalences <instance_pathname>
[-invert <instance_pathname>]
```

At times, the register output is driven to a constant, either `logic 0` or `logic 1`. The Quartus II software sets the value of the register to a constraint using the `add instance constraint` command. For more information about this command, refer to "Stuck-at Registers" on page 18–6.

```
add instance constraint <constraint_value>
```

Comparing Designs Using Encounter Conformal

This section addresses using the Encounter Conformal software to compare designs; that is, how to prove logical equivalence between two versions of the design.

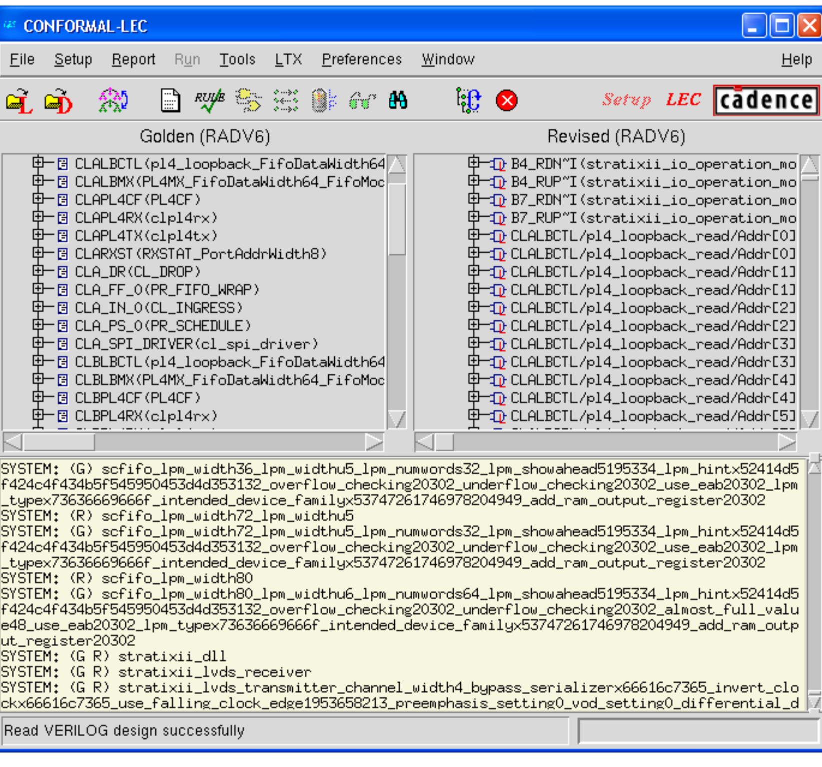
Running the Encounter Conformal Software from the GUI

To run the Encounter Conformal software from the GUI, follow these steps:

1. Open the Encounter Conformal software.
2. On the File menu, click **Do Dofile**.
3. Select the file <*path to project directory*>**/fv/conformal/**<*proj rev*>**.ctc**.

The Encounter Conformal software GUI displays the comparison results (Figure 18–8). The Golden window displays the original RTL description or the post synthesis **.vqm** netlist file from Synplify Pro, and the Revised window displays the information of the post-fit netlist generated by the Quartus II software. The message section at the bottom of the window reports the verification results and the number of unmapped and non-equivalent points found in the design.

Figure 18–8. Encounter Conformal Software GUI Display of Functional Comparisons



To investigate the verification results, click the **Mapping Manager** icon in the toolbar, or on the Tools menu, click **Mapping Manager**. The Encounter Conformal software reports the mapped, unmapped, and compared points in the **Mapped Points**, **Unmapped Points**, and **Compared Points** windows, respectively.

For more information about how to diagnose non-equivalent points, refer to the Encounter Conformal software user documentation.

Running the Encounter Conformal Software From a System Command Prompt

To run the Encounter Conformal Software without using the GUI, type the command shown in Example 18–6 at a system command prompt.

Example 18–6. Conformal LEC Command to Run Formal Verification

```
lec -dofile /<path to project directory>/fv/conformal/<proj rev>.ctc -nogui
```

To get a downloadable design example showing the formal verification flow with Quartus II software, go to www.altera.com/support/examples/quartus/exm-formal-verification.html.

For more information about the latest debugging tips and solutions for formal verification flow between Cadence Conformal LEC tool and Quartus II software, go to www.altera.com and perform an advanced search with keywords “formal verification”.

Known Issues and Limitations

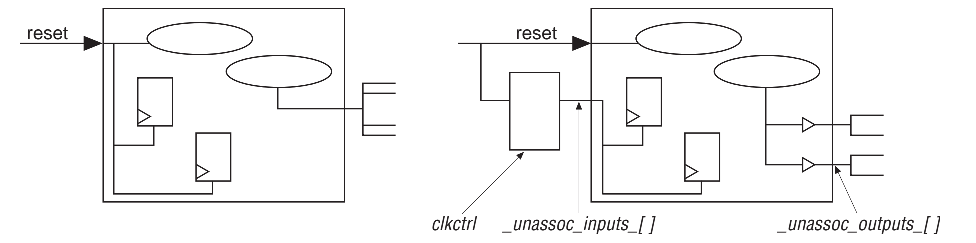
The following known issues and limitations can occur when using the formal verification flow described in this chapter:

- When a port on a black box entity drives two or more signals within the black box, the Quartus II software pushes the connections outside of the black box, and creates that many ports on the black box. This problem is only associated with Stratix II and HardCopy II designs.

  The additional ports on the black box are named `_unassoc_inputs_[]` and `_unassoc_outputs_[]` (Figure 18–9). This issue is generally associated with reset and enable signals. Figure 18–9 shows an example in which the reset pin is split into two ports outside of the black box and the `_unassoc_inputs_[]` port is driven by the `clkctrl` block. In such situations, the **.vo** netlist file generated by the Quartus II software has signals driving these black box ports, but golden RTL does not contain any signals to drive the `_unassoc_inputs_[]` port, resulting in a formal verification mismatch of the black box. The black box module definition generated by the Quartus II software in the directory <*Quartus_project*>**\fv\conformal*_blackboxes** contains these additional `_unassoc_inputs_[]` and `_unassoc_outputs_[]` ports. This black box module is read on both the golden and revised sides of the design, which results in unconnected ports on the golden side and formal verification mismatches.

  Figure 18–9 shows the creation of the `_unassoc_inputs_[]` and `_unassoc_outputs_[]` ports for the reset signal.

Figure 18–9. Creation of _unassoc_inputs_[] and _unassoc_outputs_[]

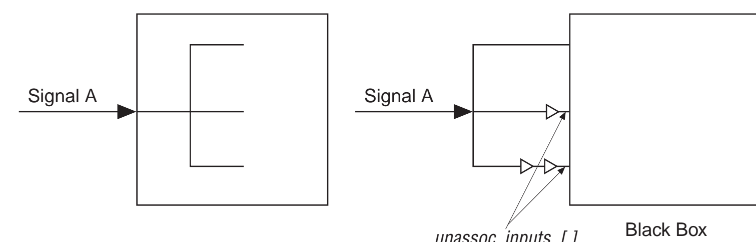


Another common occurrence of this issue is in HardCopy II designs. Whenever a port drives large fan-out within the black box, the Quartus II software inserts a buffer on the net and moves the logic outside of the black box (Figure 18–10).

To fix the problem of `_unassoc_input_[]` ports causing black box mismatches, use Cadence Conformal commands to change the type of the black box `_unassoc_input_[]` keypoint to a primary output keypoint, and then marking the appropriate pin equivalences. Similarly, to fix the problem of register mismatches due to `_unassoc_output_[]` pins from black boxes, use Conformal commands to change the type of the blackbox `_unassoc_output_[]` keypoint to a primary input, and then marking equivalent pins as such. The commands to perform these actions are written in the <*proj rev*>**.cep** file.

Figure 18–10 shows the creation of `_unassoc_inputs_[]` for a signal with large fan-out.

Figure 18–10. Creation of _unassoc_inputs_[] for a Signal with Large Fan-out



- In designs with combinational feedback loops, the Encounter Conformal software can insert extra cut points in the revised netlist, causing unmapped points and ultimately verification mismatches.
- For Cyclone II designs, Conformal LEC might report non-equivalent flipflops and extra cut points for the revised (post-fit) design when your HDL source code instantiates the `lpm_ff` primitive with an asynchronous load signal `aload` (with or without any other asynchronous control signals) and when the asynchronous clear signal `aclr` and asynchronous set signal `aset` are used together. To avoid this problem, ensure that there is a wrapper module or entity around the `lpm_ff` instantiation, and black box the module or entity that instantiates the `lpm_ff` primitive.
- For Stratix III designs, the Cadence Conformal LEC software creates cut points for the combinational loops on the golden side and might fail equivalence checking due to improper mapping. The combinational loops are due to logic around the registers emulating multiple set, resets, or both. These cut points are also reported during the mapping step in Quartus II software with warning messages. You can add Cadence Conformal commands manually to add cut points, which can result in proper mapping and formal verification.
- To perform formal verification, certain synthesis optimization options (such as register retiming, optimization through black box hierarchy boundaries, and disabling the ROM and shift register inference) are turned off, which can have an impact on the area resource and performance.
- RAM and ROM instantiations, inferences, or both are not verified using formal verification.
- Incremental Compilation for the purpose of formal verification does not support user-created design partitions.
- Formal verification does not support clear box netlists due to unconnected ports on its WYSIWYG instances.
- Formal verification does not support VHDL megafunction variations due to undriven ports on the megafunctions.
- When a black box contains bidirectional ports, the Quartus II software fails to reconstruct the hierarchy. For this reason, the black box is represented by a flat netlist, resulting in formal verification mismatches.
- ROMs in the design must be black boxed before compilation using Quartus II integrated synthesis, because the Quartus II software might perform some optimizations on the ROM, resulting in Formal Verification mismatches.

- The Conformal software might report mismatches or abort comparison of some key points when a DSP megafunction is implemented in LEs by the Quartus II software, due to implicit optimizations within the DSP and the complexity of the multiplier logic in terms of LEs.
- Unused logic optimized within and around a black box by the Quartus II software can result in a black-box interface different from the interface in the synthesized **.vqm** netlist file.

Black Box Models

The black box models are interface definitions of entities, such as primitives, atoms, LPMs, and megafunctions. These models have a parameterized interface, and do not contain any definition of behavior. They are designed and tested specifically to work with the Encounter Conformal software, which uses these models along with your design to generate black boxes for instances of the entity with varying sets of parameters in the design. Table 18–5 describes the supported black box models. Besides these black box models, you can set a black box property on a specific module or entity as explained earlier in this chapter.

Table 18–5. Supported Black Box Models (Part 1 of 3)

Entity Type	Entity Names
Megafunctions	ALT3PRAM, ALTACCUMULATE, ALTFP_MULT, ALTSQRT, ALTLVDS_RX, ALTLVDS_TX, ALTSHIFT_TAPS, SLD_VIRTUAL_JTAG, SLD_VIRTUAL_JTAG_BASIC, DCFIFO, SCFIFO, ALTSYNCRAM, ALTSQRT
LPMs	lpm_add_sub, lpm_divide

Table 18–5. Supported Black Box Models (Part 2 of 3)

Entity Type	Entity Names
Atoms *(1)*	Cyclone: cyclone_crcblock, cyclone_jtag, cyclone_pll, cyclone_ram_block, cyclone_asmiblock, cyclone_dll
	Stratix: stratix_crcblock, stratix_jtag, stratix_lvds_receiver, stratix_lvds_transmitter, stratix_pll, stratix_rublock, stratix_ram_block, stratix_dll
	Stratix II: stratixii_crcblock, stratixii_jtag, stratixii_lvds_receiver, stratixii_lvds_transmitter, stratixii_pll, stratixii_rublock, stratixii_ram_block, stratixii_asm_block, stratixii_dll, stratixii_termination, stratixii_asmiblock
	Stratix GX: stratixgx_crcblock, stratixgx_jtag, stratixgx_lvds_receiver, stratixgx_lvds_transmitter, stratixgx_pll, stratixgx_rublock, stratixgx_ram_block, stratixgx_dll
	Stratix II GX: stratixiigx_hssi_receiver, stratixiigx_hssi_transmitter, stratixiigx_hssi_central_management_unit, stratixiigx_hssi_cmu_pll, stratixiigx_hssi_cmu_clock_divider, stratixiigx_hssi_refclk_divider, stratixiigx_hssi_calibration_block, stratixiigx_crcblock, stratixiigx_ram_block, stratixiigx_lvds_transmitter, stratixiigx_lvds_receiver, stratixiigx_pll, stratixiigx_dll, stratixiigx_jtag, stratixiigx_asmiblock, stratixiigx_termination, stratixiigx_rublock
	Cyclone II: cycloneii_asmiblock, cycloneii_clk_delay_ctrl, cycloneii_clkctrl, cycloneii_jtag, cycloneii_pll, cycloneii_ram_block
	Arria GX: arriagx_asmiblock, arriagx_crcblock, arriagx_dll, arriagx_hssi_calibration_block, arriagx_hssi_central_management_unit, arriagx_hssi_cmu_clock_divider, arriagx_hssi_cmu_pll, arriagx_hssi_receiver, arriagx_hssi_refclk_divider, arriagx_hssi_transmitter, arriagx_jtag, arriagx_lvds_receiver, arriagx_lvds_transmitter, arriagx_pll, arriagx_ram_block, arriagx_rublock, arriagx_termination
	HardCopy II: hardcopyii_crcblock, hardcopyii_dll, hardcopyii_jtag, hardcopyii_lvds_receiver, hardcopyii_lvds_transmitter, hardcopyii_pll, hardcopyii_ram_block, hardcopyii_termination

Table 18–5. Supported Black Box Models (Part 3 of 3)

Entity Type	Entity Names
Atoms *(1)*	Stratix III: `stratixiii_asmiblock`, `stratixiii_crcblock`, `stratixiii_jtag`, `stratixiii_lvds_receiver`, `stratixiii_lvds_transmitter`, `stratixiii_mlab_cell`, `stratixiii_pll`, `stratixiii_ram_block`, `stratixiii_rublock`, `stratixiii_termination`, `stratixiii_tsdblock`
	Cyclone III: `cycloneiii_apfcontroller`, `cycloneiii_clkctrl`, `cycloneiii_crcblock`, `cycloneiii_ddio_oe`, `cycloneiii_ddio_out`, `cycloneiii_ff`, `cycloneiii_io_ibuf`, `cycloneiii_io_obuf`, `cycloneiii_io_pad`, `cycloneiii_jtag`, `cycloneiii_lcell_comb`, `cycloneiii_mac_mult`, `cycloneiii_mac_out`, `cycloneiii_oscillator`, `cycloneiii_pll`, `cycloneiii_pseudo_diff_out`, `cycloneiii_ram_block`, `cycloneiii_rublock`, `cycloneiii_termination`
	Stratix IV: `stratixiv_asmiblock`, `stratixiv_bias_block`, `stratixiv_crcblock`, `stratixiv_jtag`, `stratixiv_lvds_receiver`, `stratixiv_lvds_transmitter`, `stratixiv_mlab_cell`, `stratixiv_pll`, `stratixiv_ram_block`, `stratixiv_rublock`, `stratixiv_termination`, `stratixiv_tsdblock`

Note to Table 18–5:

(1) The entity names are given for the specific device family listed.

Conformal Dofile/Script Example

The following example script (Example 18–7), generated by the Quartus II software, lists some of the setup commands used in Conformal LEC software:

Example 18–7. Conformal LEC Script (Part 1 of 3)

```
// Copyright (C) 1991-2008 Altera Corporation
// Your use of Altera Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logi
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Altera Program License
// Subscription Agreement, Altera MegaCore Function License
// Agreement, or other applicable license agreement, including,
// without limitation, that your use is for the sole purpose of
// programming logic devices manufactured by Altera and sold by
// Altera or its authorized distributors. Please refer to the
// applicable agreement for further details.

// Script generated by the Quartus II software

reset
set system mode setup
set log file mfs_3prm_1a.fv.log -replace
set naming rule "%s" -register -golden
set naming rule "%s" -register -revised
// Naming rules for Verilog
set naming rule "%L.%s" "%L[%d].%s" "%s" -instance
set naming rule "%L.%s" "%L[%d].%s" "%s" -variable
```

Example 18–7. Conformal LEC Script (Part 2 of 3)

```
// Naming rules for VHDL
// set naming rule "%L:%s" "%L:%d:%s" "%s" -instance
// set naming rule "%L:%s" "%L:%d:%s" "%s" -variable
// set undefined cell black_box -both
// These are the directives that are not supported by the QIS RTL to gates FV flow
set directive off verplex ambit
set directive off assertion_library black_box clock_hold compile_off compile_on
set directive off dc_script_begin dc_script_end divider enum infer_latch
set directive off mem_rowselect multi_port multiplier operand state_vector template
add notranslate module alt3pram -golden
add notranslate module alt3pram -revised
setenv QUARTUS /data/quark/build/ajaishan/quartus
setenv PROJECT /net/quark/build/ajaishan/quartus_regtest/eda/fv/conformal/synplify/
stratix/mfs_3prm_1a_v1_/mfs_3prm_1a/qu_allopt
read design \
   $QUARTUS/eda/fv_lib/vhdl/dummy.vhd \
   -map lpm $QUARTUS/eda/fv_lib/vhdl/lpms \
   -map altera_mf $QUARTUS/eda/fv_lib/vhdl/mfs \
   -map stratix $QUARTUS/eda/fv_lib/vhdl/stratix \
   -vhdl -noelaborate -golden
read design \
   -file $PROJECT/fv/conformal/mfs_3prm_1a.clg \
   $PROJECT/p3rm_block.v \
   $PROJECT/mfs_3prm_1a.v \
   -verilog2k -merge none -golden
read design \
   $QUARTUS/eda/fv_lib/vhdl/dummy.vhd \
   -map lpm $QUARTUS/eda/fv_lib/vhdl/lpms \
   -map altera_mf $QUARTUS/eda/fv_lib/vhdl/mfs \
   -map stratix $QUARTUS/eda/fv_lib/vhdl/stratix \
   -vhdl -noelaborate -revised
read design \
   -file $PROJECT/fv/conformal/mfs_3prm_1a.clr \
   $PROJECT/fv/conformal/mfs_3prm_1a.vo \
   -verilog2k -merge none -revised
// add ignored inputs _unassoc_inputs_* -all -revised
add renaming rule r1 "~I\/" "\/" -revised
add renaming rule r2 "_I\/" "\/" -revised
set multiplier implementation rca -golden
set multiplier implementation rca -revised
set mapping method -name first
set mapping method -nounreach
set mapping method -noreport_unreach
set mapping method -nobbox_name_match
set flatten model -seq_constant
set flatten model -nodff_to_dlat_zero
set flatten model -nodff_to_dlat_feedback
set flatten model -nooutput_z
set root module mfs_3prm_1a -golden
set root module mfs_3prm_1a -revised
report messages
report black box
report design data
// report floating signals
dofile $PROJECT/fv/conformal/mfs_3prm_1a.cec
// dofile $PROJECT/fv/conformal/mfs_3prm_1a.cep
// Instance-constraints commands for constant-value registers removed
// during compilation
set system mode lec -nomap
read mapped points $PROJECT/fv/conformal/mfs_3prm_1a.cmc
```

Example 18–7. Conformal LEC Script (Part 3 of 3)

```
// Trivial mappings with same name registers
// read mapped points $PROJECT/fv/conformal/mfs_3prm_1a_trivial.cmc
// dofile $PROJECT/fv/conformal/mfs_3prm_1a.cmp
map key points
remodel -seq_constant -repeat
add compare points -all
compare
usage
// exit -f
```

Conclusion

Formal verification software enables verification of the design during all stages from RTL to placement and routing. Verifying designs takes more time as designs increase in size. Formal verification is a technique that helps reduce the time needed for your design verification cycle.

Referenced Documents

This chapter references the following documents:

- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 18–6 shows the revision history for this chapter.

Table 18–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Updated Table 18–1 on page 18–4.	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	■ Changed to 8-1/2 x 11 page size. ■ Added support for Stratix IV devices. ■ Added support for Cadence Conformal LEC version 7.2 and Synplify Pro version 9.6.2.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Added support for Cyclone III devices. ■ Updated “Black Boxes in the Encounter Conformal Flow” on page 18–8. ■ Updated Table 18–1 and Table 18–5.	Updated for the Quartus II software version 8.0.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

Section VI. Device Programming

The Quartus® II software offers a complete software solution for system designers who design with Altera® FPGA and CPLD devices. The Quartus II Programmer is part of the Quartus II software package that allows you to program Altera CPLD and configuration devices, and configure Altera FPGA devices. This section describes how you can use the Quartus II Programmer to program or configure your device after you successfully compile your design.

This section includes the following chapter:

- Chapter 19, Quartus II Programmer

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

19. Quartus II Programmer

QII53022-9.0.0

Introduction

The Quartus® II software offers a complete software solution for system designers who design with Altera® FPGA and CPLD devices. The Quartus II Programmer is part of the Quartus II software package that allows you to program Altera CPLD and configuration devices and configure Altera FPGA devices. After your design successfully compiles, you can use the Quartus II Programmer to program or configure your device.

This chapter contains the following sections:

- "Programming Flow"
- "Programming and Configuration Modes" on page 19–4
- "Programmer Overview" on page 19–6
- "Hardware Setup" on page 19–11
- "Device Programming and Configuration" on page 19–12
- "Optional Programming Files" on page 19–17
- "Flash Loaders" on page 19–19
- "JTAG Chain Debugger Tool" on page 19–20
- "Other Programming Tools" on page 19–30
- "Scripting Support" on page 19–30

Programming Flow

The programming flow begins with design compilation, in which the Quartus II Assembler generates the programming or configuration file, then proceeds with the programming or configuration file conversion for different configuration devices, or optional programming and configuration file creation. The flow ends with the configuration or programming of the FPGA, CPLD, or configuration devices with the programming or configuration file using the Quartus II Programmer.

Figure 19–1 shows the programming file generation flow. This flow covers the types of configuration and programming files that are used by the Quartus II Programmer. Refer to "Optional Programming Files" on page 19–17 for information on other optional programming files.

Figure 19–1. Programming File Generation Flow

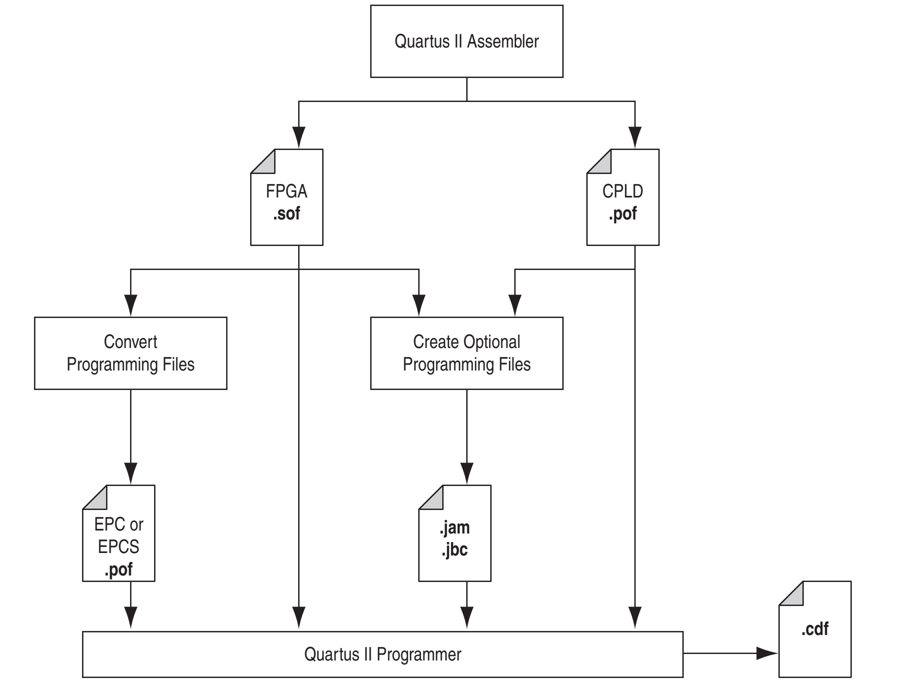


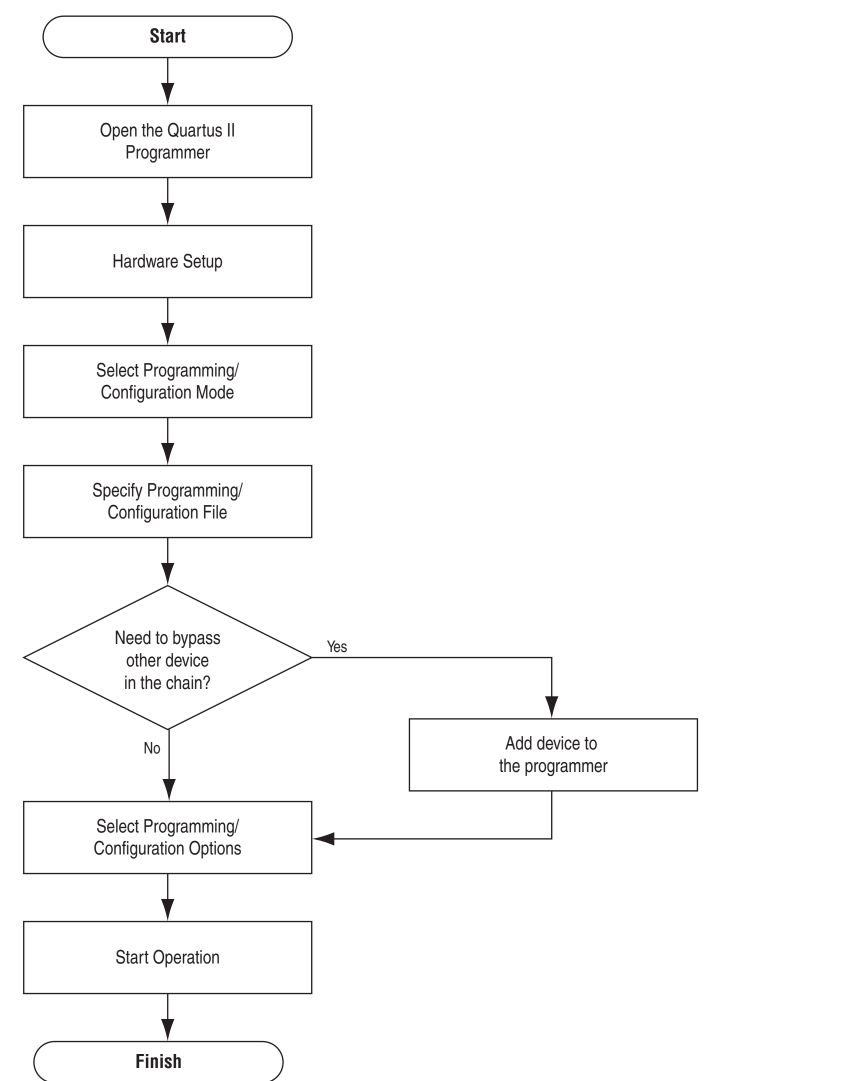
Table 19–1 shows the programming and configuration file formats supported by Altera FPGAs, CPLDs, configuration devices, enhanced configuration devices, and serial configuration devices. Chain description files (**.cdf**) are described in "Chain Description File" on page 19–16.

Table 19–1. Programming and Configuration File Format

File Format	FPGA	CPLD	Configuration Device and Enhanced Configuration Device	Serial Configuration Device
SRAM Object File (**.sof**)	✓	—	—	—
Programmer Object File (**.pof**)	—	✓	✓	✓
Jam File (**.jam**)	✓	✓	✓	—
Jam Byte-Code File (**.jbc**)	✓	✓	✓	—

Figure 19–2 shows the programming flow using the Quartus II Programmer. Refer to "Generating Optional Programming Files" on page 19–19 for detailed information about converting or creating different programming files. Refer to "Device Programming and Configuration" on page 19–12 for information about programming or configuring the device.

Figure 19–2. Programming Flow



Programming and Configuration Modes

The Quartus II Programmer supports the following four programming or configuration modes: JTAG, passive serial, active serial, and in-socket programming.

JTAG Mode

You can use the Joint Test Action Group (JTAG) mode to configure FPGA devices and program CPLDs, configuration devices, or enhanced configuration devices. The JTAG mode allows multiple FPGAs, CPLDs, and configuration devices connected in a JTAG chain to be configured or programmed at the same time. JTAG programming or configuration uses four JTAG pins: TCK, TDI, TMS, and TDO. The JTAG interface also allows you to perform boundary-scan test using third-party boundary scan tools.

.pof files are used for programming CPLDs, and configuration or enhanced configuration devices, while **.sof** files are used for configuring FPGA devices. **.jam** and **.jbc** files are used for both programming and configuration. Serial configuration devices do not support JTAG programming.

For more information about JTAG configuration or programming mode and JTAG pin connections, refer to the *Configuration Handbook*, or the device handbook or data sheet for the respective FPGA, CPLD, or configuration device.

Passive Serial Mode

You can use the passive serial (PS) mode to configure Altera FPGAs. PS configuration uses the DCLK, CONF_DONE, nCONFIG, nSTATUS, and DATA0 configuration pins. Unlike the JTAG scheme, the PS configuration scheme is used to configure the FPGA with a configuration device or enhanced configuration device, not necessarily through a download cable. If you use the configuration device or enhanced configuration device to configure the FPGA through PS mode, you can route the configuration signals out to a header so that you can also configure the FPGA through the download cable with the Quartus II Programmer. Configuration through PS mode with a download cable is useful in the design stage. This configuration method allows you to configure your FPGA device directly from the Quartus II Programmer as you make changes to your design for debugging and testing.

PS mode supports configuration of an FPGA chain. **.sof** files are used for configuration through PS mode. Every FPGA device in the chain requires an **.sof** file, so the number of **.sof** files used depends on the number of FPGA devices in the chain.

For more information about PS configuration mode and PS pin connection, refer to the *Configuration Handbook* or the chapter on configuration in the appropriate FPGA device handbook.

© March 2009 Altera Corporation

Active Serial Mode

You can use the active serial (AS) mode to program serial configuration devices. After programming is completed, the serial configuration device configures the FPGA. AS programming uses the DATA, DCLK, nCS, and ASDI pins. If the download cable is connected to the nCONFIG and nCE pins of the FPGA, the download cable disables the FPGA's access to the AS interface by holding the nCE pin high and the nCONFIG pin low. Upon completion of the programming, the nCE and nCONFIG pins are released and the FPGA configuration begins.

For more information about programming the serial configuration device, configuring the FPGA with the serial configuration device through AS mode, or the AS pin connections, refer to the *Serial Configuration Data Sheet* in the *Configuration Handbook* or the chapter on configuration in the appropriate FPGA device handbook.

In-Socket Programming Mode

The in-socket programming mode is used for programming a single device. This programming mode supports programming the MAX® 7000 and MAX 3000 CPLD families, configuration devices, enhanced configuration devices, and serial configuration devices. Instead of using a download cable, in-socket programming mode uses the Altera Programming Unit (APU) hardware together with the programming adapter for the corresponding device to program the device. The programming unit with the programming adapter has a socket for the device and the hardware powers the device for programming. In-socket programming is normally used in the production environment to pre-program devices before they are mounted on the printed circuit boards on the assembly line.

Refer to www.altera.com or the Quartus II Help for a list of programming adapters available for Altera devices.

Table 19–2 shows the programming and configuration modes supported by Altera devices.

Table 19–2. Programming and Configuration Modes

Mode	FPGA	CPLD	Configuration Device and Enhanced Configuration Device	Serial Configuration Device
JTAG	✓	✓	✓	—
PS	✓	—	—	—
AS	—	—	—	✓
In-Socket Programming	—	✓ *(1)*	✓	✓

Note to Table 19–2:

(1) MAX II CPLDs do not support in-socket programming mode.

Programmer Overview

The Quartus II Programmer GUI is a window in which you can add your programming and configuration files, specify the programming options and hardware, and then proceed with the programming or configuration of the device.

To open the Programmer window, on the Tools menu, click **Programmer**. Figure 19–3 shows the programmer GUI. The status of each operation, whether programming is successful or not, is reported in the Quartus II message window. Figure 19–4 shows a typical programming message after the programmer has successfully programmed a device.

Figure 19–3. The Programmer Window

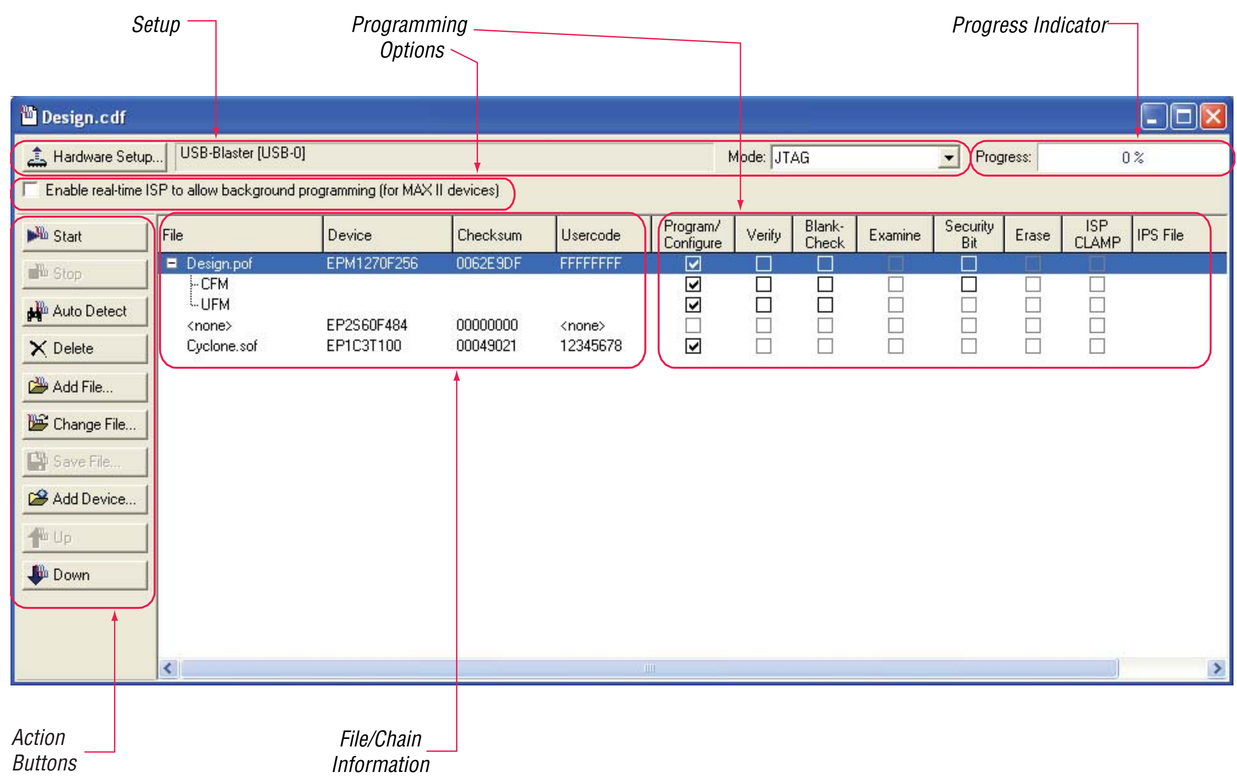
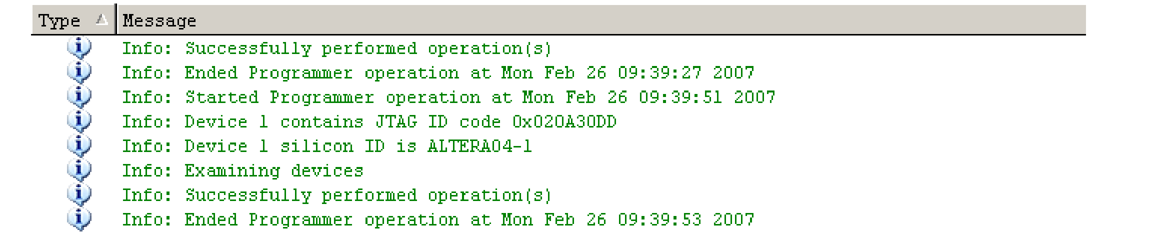


Figure 19–4. Status Report in the Message Window



© March 2009 Altera Corporation

Table 19–3 describes the items available in the programmer window.

Table 19–3. Programmer Window Items (Part 1 of 2)

Items	Description
Hardware Setup	Opens the **Hardware Setup** dialog box in the programmer and enables you to perform the following functions: ■ Add and remove hardware items from the **Hardware** list ■ Add and remove JTAG servers from the **JTAG Servers** list ■ Configure your local JTAG server ■ Specify a programming hardware or download cable for device programming and configuration
Mode	Specifies the programming or configuration mode (either JTAG, In-Socket Programming, Passive Serial, or Active Serial Programming).
Progress	Shows the progress of a specific operation.
Action Buttons	
Start	Starts the operations of the specified programming options.
Stop	Stops all operations in progress.
Auto Detect	Scans the JTAG chain to check for devices in the chain and the chain connection.
Delete	Removes the selected programming or configuration files from the programmer.
Add File	Adds programming or configuration files to the programmer.
Change File	Replaces the selected programming or configuration file with another file.
Save File	Allows you to save the data read out from CPLD or configuration devices using the “examine” process into a **.pof** file.
Add Device	Adds a device into the JTAG device chain in the programmer. If no programming or configuration file is specified, the programmer bypasses this device during programming or configuration. You can also add your user-defined device into the chain.
Up	Moves the selected programming/configuration file or device up in the programmer window.
Down	Moves the selected programming/configuration file or device down in the programmer window.
File or Device Chain Information	
File	Displays the programming or configuration file name.
Device	The **Device** column shows the following items: ■ The target device of the file, if you add a programming or configuration file into the programmer ■ The devices in the JTAG chain detected by the programmer, if you click **Auto Detect** in JTAG mode ■ The device added to the programmer, if you manually add a device into the programmer
Checksum	The **Checksum** column shows the following items: ■ The checksum of the file, if you add a programming or configuration file into the programmer ■ The checksum for the data read out, if you examine a device The checksum is calculated by the Quartus II software. The programmer does not display the checksum for the **.jam** or **.jbc** files generated for a multi-device JTAG chain.

Table 19–3. Programmer Window Items (Part 2 of 2)

Items	Description
Usercode	The **Usercode** column shows the following items: ■ The usercode of the file, if you add a programming or configuration file into the programmer ■ The usercode read out from the device, if you examine a device You can specify the usercode before design compilation, or use the Auto usercode feature that uses the checksum as the usercode. The programmer does not show the usercode information in PS configuration mode or for the Jam or **.jbc** files generated for a multi-device JTAG chain.
Programming Options	
Enable real-time ISP to allow background programming	Can only be turned on if you are targeting a MAX II device, and is turned off for all other device families. When this option is turned on, you can do the real-time in-system programming (ISP) for the MAX II device. The existing design in the MAX II device functions normally during and after the real-time ISP is completed. The new design starts to function after a power cycle to the device occurs.
Program or Configure	Can be used for programming CPLDs, configuration devices, or configuring FPGA devices.
Verify	Verifies the content of the CPLD and all configuration devices against the respective programming files. This option is not available for FPGAs. Verification fails if the data in the file is different from the data in the device. Stand-alone verification for the CPLD with the programming file used for the programming fails if the security bit is set when the device is programmed initially.
Blank-Check	Checks whether the CPLD or configuration device is blank.
Examine	Reads back the contents of the CPLD or configuration device. You can then save the examined data as a **.pof** file. Examining a CPLD with the security bit set does not produce a usable **.pof** file. This option is not available for MAX 7000S devices.
Security Bit	Protects the design in the CPLD from being examined. If the security bit is set when the CPLD is programmed, you cannot read the correct data out using the examine process. Security bits cannot be set for the configuration devices or FPGAs.
Erase	Erases the contents of the CPLD and all configuration devices. You can also erase the user flash memory (UFM) of the MAX II CPLD. This option is not available for MAX 7000S devices.
ISP Clamp	Allows the MAX II or MAX 7000B CPLD's I/O pins to be clamped to certain states during normal programming. ISP Clamp can only be turned on if certain pins of the device have the ISP Clamp State assignment enabled, or you have added an I/O Pin State (**.ips**) file in the programmer.
IPS File	Shows the IPS file used for ISP Clamp of the MAX II or MAX 7000B CPLDs. The **IPS File** column only appears if your programmer window has a MAX II or MAX 7000B **.pof** file. To add the **.ips** file, click once on the row of the programming file and on the Edit menu, click **Add IPS File**.

Table 19–4 shows the programming and configuration options supported by Altera devices.

Table 19–4. Programming and Configuration Options

Option	FPGA	CPLD	Configuration Device and Enhanced Configuration Device	Serial Configuration Device
Program or Configure	✓	✓	✓	✓
Verify	—	✓	✓	✓
Blank-Check	—	✓	✓	✓
Examine	—	✓	✓	✓
Security Bit	—	✓	—	—
Erase	—	✓	✓	✓
ISP Clamp	—	✓ *(1)*	—	—
IPS File *(2)*	—	✓	—	—
Real-time ISP	—	✓ *(3)*	—	—

Notes to Table 19–4:

(1) Only MAX II and MAX 7000B CPLDs support the ISP Clamp feature.

(2) **.ips** file is used for ISP Clamp.

(3) Only MAX II CPLDs support the real-time ISP feature.

Tools Menu

More programmer options are available from the Tools menu. On the Tools menu, click **Options**. In the **Category** list, click **Programmer**. For descriptions of these options, refer to Table 19–5.

Table 19–5. Programmer Options (Part 1 of 2)

Option	Description
Show checksum without usercode	Determines whether the checksum values displayed in the programmer are calculated with or without JTAG user codes. This option allows you to have multiple versions of a programming or configuration file with different user codes, but share the same checksum.
Initiate configuration after programming	Specifies that configuration devices configure attached FPGA devices automatically after the programmer completes programming the configuration devices.
Display message when programming finishes	Displays a message when programming or other operation such as examining or blank-checking is complete.
Enable real-time ISP to allow background programming (for MAX II devices)	Can only be turned on if you are targeting a MAX II device. This option is turned off for all other device families. When this option is turned on, you can do the real-time in-system programming (ISP) for the MAX II device. The existing design in the MAX II device functions normally during and after the real-time ISP is completed. The new design starts to function after a power cycle to the device occurs. This option is also available in the programmer window.

Table 19–5. Programmer Options (Part 2 of 2)

Option	Description
Halt on-chip configuration controller	Halts the on-chip auto-configuration controller of the FPGA device for AS configuration, or the configuration device for PS or Fast Passive Parallel (FPP) configuration to allow JTAG configuration through a download cable. If you want to configure your FPGA through JTAG while the FPGA `MSEL` pins are set to AS mode, you should halt the on-chip configuration controller if any of the following circumstances occur: ■ The active serial configuration device connected to your FPGA is blank ■ The active serial configuration device is not present ■ An error occurs during AS configuration prior to JTAG configuration If the `MSEL` pins are set to PS or FPP mode, halt the configuration controller of the configuration device if an error occurs during PS or FPP configuration prior to JTAG configuration. The FPGA pulls the `nSTATUS` pin (which is connected to the `OE` pin of the configuration device) low to disable the configuration device.
Automatically check the Program/Configure checkbox when adding SOF	Automatically enables the program or configuration operation when adding an **.sof** file to the file list in the programmer window.
Configure volatile design security key (for Stratix III/IV devices)	Stratix III and Stratix IV GX devices support both volatile and nonvolatile design security keys. Checking this box before programming the key allows the volatile key to be programmed into the device.
Use bitstream compression to configure devices when available	Some Altera FPGAs support bitstream compression for PS configuration mode. To reduce configuration time, this option allows the devices to be configured with compressed configuration data from the Quartus II Programmer through PS mode.
Automatically open as detached window	Allows the Quartus II Programmer window to be detached from the Quartus II GUI when launched.
Use the enhanced mode Serial Flash Loader (SFL) IP for the factory default helper image	Allows the Quartus II Programmer to increase the speed of the programming and verification by filtering the extra paddings that are introduced by third-party programmer tools. In addition, this option increases the speed of programming and verification.
Check block CRCs to accelerate PFL/SFL verification when available	Reduces the time required to verify the data programmed into a flash device through the PFL. This option allows the Quartus II Programmer to verify the flash data by checking the cyclic redundancy code (CRC) generated from the data of the programming file, as well as of the flash device. This option can be used only with PFL designs created with the Quartus II software version 8.1 and later.

© March 2009 Altera Corporation

Hardware Setup

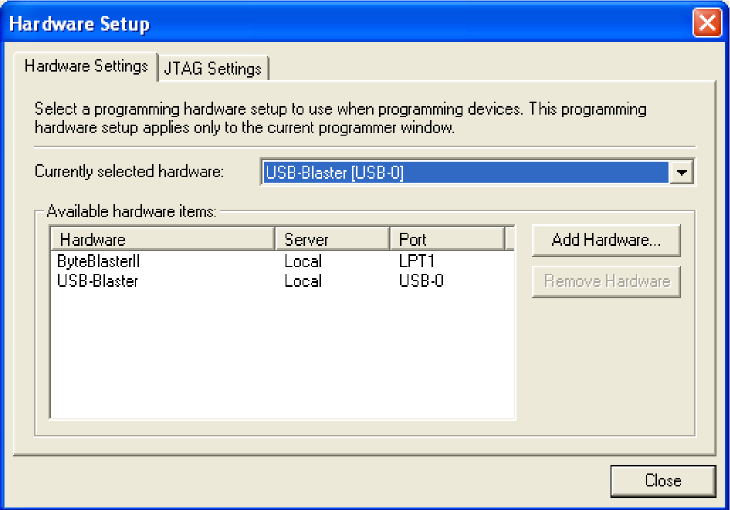
The Quartus II Programmer provides the flexibility to choose a download cable or the programming hardware. Before you can program or configure your device, you must have the correct hardware setup.

Hardware Settings

Click **Hardware Setup** to bring up the **Hardware Setup** dialog box. On the **Hardware Settings** tab (Figure 19–5), you can select a download cable or programming hardware available from the **Currently selected hardware** list. If the download cable or programming hardware you require is not displayed, click **Add Hardware** and specify the download cable or programming hardware. Make sure that you have installed the download cable driver before adding the hardware.

You do not have to manually add the USB-Blaster™ download cable to the list. After installing the driver, simply connect the download cable to the PC before opening the **Hardware Setup** dialog box. The USB-Blaster appears automatically in the list when the dialog box is opened.

Figure 19–5. Hardware Settings



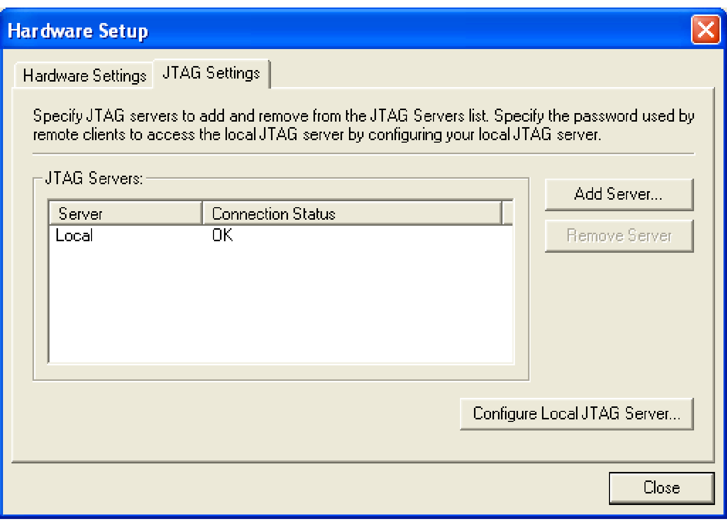
More information about programming hardware driver installation is available in the Design Software Support page on the Altera website.

JTAG Settings

The JTAG server allows programs such as the Quartus II Programmer to access the JTAG hardware. This application software is installed together with the Quartus II software. You can also access the JTAG download cable or programming hardware connected to a remote computer through the JTAG server of that computer. With the JTAG server, you can control the programming or configuration of devices from a single computer through other computers at remote locations. The JTAG server uses the TCP/IP communications protocol.

To view the **Hardware Setup** dialog box, click **Hardware Setup**. On the **JTAG Settings** tab (Figure 19–6), you can add or remove JTAG servers from the list. By default, you have only the local JTAG server (which is on your computer) in the list. By adding a remote JTAG server, you can access the JTAG hardware in that remote computer from your computer. You must have the password of the remote JTAG server to add the server to your list. Click **Add Server**, then enter the IP address of that computer in the **Server name** box and the password in the **Server password** box.

Figure 19–6. JTAG Settings



You can also allow remote clients to access the JTAG server on your computer and program or configure devices connected to your computer through the JTAG interface of your computer. Click **Configure Local JTAG Server** to enable the server and then enter the password that the remote clients require to access your JTAG server.

Device Programming and Configuration

The Quartus II Programmer supports single- or multi-device programming and configuration. This section describes the steps required to program or configure Altera devices, as well as how to bypass Altera and non-Altera devices in a JTAG chain.

Single Device Programming and Configuration

To program or configure a single device with the Quartus II Programmer, perform the following steps:

1. On the Tools menu, click **Programmer** to open the Programmer window.
2. Click **Hardware Setup** and select the programming hardware or download cable. If you are using JTAG mode, you can specify the correct JTAG settings for programming or configuration involving remote JTAG servers.
3. Click **Close**.
4. From the **Mode** list, select the programming or configuration mode.

5. Click **Add File** to add the **.pof** or **.sof** file to the programmer (you can omit this step if the file is already displayed). To change the file, select it and click **Change File**. To remove the file from the programmer, select it and click **Delete**.

If you use JTAG, AS, or in-socket programming mode, after the file has been added to the programmer, select the programming or configuration option by turning on the corresponding check box in the programmer.

6. Click **Start**.

Multi-Device Programming and Configuration

JTAG and PS modes allow you to program or configure a device chain. A JTAG chain can consist of a combination of FPGA, CPLD, and configuration devices that support JTAG mode. A PS chain consists of FPGAs that support PS mode. The steps for programming or configuring a device chain are similar to the steps for programming or configuring a single device. One exception is that in a device chain you must specify all the programming or configuration files for the devices you want to program or configure. JTAG mode allows you to bypass some of the devices in the JTAG chain while programming or configuring the rest of the devices. PS mode does not allow you to bypass devices in the FPGA chain.

Bypassing an Altera Device

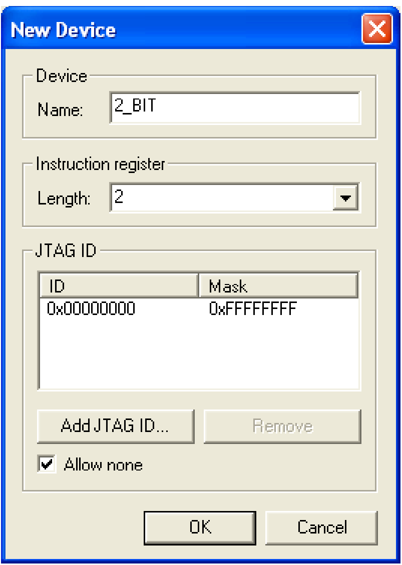
If you do not want to program an Altera device in the chain, turn off all the options in the row of that device. If you do not have the programming or configuration file for that device, click **Add Device** to specify the device.

Bypassing a Non-Altera Device

The JTAG chain you want to program or configure may contain non-Altera devices. To program or configure your Altera device in the JTAG chain, you must bypass those non-Altera devices. If the non-Altera devices are not in the list of devices that you can select, click **Add Device** in the programmer.

To bypass the devices, you must manually enter each device name and its JTAG instruction register length and JTAG ID code. Click **Add Device** to open the **Select Device** dialog box. Click **New** to define a device. In the **New Device** dialog box (Figure 19–7), enter the name of the device and the JTAG instruction register length of the device. You can find the JTAG instruction register length in the device's data sheet. You can also specify the JTAG ID code for the device by clicking **Add JTAG ID**. This is optional and you can turn on **Allow none** to set the ID code to all zeros. If you do not specify the JTAG ID code, the default value is all zeros.

Figure 19–7. New Device Dialog Box



After defining the device, the device appears in the device list (Figure 19–8). Click **Export** to save the information in a Quartus User-Defined Device (**.qud**) file. This file saves the information for the user-defined devices that appear under **Device name** in the dialog box and can be used by other Quartus II projects as well. To obtain information on the user-defined devices from the **.qud** file, click **Import** and the devices are listed under **Device name**.

Figure 19–8. Select Devices

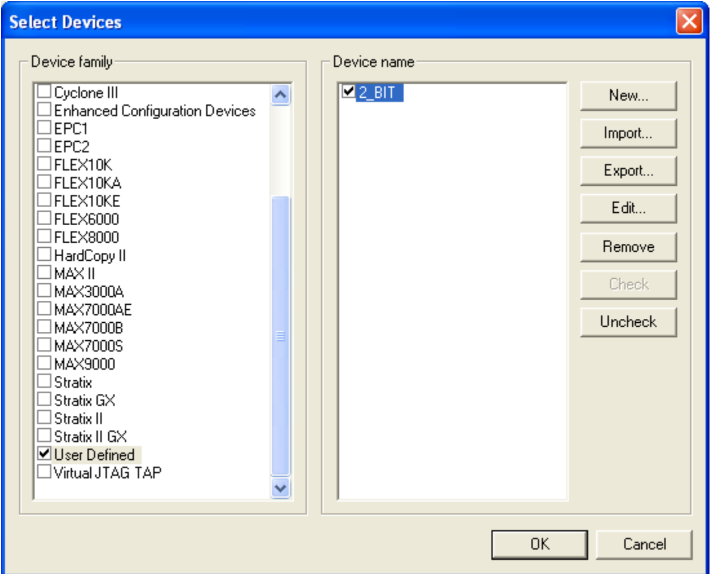
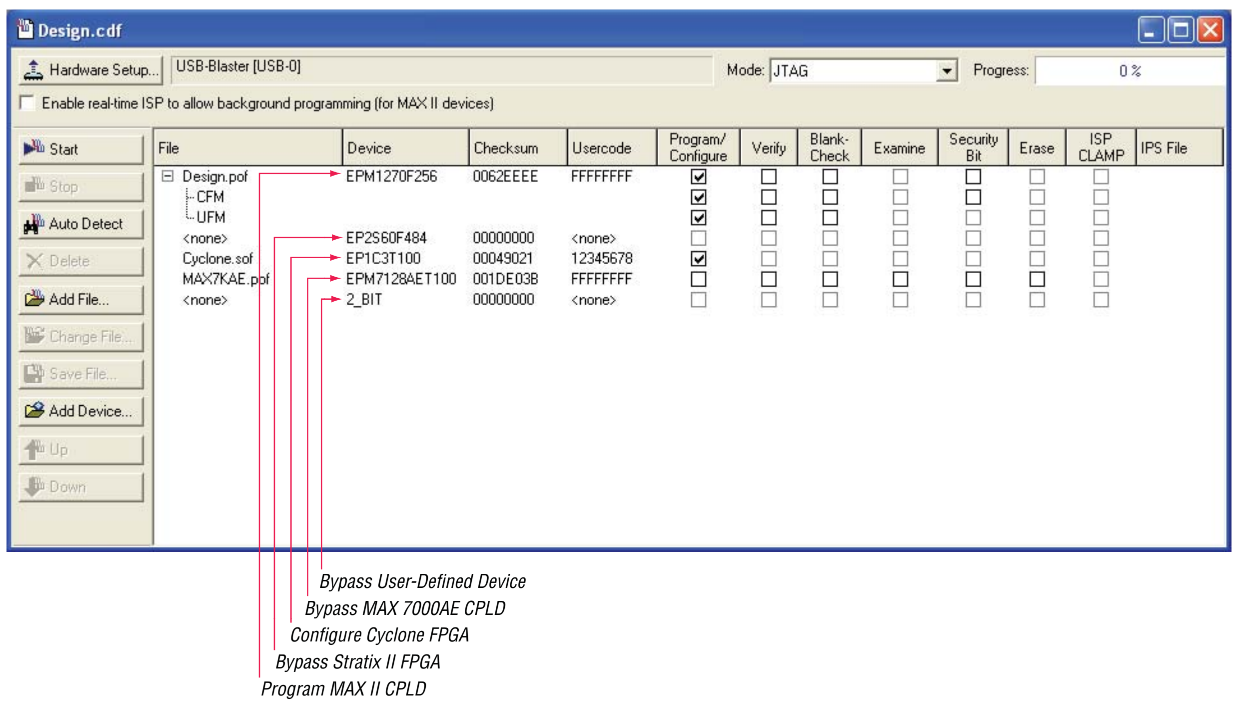


Figure 19–9 shows the programmer window for a JTAG chain.

Figure 19–9. Multi-Device JTAG Chain



Chain Description File

All the information in the Quartus II Programmer, including the programming or configuration mode, programming or configuration files used, device chain information, and the programming options specified can be saved in a chain description file (**.cdf**). You do not have to specify the information each time you program the device chain. Simply open the **.cdf** file in the Quartus II software and the information appears in the Quartus II Programmer GUI.

Design Security Key Programming

The Quartus II Programmer supports the generation of encryption key programming files and encrypted configuration files for Altera FPGAs that support the design security feature. You can also use the Quartus II Programmer to program the encryption key into the FPGA.

For more information about using the design security feature with the Quartus II software, refer to *AN 341: Using the Design Security Feature in Stratix II and Stratix II GX Devices* and *AN 512: Using the Design Security Feature in Stratix III Devices*.

Optional Programming Files

The Quartus II software can generate optional programming or configuration files in various formats to be used with programming tools other than the Quartus II Programmer. In addition, you can convert the FPGA configuration files to programming files for configuration devices.

Types of Programming and Configuration Files

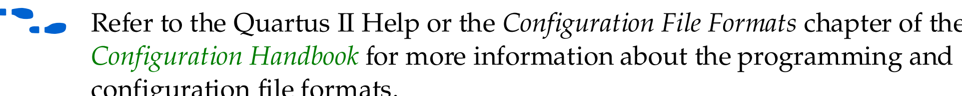
The Quartus II software generates programming files of various formats for use with different programming tools. Table 19–6 shows the programming and configuration files generated by the Quartus II software.

Table 19–6. Types of Programming and Configuration Files (Part 1 of 2)

File Format	Generated by the Quartus II Software	Supported by the Quartus II Programmer	Description
.sof	✓	✓	This configuration data file is used for configuring FPGA devices. The Quartus II Assembler generates this file when you compile your FPGA design.
.pof	✓	✓	This programming data file is used for programming CPLDs and configuration devices. The Quartus II Assembler generates the CPLD **.pof** file when you compile your CPLD design. The configuration device **.pof** file is converted from the FPGA **.sof** file.
.jam	✓	✓	This ASCII-format file is used for configuring or programming one or more FPGAs, CPLDs, and configuration devices in a JTAG chain. The **.jam** file includes both programming algorithm and data. Apart from the Quartus II Programmer, you can use Altera's Jam Standard Test and Programming Language (STAPL) player, the `quartus_jli` executable, or other third-party programming tools together with the **.jam** file. The **.jam** file is also suitable for embedded processor-type programming environments.
.jbc	✓	✓	Similar to the **.jam** file, this binary-format file is used for configuring or programming one or more FPGAs, CPLDs, and configuration devices in a JTAG chain. The **.jbc** file includes both the programming algorithm and data, and the size is smaller than the **.jam** file. In addition to the Quartus II Programmer, you can use Altera's Jam Byte-Code player, the `quartus_jli` executable, or other third-party programming tools together with the **.jbc** file. The **.jbc** file is also suitable for embedded processor-type programming environments.
Serial Vector Format File (**.svf**)	✓	—	This ASCII-format file is used for configuring, programming, blank-checking, and verifying one or more FPGAs, CPLDs, and configuration devices in a JTAG chain. The **.svf** file, which includes programming algorithm and data, is suitable for an automated test equipment (ATE) environment that requires a fixed programming algorithm.

Table 19–6. Types of Programming and Configuration Files (Part 2 of 2)

File Format	Generated by the Quartus II Software	Supported by the Quartus II Programmer	Description
In-System Configuration File (**.isc**)	✓	—	This data file is used with the IEEE 1532 BSDL file for programming a single device that supports IEEE 1532 programming. The Quartus II software supports the **.isc** file for MAX 7000AE, MAX 7000B, and MAX 3000A CPLDs.
Hexadecimal Intel-Format Output File (**.hexout**)	✓	—	The **.hexout** file is used for programming FPGA configuration data into enhanced configuration devices or other storage devices. For enhanced configuration devices, use the enhanced configuration device **.pof** file to generate the **.hexout** file. Use the FPGA **.sof** file to generate the **.hexout** file for other storage devices (for example, the flash or EEPROM devices). You can use a microcontroller to read back the data from the storage device and configure the FPGA. To program the enhanced configuration device or other storage devices with the **.hexout** file, you can use other third-party programming tools.
Raw Binary File (**.rbf**)	✓	—	This binary file contains configuration data for one or more FPGAs. You can use Microblaster or Altera's JRunner software to configure your FPGA device with the **.rbf** file. The **.rbf** file is also suitable for embedded processor configuration environments.
Tabular Text File (**.ttf**)	✓	—	This ASCII file contains configuration data for one or more FPGAs. The **.ttf** file is used for embedded processor-type configuration.
Raw Programming Data (**.rpd**)	✓	—	This binary file is used for programming serial configuration devices. Use the serial configuration device **.pof** file to generate this file. You can use Altera's SRunner software to program your serial configuration device with the **.rpd** file.
JTAG Indirect Configuration File (**.jic**)	✓	✓	The **.jic** file is used for programming serial configuration devices through JTAG with the Quartus II Programmer and Altera FPGAs that support AS configuration mode.


Refer to the Quartus II Help or the *Configuration File Formats* chapter of the *Configuration Handbook* for more information about the programming and configuration file formats.

For more information about using the **.jam** and **.jbc** programming files with the Jam STAPL Player, Jam STAPL Byte-Code Player, and the `quartus_jli` command-line executable, refer to *AN 425: Using Command-Line Jam STAPL Solution for Device Programming*.

Generating Optional Programming Files

When you compile your design, the Quartus II Assembler generates the **.sof** file for an FPGA or a **.pof** file for a CPLD. With the **.sof** or **.pof** file for your design, you can then create other optional programming or configuration files, or convert the **.sof** file to target a particular configuration device.

Create Programming Files

The Quartus II software allows you to create **.jam**, **.jbc, .svf**, or **.isc** programming or configuration files. In addition, you can create **.jam**, **.jbc**, and **.svf** files for a JTAG chain that consists of more than one device.

To create the files, open the Quartus II Programmer, set the programming or configuration mode to JTAG, and then add the programming or configuration files or devices to the programmer. On the File menu, click **Create/Update** and then click **Create JAM, SVF, or ISC File**. Select the file format and name the file accordingly.

An **.svf** file can only be created for programming or verification. In addition, you can specify whether or not to do the optional blank-check operation with the **.svf** file.

Convert Programming Files

To store the FPGA data into configuration devices, you can convert the **.sof** data to another format and program the configuration device. The Quartus II software supports converting the data into **.pof**, **.hexout**, **.rbf**, **.ttf**, **.rpd**, or **.jic** format.

For more information about converting programming files with the Quartus II software, refer to the *Configuration File Formats* chapter of the *Configuration Handbook*.

Generating Optional Programming or Configuration Files During Compilation

The Quartus II software can generate optional programming or configuration files automatically when you compile your design. To select the format of the optional programming or configuration files to be generated during compilation, on the Assignments menu, click **Settings**. Under **Device**, click **Device and Pin Options**.

You can select the configuration device from the **Configuration** tab for the **.pof** file generation. For other optional programming and configuration file generation, you can select the file format under the **Programming Files** tab.

Flash Loaders

Serial configuration devices and the common flash interface (CFI) flash devices do not support the JTAG interface and cannot be programmed directly through the normal JTAG programming. Flash loaders allow the programming of the serial configuration device and the CFI flash from the Quartus II Programmer through JTAG.

Parallel Flash Loader

The parallel flash loader (PFL) performs two functions:

- Allows the programming of the CFI flash through the JTAG interface
- Acts as the configuration controller that reads the configuration data from the CFI flash and configures the FPGA

To program the CFI flash, the PFL uses the MAX II device as a bridge between the JTAG interface of the Quartus II Programmer and the CFI of the CFI flash device. You can program FPGA configuration data and user data into the flash with a flash **.pof** file generated by the Quartus II software. After the flash is programmed with the FPGA configuration data, the PFL is then used to read the configuration data back from the CFI flash to configure the FPGA.

For more information about the PFL, refer to *AN 386: Using the MAX II Parallel Flash Loader with the Quartus II Software*.

Serial Flash Loader

The serial flash loader (SFL) allows programming of the serial configuration devices through JTAG. The SFL uses the FPGA device that supports AS configuration mode as a bridge between the active serial memory interface (ASMI) of the serial configuration device and the JTAG interface of the programmer. The Quartus II Programmer uses the **.jic** file converted from the FPGA **.sof** file to program the serial configuration device though JTAG.

For more information about the SFL, refer to *AN 370: Using the Serial Flash Loader with the Quartus II Software*.

JTAG Chain Debugger Tool

The JTAG Chain Debugger tool is a Quartus II Programmer feature that allows you to test the JTAG chain integrity and detect intermittent failures of the JTAG chain. In addition, the tool allows you to shift in JTAG instructions and data through the JTAG interface as well as stepping through the test access port (TAP) controller state machine for debugging purpose.

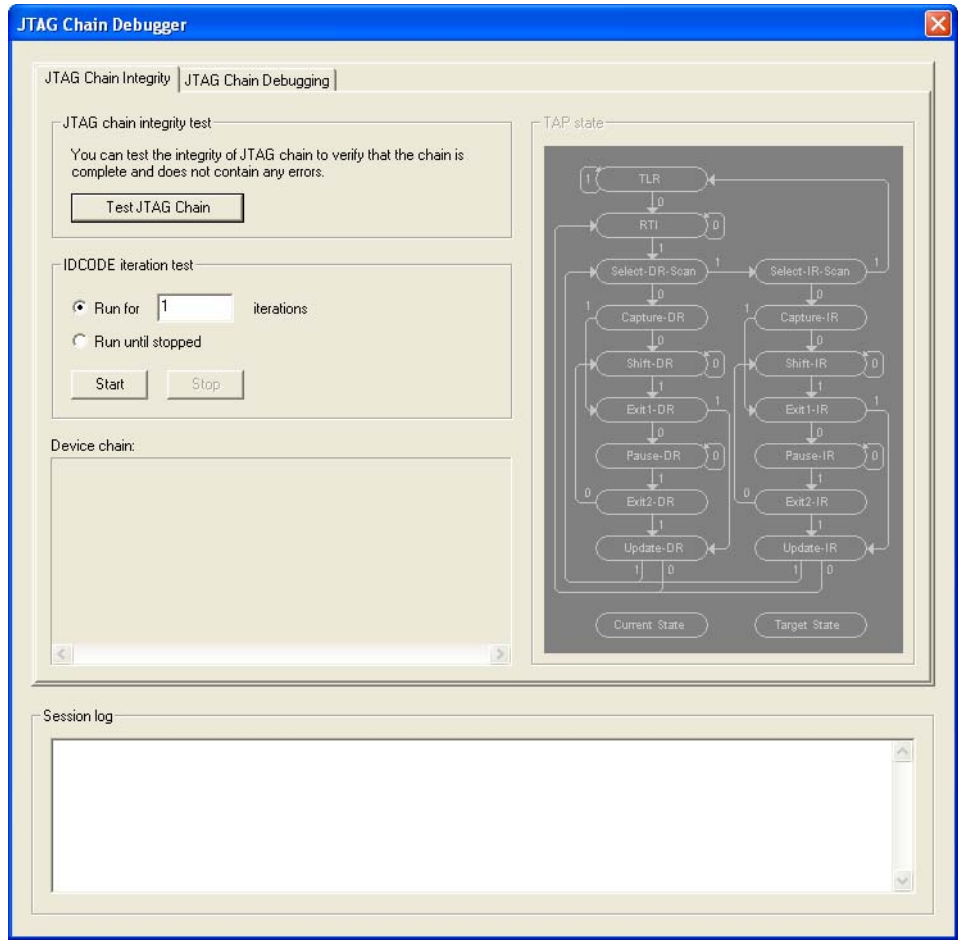
To launch the JTAG Chain Debugger, open the Quartus II Programmer. From the Processing menu, click **JTAG Chain Debugger**. The Quartus II programmer also prompts you to launch the JTAG Chain Debugger if the **Auto Detect** operation fails to detect the device chain.

The JTAG Chain Debugger GUI is divided into two sections: the JTAG Chain Integrity section and the JTAG Chain Debugging section.

JTAG Chain Integrity

From the **JTAG Chain Integrity** tab in the JTAG Chain Debugger tool, you can perform the JTAG chain integrity test and the IDCODE iteration test. Figure 19–10 shows the **JTAG Chain Integrity** tab in the JTAG Chain Debugger tool.

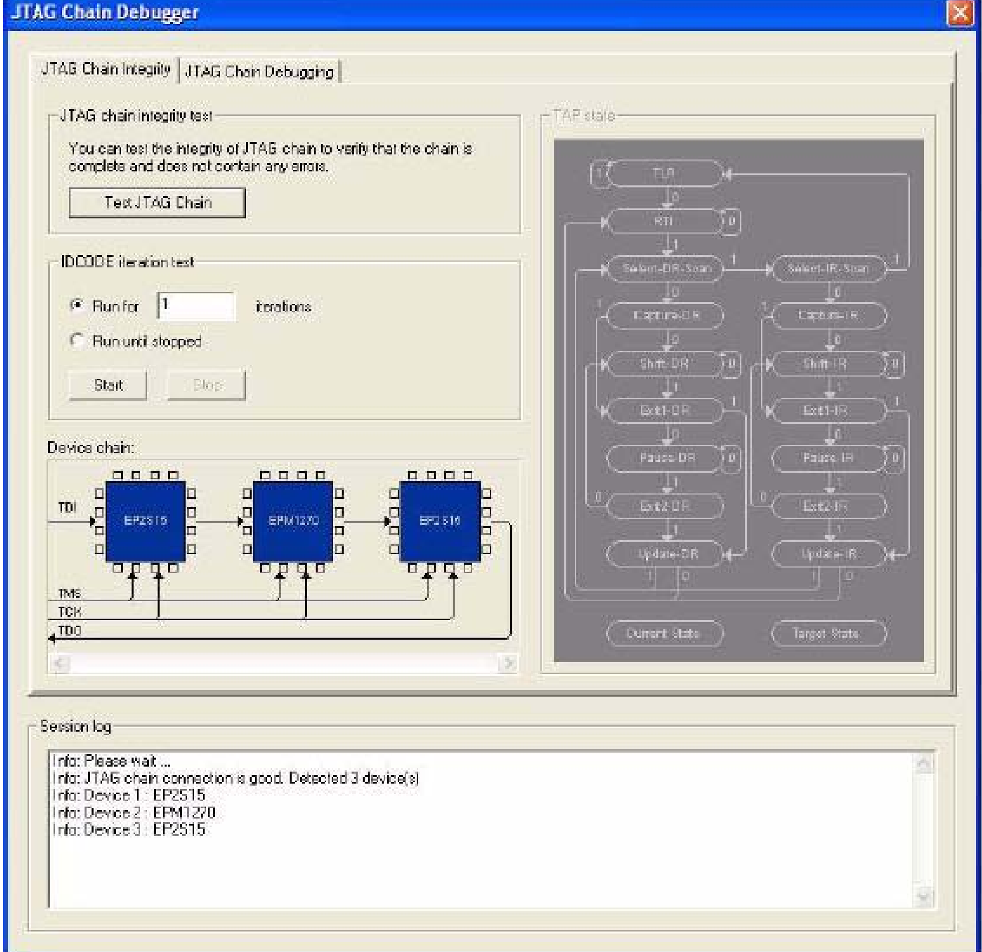
Figure 19–10. JTAG Chain Integrity Tab



JTAG Chain Integrity Test

The JTAG chain integrity test performs a simple test to ensure that there is t_{DO}-to-t_{CO} connectivity for the JTAG chain under test. When you click the **Test JTAG Chain** button, the tool attempts to read out the 32-bit IDCODE of the device or devices in the chain. If the JTAG chain connection is good, the device IDCODEs are shifted out from the chain through the `TDO` output. The **Device chain** box displays the device or devices detected in the chain based on the IDCODEs shifted out, and the **Session log** message box reports the test result. Figure 19–11 shows that the tool has detected three devices in the JTAG chain and the connection is good.

Figure 19–11. Good JTAG Chain Connection

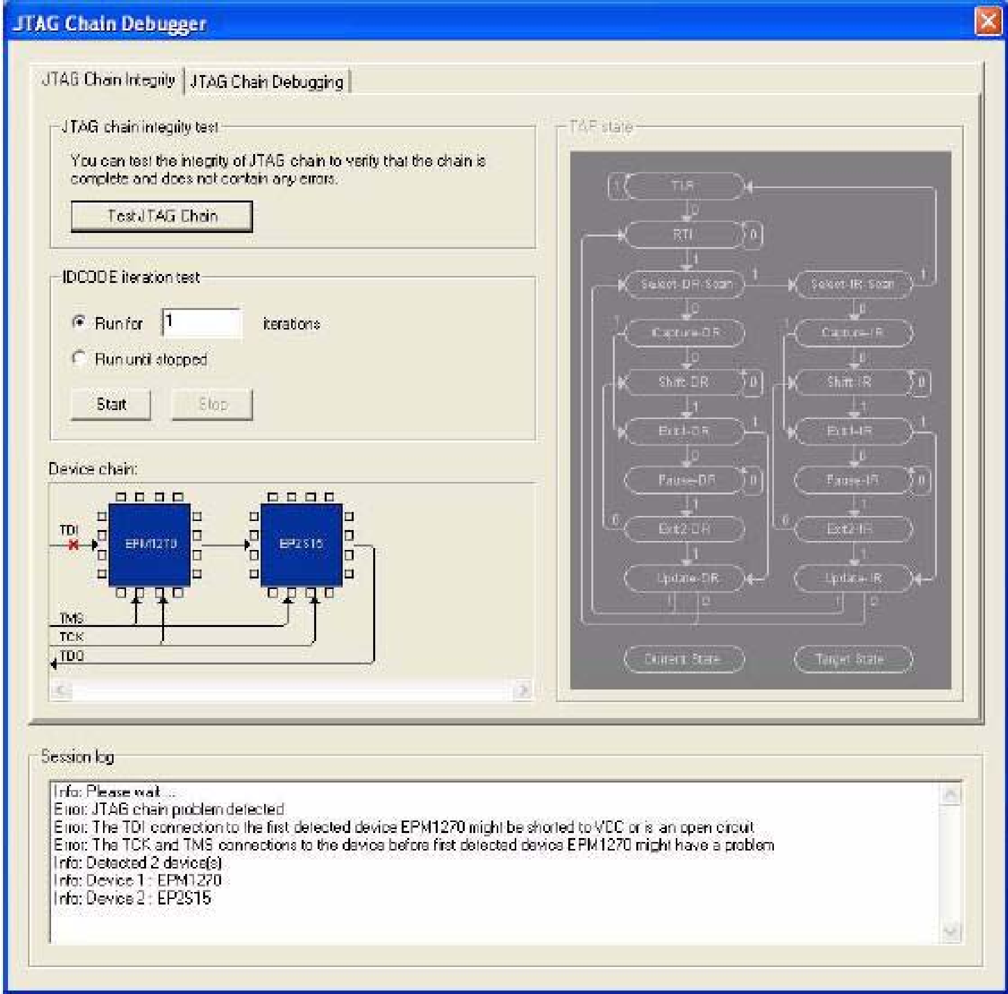


The IDCODE of the device cannot be shifted out through the chain if one or both of the following two circumstances exist:

- There is a connection problem between the `TDO` pin of a device in the chain and the `TDI` pin of the subsequent device in the chain (in which case the connection might be open)
- The device has `TCK` or `TMS` connection problems

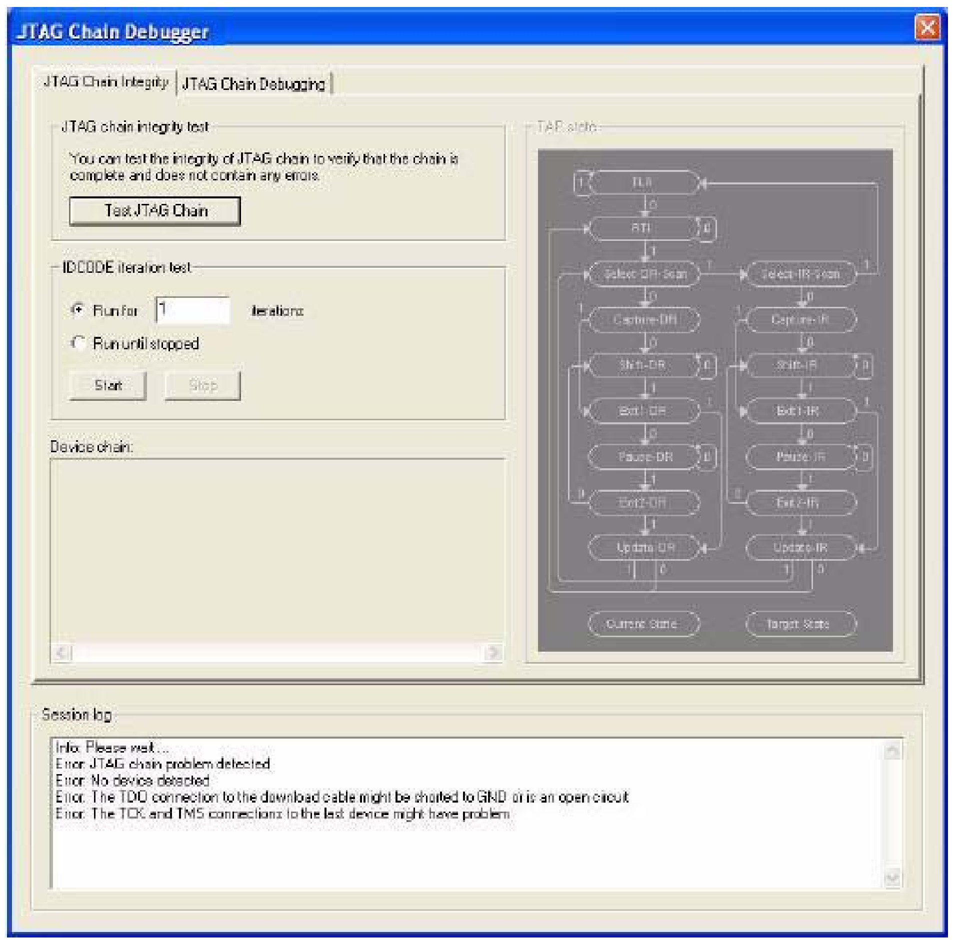
The tool can detect only the subsequent devices in the chain and report the potential problems in the chain. Figure 19–12 shows that the tool detects only two devices in the chain, and the first device in the chain has a connection problem.

Figure 19–12. Incomplete JTAG Chain Detected



If there is a connection problem between the `TDO` pin of the last device in the chain and the download cable, the JTAG Chain Debugger is not able to capture any data from the chain. The tool reports the error, as shown in Figure 19–13. In addition to the `TDO` connection problem, the tool also reports this problem if the devices in the chain are not powered up or the download cable is not connected to the JTAG chain.

Figure 19–13. No Devices Detected



IDCODE Iteration Test

You can perform the IDCODE iteration test to check the consistency of the JTAG chain. The operation is similar to the JTAG chain integrity test, except that the test can be repeated a number of times. The IDCODE test is able to detect intermittent failures that might happen in the chain based on the number of tests done. The tool reports whether the chain is consistent by checking for chain intermittent discontinuity, apart from showing the device detected.

You can specify the number of iterations to run (up to 9999 iterations) or manually stop the test by selecting **Run until stopped**. Press the **Stop** button to stop the test if the **Run until stopped** option is selected.

The **Auto Detect**, **JTAG chain integrity test**, and **IDCODE iteration test** features are used for checking the JTAG chain connection and detecting the devices in the chain. However, there are some differences between the three features. Table 19–7 shows the differences between the **Auto Detect**, **JTAG chain integrity test**, and **IDCODE iteration test** features.

Table 19–7. Differences between Auto Detect, JTAG Chain Integrity Test, and IDCODE Iteration Test

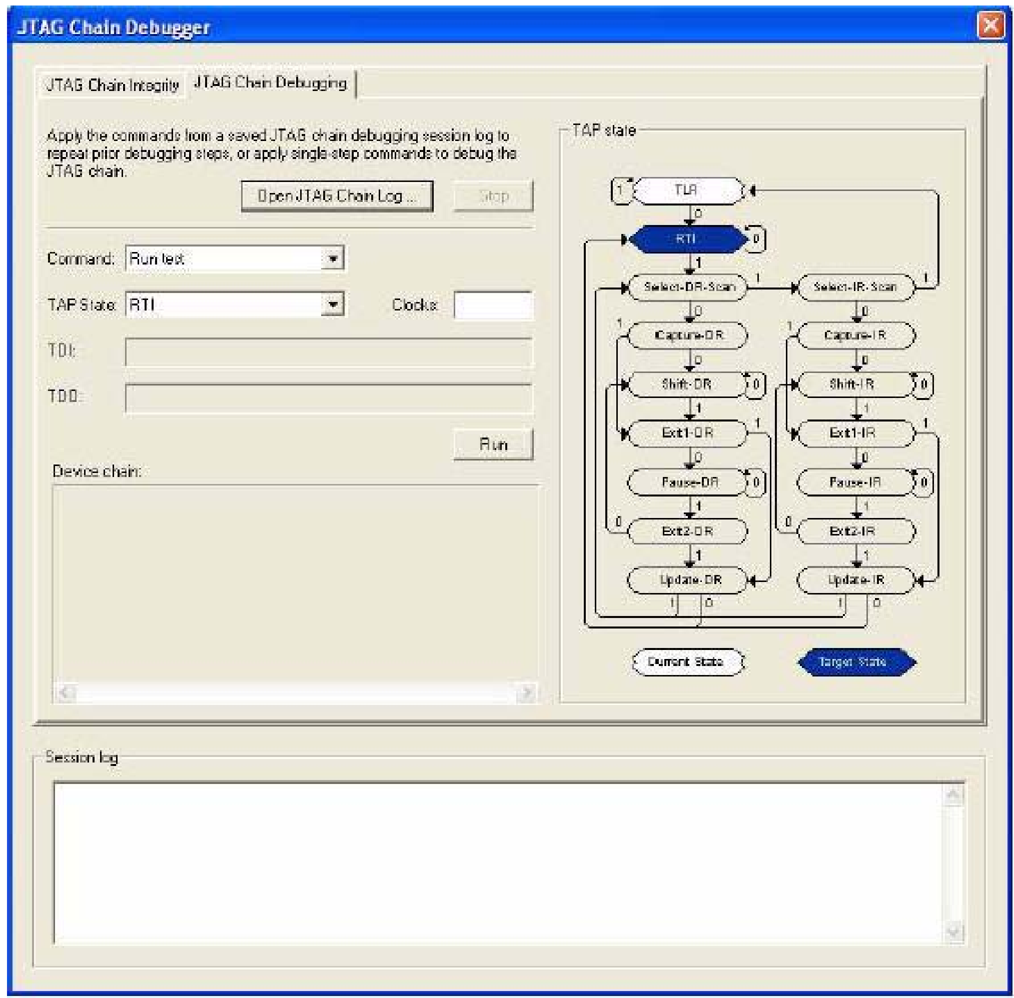
Auto Detect	JTAG Chain Integrity Test	IDCODE Iteration Test
Does not list any device detected in the chain if there is any connection problem in the chain	Lists device or devices detected in the chain, regardless of whether any of the following circumstances exist: ■ Connection problem between the download cable and the `TDI` input of the first device in the chain ■ `TDO` and `TDI` connection problem between the devices in the chain ■ `TCK` or `TMS` connection problem of devices in the chain other than the last device in the chain.	Lists device(s) detected in the chain, regardless of whether any of the following circumstances exist: ■ Connection problem between the download cable and the `TDI` input of the first device in the chain ■ `TDO` and `TDI` connection problem between the devices in the chain ■ `TCK` or `TMS` connection problem of devices in the chain other than the last device in the chain.
Error reporting more general; does not report what problem in the chain is	Error reporting more specific; reports the potential problem in the chain	Error reporting more specific; reports the potential problem in the chain
Test is only performed once	Test is only performed once	Test can be repeated based on the number specified, or until stopped by the user
Not for detecting intermittent chain connection problem	Not for detecting intermittent chain connection problem	For detecting intermittent chain connection problem

JTAG Chain Debugging

The **JTAG Chain Debugging** tab allows you to shift the JTAG instructions and data manually into the JTAG chain, as well as control the test access port (TAP) state machine of the device or devices in the JTAG chain.

Figure 19–14 shows the **JTAG Chain Debugging** tab.

Figure 19–14. JTAG Chain Debugging Tab



There are four JTAG chain debugging commands. Table 19–8 lists the commands and the descriptions.

Table 19–8. JTAG Chain Debugging Commands

Command	Description
Run Test	Forces the TAP state machine to run for a specific number of TCK cycles in one of the following states. ■ TLR (Test-Logic-Reset) ■ RTI (Run-Test-Idle) ■ Pause-DR ■ Pause-IR You must specify the number of TCK cycles the TAP state machine needs to run in the specified state.
Scan Instruction Register	Allows you to shift the JTAG instruction(s) into the instruction register(s) of the device(s) in the chain. You can specify one of the following states in which the TAP state machine stops when finished shifting in the instructions: ■ RTI ■ Pause-IR ■ TLR You must specify the instruction to be shifted in through TDI, and the number of TCK cycles
Scan Data Register	Allows you to shift the data into the device(s) in the chain. You can specify one of the following states in which the TAP state machine stops when it has completed shifting in the data: ■ RTI ■ Pause-DR ■ TLR You must specify the data to be shifted in through TDI, and the number of TCK cycles.
Goto State	Forces the TAP state machine to any one of the specified states.

To perform JTAG chain debugging with the tool, perform the following steps:

1. In the **Command** pull-down list, select the appropriate JTAG command (Figure 19–14).
2. Depending on the command you selected, make the following specifications:
   - The TAP state that is related to the JTAG command
   - The number of `TCK` cycles
   - The value to be scanned into the chain, through `TDI`, expressed as a hex value
3. To execute the command selected, click **Run**.

When you shift the instructions or data into the JTAG chain, the scanned-out value is displayed in hex by the tool.

The tool shows the current and target TAP state in the TAP state diagram. The target TAP state shown in the TAP state diagram depends on the state specified in the **TAP State** pull-down list. This is where the TAP state machine stops upon the successful execution of the JTAG command.

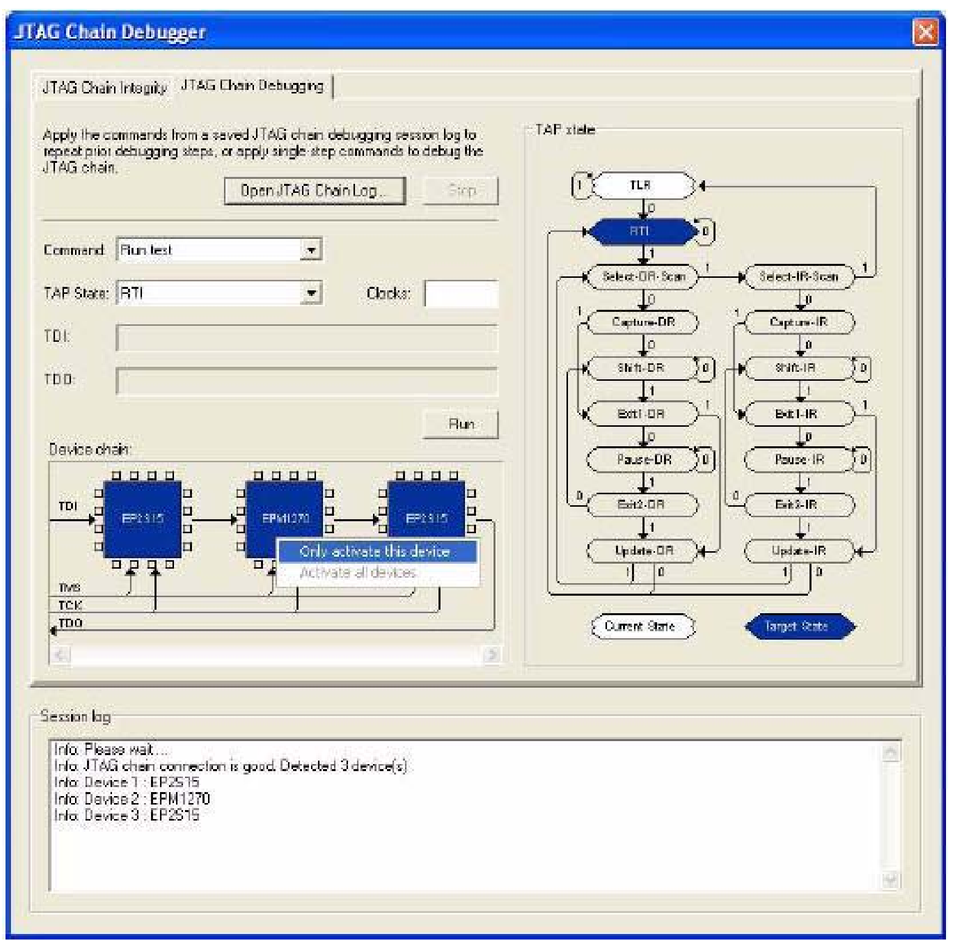
You can also double-click on a specific state in the state diagram, causing the TAP state machine to go to that state. This is similar to selecting **Goto State** from the **Command** pull-down list and clicking **Run**.

Bypassing Devices in the Chain

For a multi-device JTAG chain, you can isolate one device on the chain and perform your debugging on that device only.

1. Perform the JTAG chain integrity test before initiating the debug session. The tool displays the devices detected in the chain.
2. In the **JTAG Chain Debugging** tab, right-click the one device in the **Device chain** box that you do *not* want to bypass.
3. Select **Only activate this device**, as shown in Figure 19–15. Only the active device is in dark blue. All other devices are in bypass mode.

Figure 19–15. Activating Only One Device in the JTAG Chain

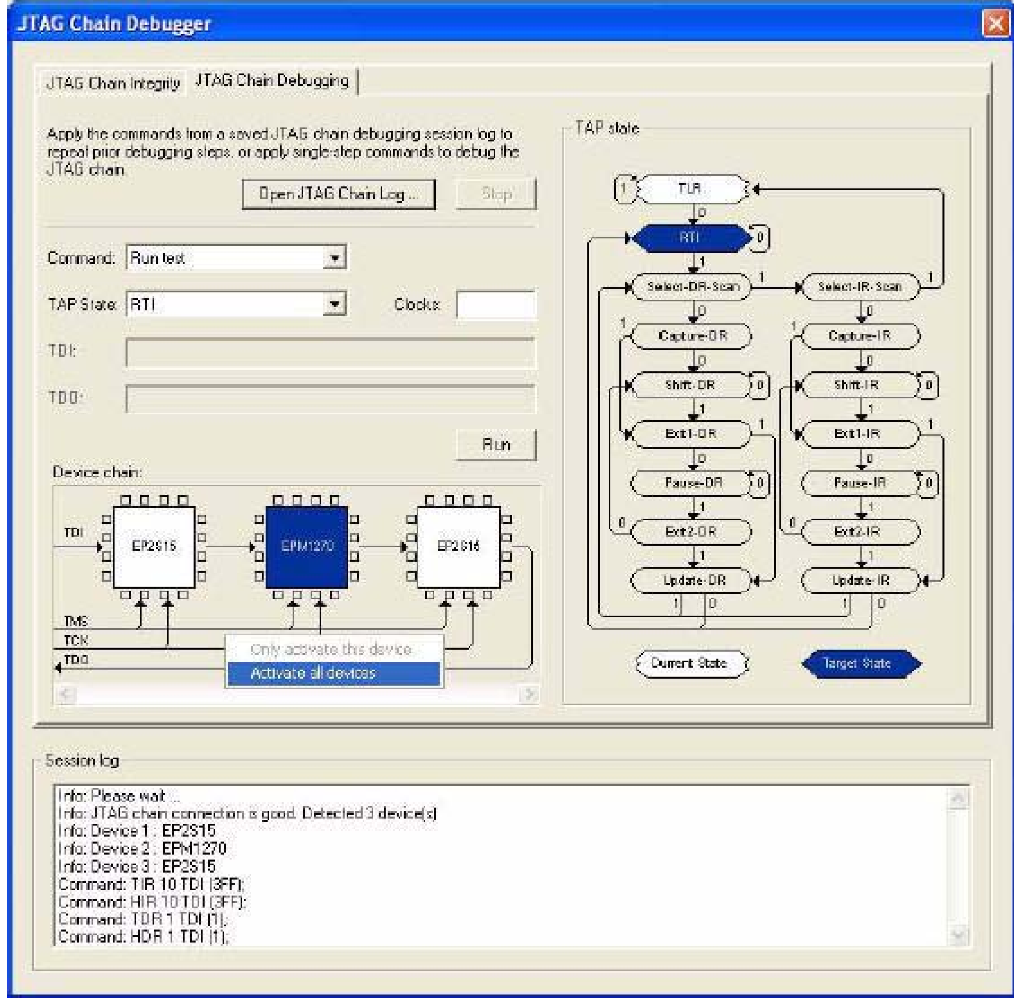


When only one device is enabled in the chain, the JTAG Chain Debugger automatically pads the additional bits for the bypassed devices when scanning the value into the chain. You can debug the chain as if this was the only one device in the chain.

© March 2009 Altera Corporation

To enable all of the bypassed devices in the chain, right-click the area inside the **Device chain** box (other than on the devices), and select **Activate all devices**, as shown in Figure 19–16.

Figure 19–16. Activating All Devices in the Chain



JTAG Chain Log

In addition to the test results and messages, all of the JTAG chain debugging operations you performed are listed in the Session log window. You can save the session log so you can play back the debugging sequences later. The commands, messages, and test results are saved in the session log file.

To save the session log, right-click the session log and select **Save Session Log**. Name the file accordingly. To clear the session log, right-click the session log and select **Clear Session Log**.

To play back the previously saved session log, click the **Open JTAG Chain Log** button and open the saved file. The JTAG Chain Debugger then executes the debug sequence commands in the file. Click the **Stop** button to stop the execution of the file.

Other Programming Tools

This section covers other programming tools that are related to the Quartus II Programmer and can be used for programming or debugging programming problems.

Quartus II Stand-Alone Programmer

If you do not have the full version of the Quartus II software, Altera offers the free Quartus II Stand-Alone Programmer. This stand-alone programmer has the full function of the normal Quartus II Programmer, and enables you to create or convert programming files from the **.sof** or **.pof** file of your design. You can download the Quartus II Stand-Alone Programmer from the Download Center on the Altera website.

jtagconfig Debugging Tool

The `jtagconfig` command-line utility is included with the Quartus II software. You can use this utility (which is similar to the auto detect operation in the Quartus II Programmer) to check the devices in a JTAG chain and the user-defined devices.

For more information about the `jtagconfig` utility, type one of the following commands at the command prompt:

```
jtagconfig -h ↵
jtagconfig --help ↵
```

Scripting Support

In addition to the Quartus II Programmer GUI, you can use the Quartus II command-line programmer (`quartus_pgm`) to enter commands. The `quartus_pgm` command-line programmer comes with the Quartus II Programmer. You can run this programmer separately from the Quartus II software. You can also run the procedures for the programmer in a Tcl script. The programmer accepts the **.pof**, **.sof**, and **.jic** programming or configuration files. You can also use the **.cdf** file.

For more information about the command-line syntax, type one of the following commands at the command prompt:

```
quartus_pgm -h ↵
quartus_pgm --help ↵
```

For more information about a specific programmer option or topic, type the following command at the command prompt:

```
quartus_pgm --help=<option|topic> ↵
```

Example 19–1 shows a command that programs a device:

Example 19–1. Programming a Device

```
quartus_pgm -c byteblasterII -m jtag -o bpv;design.pof ↵
```

Where:

`-c byteblasterII` specifies the ByteBlaster II download cable
`-m jtag` specifies the JTAG programming mode
`-o bpv` represents the blank-check, program, and verify operations
`design.pof` represents the **.pof** file used for the programming

The programmer automatically executes the erase operation before programming the device.

For detailed information about scripting command options, you can also refer to the Quartus II Command-Line and Tcl API Help browser. To run the Help browser, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF format.

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For information about all settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Manual*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Conclusion

The Quartus II Programmer offers you a wide variety of options to program and configure your Altera devices. With the Quartus II Programmer, the Quartus II software provides you with a complete solution for your FPGA or CPLD design prototyping, which can even be performed in the production environment.

Referenced Documents

This chapter references the following documents:

- *AN 341: Using the Design Security Feature in Stratix II and Stratix II GX Devices*
- *AN 370: Using the Serial FlashLoader with the Quartus II Software*
- *AN 386: Using the MAX II Parallel Flash Loader with the Quartus II Software*
- *AN 425: Using Command-Line Jam STAPL Solution for Device Programming*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Configuration File Formats* chapter of the *Configuration Handbook*
- *Configuration Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Manual*

- *Serial Configuration Devices (EPCS1, EPCS4, EPCS16, EPCS64 and EPCS128) Data Sheet* of the *Configuration Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 19–9 shows the revision history for this chapter.

Table 19–9. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ Added a row to Table 19–5 on page 19–9 ■ Changed references from “JTAG Chain Debug” to “JTAG Chain Debugger” ■ Updated Figure 19–10 through Figure 19–16	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	Added the new section “JTAG Chain Debug Tool” on page 19–20 and associated sub-headings, including: ■ “JTAG Chain Integrity” on page 19–21 ■ “JTAG Chain Integrity Test” on page 19–22 ■ “IDCODE Iteration Test” on page 19–24 ■ “JTAG Chain Debugging” on page 19–25 ■ “Bypassing Devices in the Chain” on page 19–28 ■ “JTAG Chain Log” on page 19–29 Minor editorial updates Updated entire chapter using 8½” × 11” chapter template	Updated according to Quartus II software version 8.1 release, including the JTAG Chain Debugging Tool, which is new with the current release.
May 2008 v8.0.0	Updated date and part number and added hypertext links. Also updated file format naming conventions.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

 © March 2009 Altera Corporation

About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 9.0.

How to Contact Altera

For the most up-to-date information about Altera products, see the following table.

Contact (Note 1)	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note:

(1) You can also contact your local Altera sales office or sales representative.

Third-Party Software Product Information

Third-party software products described in this handbook are not Altera products, are licensed by Altera from third parties, and are subject to change without notice. Updates to these third-party software products may not be concurrent with Quartus II software releases. Altera has assumed responsibility for the selection of such third-party software products and its use in the Quartus II 9.0 software release. To the extent that the software products described in this handbook are derived from third-party software, no third party warrants the software, assumes any liability regarding use of the software, or undertakes to furnish you any support or information relating to the software. EXCEPT AS EXPRESSLY SET FORTH IN THE APPLICABLE ALTERA PROGRAM LICENSE SUBSCRIPTION AGREEMENT UNDER WHICH THIS SOFTWARE WAS PROVDED TO YOU, ALTERA AND THIRD-PARTY LICENSORS DISCLAIM ALL WARRANTIES WITH RESPECT TO THE USE OF SUCH THIRD-PARTY SOFTWARE CODE OR DOCUMENTATION IN THE SOFTWARE, INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT. For more information, including the latest available version of specific third-party software products, refer to the documentation for the software in question.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, file names, file name extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>***.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
“Subheading Title”	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: “Typographic Conventions.”
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user’s work.
WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
(feet)	The feet direct you to more information on a particular topic.

Quartus II Handbook Version 9.0

Volume 4: SOPC Builder

101 Innovation Drive
San Jose, CA 95134
www.altera.com

QII5V4-9.0

Copyright © 2009 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

nsai

I.S. EN ISO 9001

Contents

Chapter Revision Dates xi

Section I. SOPC Builder Features

Chapter 1. Introduction to SOPC Builder

Quick Start Guide . . . 1-1
Overview . . . 1-1
Architecture of SOPC Builder Systems . . . 1-2
SOPC Builder Modules . . . 1-2
Example System . . . 1-2
Available Components . . . 1-3
Custom Components . . . 1-4
Third-Party Components . . . 1-4
Functions of SOPC Builder . . . 1-5
Defining and Generating the System Hardware . . . 1-5
Creating a Memory Map for Software Development . . . 1-6
Creating a Simulation Model and Test Bench . . . 1-6
Visualization of Large SOPC Builder Systems . . . 1-6
Operating System Support . . . 1-6
Talkback Support . . . 1-7
Referenced Documents . . . 1-7
Document Revision History . . . 1-8

Chapter 2. System Interconnect Fabric for Memory-Mapped Interfaces

Introduction . . . 2-1
High-Level Description . . . 2-1
Fundamentals of Implementation . . . 2-3
Functions of System Interconnect Fabric . . . 2-3
Address Decoding . . . 2-4
Datapath Multiplexing . . . 2-5
Wait State Insertion . . . 2-5
Pipelined Read Transfers . . . 2-6
Dynamic Bus Sizing and Native Address Alignment . . . 2-7
Dynamic Bus Sizing . . . 2-7
Wider Master . . . 2-7
Narrower Master . . . 2-8
Native Address Alignment . . . 2-8
Arbitration for Multimaster Systems . . . 2-9
Traditional Shared Bus Architectures . . . 2-9
Slave-Side Arbitration . . . 2-10
Arbiter Details . . . 2-10
Arbitration Rules . . . 2-11
Setting Arbitration Parameters in SOPC Builder . . . 2-11
Fairness-Based Shares . . . 2-12
Round-Robin Scheduling . . . 2-13
Burst Transfers . . . 2-13
Burst Adapters . . . 2-13

Interrupts 2-14
Individual Requests IRQ Scheme 2-15
Priority Encoded Interrupt Scheme 2-15
Assigning IRQs in SOPC Builder 2-16
Reset Distribution 2-16
Referenced Documents 2-17
Document Revision History 2-17

Chapter 3. System Interconnect Fabric for Streaming Interfaces

Introduction 3-1
High-Level Description 3-1
Avalon Streaming and Avalon Memory-Mapped Interfaces 3-2
Adapters 3-3
Data Format Adapter 3-4
Timing Adapter 3-4
Channel Adapter 3-4
Error Adapter 3-5
Multiplexer Examples 3-5
Example to Double Clock Frequency 3-5
Example to Double Data Width and Maintain Frequency 3-5
Example to Boost the Frequency 3-6
Referenced Documents 3-6
Document Revision History 3-7

Chapter 4. SOPC Builder Components

Component Providers 4-1
Component Hardware Structure 4-2
Components Inside the SOPC Builder System 4-3
Static HDL Components 4-3
Dynamic HDL Components 4-3
Components Outside the SOPC Builder System 4-3
Exported Connection Points—Conduit Interfaces 4-4
SOPC Builder Component Search Path 4-4
Installing Additional Components 4-5
Copy to the IP Root Directory 4-5
Reference Components in an .ipx File 4-6
Understanding IPX File Syntax 4-6
Upgrading from Earlier Versions 4-7
Component Structure 4-7
Component Description File (_hw.tcl) 4-7
Component File Organization 4-8
Classic Components in SOPC Builder 4-8
Referenced Documents 4-9
Document Revision History 4-9

Chapter 5. Using SOPC Builder with the Quartus II Software

Introduction 5-1
Quartus II IP File 5-1
Quartus II Incremental Compilation 5-1

TimeQuest Timing Analyzer 5-2
Analyzing PLLs 5-2
Analyzing Slow Asynchronous I/O Paths 5-3
Analyzing Single Data Rate SDRAM and SSRAM 5-4
Analyzing Tristate Bridges and Asynchronous Devices 5-6
Analyzing DDR and DDR2 Memories 5-6
Referenced Documents 5-7
Document Revision History 5-7

Chapter 6. Component Editor

Introduction 6-1
Component Hardware Structure 6-2
Starting the Component Editor 6-2
HDL Files Tab 6-2
Bottom-Up Design 6-3
Top-Down Design 6-3
Signals Tab 6-3
Naming Signals for Automatic Type and Interface Recognition 6-4
Templates for Interfaces to External Logic 6-5
Interfaces Tab 6-6
Component Wizard Tab 6-6
Identifying Information 6-6
Parameters 6-7
Saving a Component 6-8
Editing a Component 6-8
Software Assignments 6-8
Component GUI 6-8
Referenced Documents 6-9
Document Revision History 6-9

Chapter 7. Component Interface Tcl Reference

Introduction 7-1
Information in a Hardware Component Description File 7-1
Component Phases 7-2
Writing a Hardware Component Description File 7-2
Providing Basic Information 7-2
Declaring Parameters 7-3
Declaring Interfaces 7-6
Adding Files and Guiding Generation 7-6
Default Behaviors 7-7
Validation Phase Behavior 7-7
Elaboration Phase Behavior 7-7
Generation Phase Behavior 7-7
Editor Phase Behavior 7-8
Overriding Default Behaviors 7-9
Validation Callback 7-9
Elaboration Callback 7-10
Generation Callback 7-10
Editor Callback 7-11
Hardware Tcl Command Reference 7-12

Module Definition . . . 7-14
get_module_properties . . . 7-14
get_module_property . . . 7-15
set_module_property . . . 7-16
get_module_ports . . . 7-16
get_module_assignment . . . 7-16
set_module_assignment . . . 7-17
add_file . . . 7-17
get_files . . . 7-18
get_file_property . . . 7-18
set_file_property . . . 7-18
send_message . . . 7-19
Parameters . . . 7-19
get_parameter_properties . . . 7-19
add_parameter . . . 7-22
get_parameters . . . 7-22
get_parameter_property . . . 7-23
set_parameter_property . . . 7-23
get_parameter_value . . . 7-23
set_parameter_value . . . 7-24
decode_address_map . . . 7-24
add_display_item . . . 7-24
Interfaces and Ports . . . 7-25
add_interface . . . 7-25
get_interfaces . . . 7-26
get_interface_properties . . . 7-27
get_interface_property . . . 7-27
set_interface_property . . . 7-27
add_interface_port . . . 7-28
get_interface_ports . . . 7-28
get_port_properties . . . 7-29
get_port_property . . . 7-29
set_port_property . . . 7-30
get_interface_assignment . . . 7-30
set_interface_assignment . . . 7-31
Generation . . . 7-31
get_generation_properties . . . 7-31
get_generation_property . . . 7-32
get_project_property . . . 7-32
Referenced Document . . . 7-33
Document Revision History . . . 7-33

Chapter 8. Archiving SOPC Builder Projects

Introduction . . . 8-1
Limitations . . . 8-1
Required Files . . . 8-2
Referenced Documents . . . 8-3
Document Revision History . . . 8-3

Section II. Building Systems with SOPC Builder

Chapter 9. SOPC Builder Memory Subsystem Development Walkthrough

Introduction . . . 9-1
Example Design . . . 9-1
Example Design Structure . . . 9-2
Example Design Starting Point . . . 9-3
Hardware and Software Requirements . . . 9-3
Design Flow . . . 9-4
Component-Level Design in SOPC Builder . . . 9-4
SOPC Builder System-Level Design . . . 9-4
Simulation . . . 9-5
Quartus II Project-Level Design . . . 9-5
Board-Level Design . . . 9-5
Simulation Considerations . . . 9-5
Generic Memory Models . . . 9-5
Vendor-Specific Memory Models . . . 9-6
On-Chip RAM and ROM . . . 9-6
Component-Level Design for On-Chip Memory . . . 9-6
Memory Type . . . 9-6
Size . . . 9-7
Read Latency . . . 9-7
Non-Default Memory Initialization . . . 9-7
Enable In-System Memory Content Editor Feature . . . 9-8
SOPC Builder System-Level Design for On-Chip Memory . . . 9-8
Simulation for On-Chip Memory . . . 9-8
Quartus II Project-Level Design for On-Chip Memory . . . 9-8
Board-Level Design for On-Chip Memory . . . 9-8
Example Design with On-Chip Memory . . . 9-8
EPCS Serial Configuration Device . . . 9-9
Component-Level Design for an EPCS Device . . . 9-9
SOPC Builder System-Level Design for an EPCS Device . . . 9-9
Simulation for an EPCS Device . . . 9-10
Quartus II Project-Level Design for an EPCS Device . . . 9-10
Board-Level Design for an EPCS Device . . . 9-10
Example Design with an EPCS Device . . . 9-10
SDR SDRAM . . . 9-11
Component-Level Design for SDRAM . . . 9-11
SOPC Builder System-Level Design for SDRAM . . . 9-11
Simulation for SDRAM . . . 9-12
Quartus II Project-Level Design for SDRAM . . . 9-12
Connecting and Assigning the SDRAM-Related Pins . . . 9-12
Accommodating Clock Skew . . . 9-12
Board-Level Design for SDRAM . . . 9-13
Example Design with SDR SDRAM . . . 9-13
DDR SDRAM . . . 9-14
DDR2 SDRAM . . . 9-14

Off-Chip SRAM and Flash Memory . . . 9-15
Component-Level Design for SRAM and Flash Memory . . . 9-15
Avalon-MM Tristate Bridge . . . 9-16
Flash Memory . . . 9-16
SRAM . . . 9-17
SOPC Builder System-Level Design for SRAM and Flash Memory . . . 9-18
Simulation for SRAM and Flash Memory . . . 9-18
Quartus II Project-Level Design for SRAM and Flash Memory . . . 9-18
Board-Level Design for SRAM and Flash Memory . . . 9-19
Aligning the Least-Significant Address Bits . . . 9-19
Aligning the Most-Significant Address Bits . . . 9-20
Example Design with SRAM and Flash Memory . . . 9-21
Adding the Avalon-MM Tristate Bridge . . . 9-21
Adding the Flash Memory Interface . . . 9-21
Adding the SRAM Interface . . . 9-21
SOPC Builder System Contents Tab . . . 9-21
Connecting and Assigning Pins in the Quartus II Project . . . 9-22
Connecting FPGA Pins to Devices on the Board . . . 9-23
Referenced Documents . . . 9-25
Document Revision History . . . 9-26

Chapter 10. SOPC Builder Component Development Walkthrough
Introduction . . . 10-1
SOPC Builder Components and the Component Editor . . . 10-1
Prerequisites . . . 10-1
Hardware and Software Requirements . . . 10-2
Component Development Flow . . . 10-2
Typical Design Steps . . . 10-2
Hardware Design . . . 10-3
Design Example: Checksum Hardware Accelerator . . . 10-4
Software Design . . . 10-5
Verifying the Component . . . 10-6
System Console . . . 10-6
System-Level Verification . . . 10-7
Sharing Components . . . 10-7
.sopcinfo Files . . . 10-7
Referenced Documents . . . 10-8
Document Revision History . . . 10-9

Section III. Interconnect Components

Chapter 11. Avalon Memory-Mapped Bridges
Introduction to Bridges . . . 11-1
Structure of a Bridge . . . 11-1
Reasons for Using a Bridge . . . 11-2
Address Mapping for Systems with Avalon-MM Bridges . . . 11-5
Tools for Visualizing the Address Map . . . 11-7
Differences between Avalon-MM Bridges and Avalon-MM Tristate Bridges . . . 11-7

Avalon-MM Pipeline Bridge 11-7
Component Overview 11-7
Functional Description 11-8
Interfaces 11-8
Pipeline Stages and Effects on Latency 11-9
Burst Support 11-9
Example System with Avalon-MM Pipeline Bridges 11-9
Clock Crossing Bridge 11-10
Choosing Clock Crossing Methodology 11-10
Functional Description 11-11
Interfaces 11-11
Clock Crossing Bridge and FIFOs 11-12
Burst Support 11-12
Example System with Avalon-MM Clock-Crossing Bridges 11-13
Instantiating the Avalon-MM Clock-Crossing Bridge in SOPC Builder 11-14
Clock Domain Crossing Logic 11-15
Description of Clock Domain Adapter 11-15
Location of Clock Domain Adapter 11-16
Duration of Transfers Crossing Clock Domains 11-17
Implementing Multiple Clock Domains in SOPC Builder 11-17
Avalon-MM DDR Memory Half-Rate Bridge 11-18
Resource Usage and Performance 11-19
Functional Description 11-19
Instantiating the Core in SOPC Builder 11-20
Example System 11-21
Device Support 11-22
Hardware Simulation Considerations 11-22
Software Programming Model 11-22
Referenced Documents 11-22
Document Revision History 11-23

Chapter 12. Avalon Streaming Interconnect Components

Introduction to Interconnect Components 12-1
Interconnect Component Usage 12-1
Address Mapping 12-3
Timing Adapter 12-3
Resource Usage and Performance 12-4
Instantiating the Timing Adapter in SOPC Builder 12-4
Data Format Adapter 12-6
Resource Usage and Performance 12-6
Instantiating the Data Format Adapter in SOPC Builder 12-7
Channel Adapter 12-8
Resource Usage and Performance 12-8
Instantiating the Channel Adapter in SOPC Builder 12-8
Error Adapter 12-9
Instantiating the Error Adapter in SOPC Builder 12-9
Installation and Licensing 12-10
Hardware Simulation Considerations 12-10
Software Programming Model 12-10
Referenced Documents 12-11
Document Revision History 12-11

Additional Information

About this Handbook Info-1
How to Contact Altera Info-1
Third-Party Software Product Information Info-1
Typographic Conventions Info-2

Chapter Revision Dates

The chapters in this book, *Quartus II Handbook Version 9.0 Volume 4: SOPC Builder*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1 Introduction to SOPC Builder
Revised: *March 2009*
Part Number: *QII54001-9.0.0*

Chapter 2 System Interconnect Fabric for Memory-Mapped Interfaces
Revised: *March 2009*
Part Number: *QII54003-9.0.0*

Chapter 3 System Interconnect Fabric for Streaming Interfaces
Revised: *March 2009*
Part Number: *QII54019-9.0.0*

Chapter 4 SOPC Builder Components
Revised: *March 2009*
Part Number: *QII54004-9.0.0*

Chapter 5 Using SOPC Builder with the Quartus II Software
Revised: *March 2009*
Part Number: *QII54023-9.0.0*

Chapter 6 Component Editor
Revised: *March 2009*
Part Number: *QII54005-9.0.0*

Chapter 7 Component Interface Tcl Reference
Revised: *March 2009*
Part Number: *QII54022-9.0.0*

Chapter 8 Archiving SOPC Builder Projects
Revised: *March 2009*
Part Number: *QII54017-9.0.0*

Chapter 9 SOPC Builder Memory Subsystem Development Walkthrough
Revised: *March 2009*
Part Number: *QII54006-9.0.0*

Chapter 10 SOPC Builder Component Development Walkthrough
Revised: *March 2009*
Part Number: *QII54007-9.0.0*

Chapter 11 Avalon Memory-Mapped Bridges
Revised: *March 2009*
Part Number: *QII54020-9.0.0*

Chapter 12 Avalon Streaming Interconnect Components
Revised: *March 2009*
Part Number: *QII54021-9.0.0*

Section I. SOPC Builder Features

This section introduces the SOPC Builder system integration tool. Chapters in this section answer the following questions:

- What is SOPC Builder?
- What features does SOPC Builder provide?

This section includes the following chapters:

- Chapter 1, Introduction to SOPC Builder
- Chapter 2, System Interconnect Fabric for Memory-Mapped Interfaces
- Chapter 3, System Interconnect Fabric for Streaming Interfaces
- Chapter 4, SOPC Builder Components
- Chapter 5, Using SOPC Builder with the Quartus II Software
- Chapter 6, Component Editor
- Chapter 7, Component Interface Tcl Reference
- Chapter 8, Archiving SOPC Builder Projects

☞ For information about the revision history for chapters in this section, refer to each individual chapter's revision history.

1. Introduction to SOPC Builder

QII54001-9.0.0

Quick Start Guide

For a quick introduction on how to use SOPC Builder, follow these general steps:

- Install the Quartus® II software, which includes SOPC Builder. This is available at www.altera.com.
- Take advantage of the one-hour online course, *Using SOPC Builder*.
- Download and run the checksum sample design described in the *SOPC Builder Memory Subsystem Development Walkthrough* chapter in volume 4 of the *Quartus II Handbook*.

Overview

SOPC Builder is a powerful system development tool. SOPC Builder enables you to define and generate a complete system-on-a-programmable-chip (SOPC) in much less time than using traditional, manual integration methods. SOPC Builder is included as part of the Quartus II software.

You may have used SOPC Builder to create systems based on the Nios® II processor. However, SOPC Builder is more than a Nios II system builder; it is a general-purpose tool for creating systems that may or may not contain a processor and may include a soft processor other than the Nios II processor.

SOPC Builder automates the task of integrating hardware components. Using traditional design methods, you must manually write HDL modules to wire together the pieces of the system. Using SOPC Builder, you specify the system components in a GUI and SOPC Builder generates the interconnect logic automatically. SOPC Builder generates HDL files that define all components of the system, and a top-level HDL file that connects all the components together. SOPC Builder generates either Verilog HDL or VHDL equally.

In addition to its role as a system generation tool, SOPC Builder provides features to ease writing software and to accelerate system simulation. This chapter includes the following sections:

- “Architecture of SOPC Builder Systems” on page 1–2
- “Functions of SOPC Builder” on page 1–5
- “Operating System Support” on page 1–6
- “Talkback Support” on page 1–7

Architecture of SOPC Builder Systems

An SOPC Builder component is a design module that SOPC Builder recognizes and can automatically integrate into a system. You can also define and add custom components or select from a list of provided components. SOPC Builder connects multiple modules together to create a top-level HDL file called the SOPC Builder system. SOPC Builder generates system interconnect fabric that contains logic to manage the connectivity of all modules in the system.

SOPC Builder Modules

This document refers to *components* as the class definition for a module, while *module* is the instance of the component class.

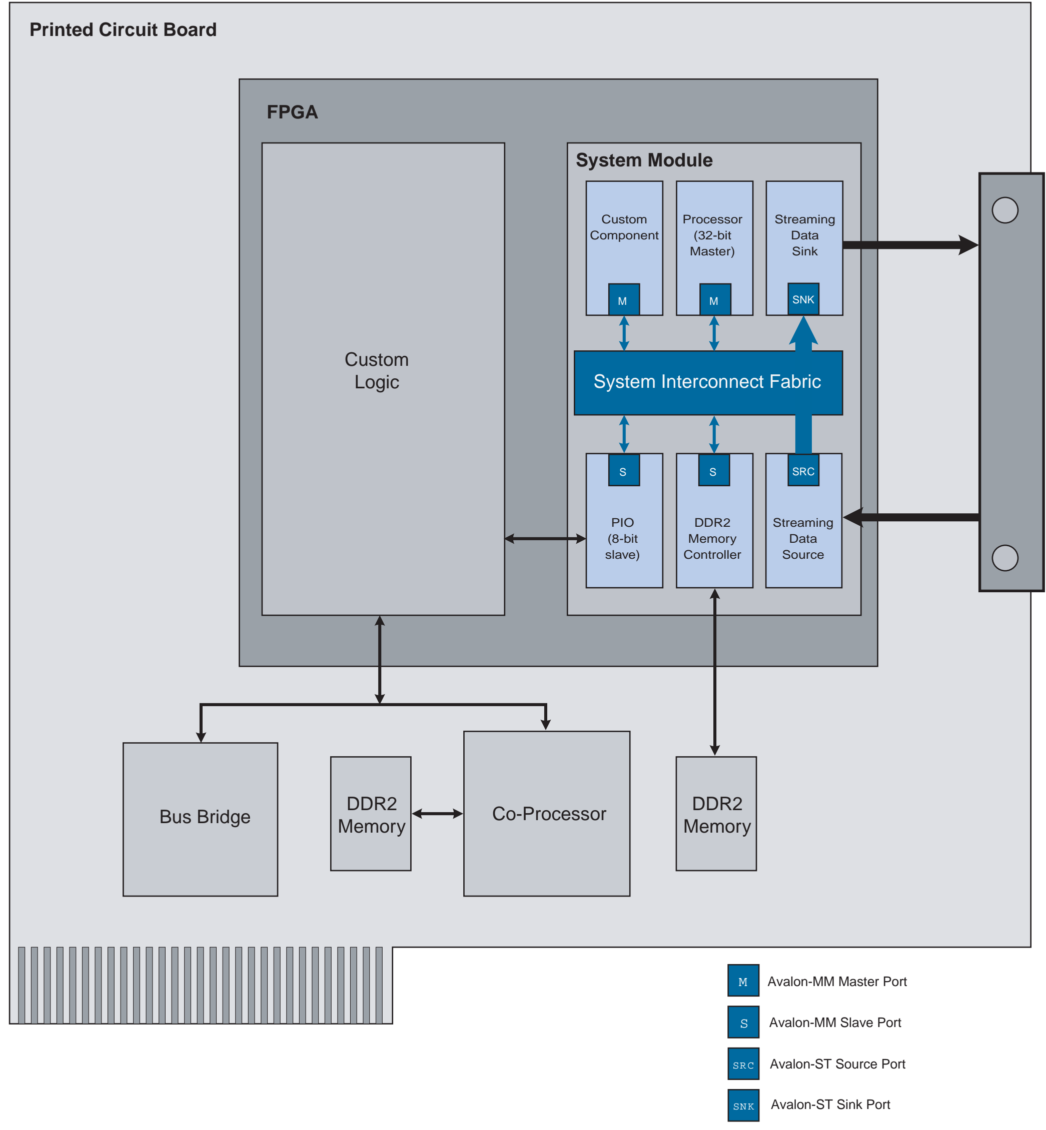
SOPC Builder modules are the building blocks for creating an SOPC Builder system. SOPC Builder modules use Avalon® interfaces, such as memory-mapped, streaming, and IRQ, for the physical connection of components. You can use SOPC Builder to connect any logical device (either on-chip or off-chip) that has an Avalon interface. There are different types of Avalon interfaces, as described in the *Avalon Interface Specifications*.

For details on the Avalon-MM interface refer to *System Interconnect Fabric for Memory-Mapped Interfaces* in chapter in volume 4 of the *Quartus II Handbook*. For details on the Avalon-ST interface, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*. For details about the Avalon-ST interface protocol, refer to *Avalon Interface Specifications*.

Example System

Figure 1–1 shows an FPGA design that includes an SOPC Builder system and custom logic modules. You can integrate custom logic inside or outside the SOPC Builder system. In this example, the custom component inside the SOPC Builder system communicates with other modules through an Avalon-MM master interface. The custom logic outside of the SOPC Builder system is connected to the SOPC Builder system through a PIO interface. The SOPC Builder system includes two SOPC Builder components with Avalon-ST source and sink interfaces. The system interconnect fabric connects all of the SOPC Builder components using the Avalon-MM or Avalon-ST system interconnect as appropriate.

Figure 1–1. Example of an FPGA with a SOPC Builder System Generated by SOPC Builder



A component can be a logical device that is entirely contained within the SOPC Builder system, such as the processor component shown in Figure 1–1. Alternately, a component can act as an interface to an off-chip device, such as the DDR2 interface component in Figure 1–1. In addition to the Avalon interface, a component can have other signals that connect to logic outside the SOPC Builder system. Non-Avalon signals can provide a special-purpose interface to the SOPC Builder system, such as the PIO in Figure 1–1. These non-Avalon signals are described in *Conduit Interface* chapter in the *Avalon Interface Specifications.*

Available Components

Altera and third-party developers provide ready-to-use SOPC Builder components, including:

- Microprocessors, such as the Nios II processor

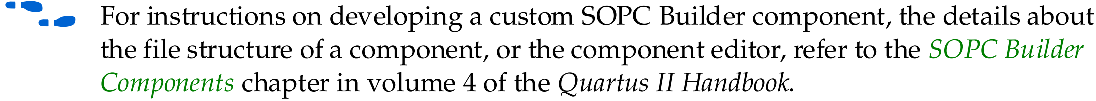
- Microcontroller peripherals, such as a Scatter-Gather DMA Controller and timer
- Serial communication interfaces, such as a UART and a serial peripheral interface (SPI)
- General purpose I/O
- Communications peripherals, such as a 10/100/1000 Ethernet MAC
- Interfaces to off-chip devices

Custom Components

You can import HDL modules and entities that you write using Verilog HDL or VHDL into SOPC builder as custom components. You use the following design flow to integrate custom logic into an SOPC Builder system:

1. Determine the interfaces used to interact with your custom component.
2. Create the component logic using either Verilog HDL or VHDL.
3. Use the SOPC Builder component editor to create an SOPC Builder component with your HDL files.
4. Instantiate your component in the system.

Once you have created an SOPC Builder component, you can use the component in other SOPC Builder systems, and share the component with other design teams.



For instructions on developing a custom SOPC Builder component, the details about the file structure of a component, or the component editor, refer to the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*.

For further details, refer to the *System Interconnect Fabric for Memory-Mapped Interfaces* and *System Interconnect Fabric for Streaming Interfaces* chapters in volume 4 of the *Quartus II Handbook*.

Third-Party Components

You can also use SOPC-ready components that were developed by third-parties. Altera awards the SOPC Builder Ready certification to IP functions that are ready to integrate with the Nios II embedded processor or the system interconnect fabric via SOPC Builder. These cores support the Avalon-MM interface or the Avalon Streaming (Avalon-ST) interface and include constraints, software drivers, and simulation models when applicable.

To find third-party components that you can purchase and use in SOPC Builder systems, complete the following steps:

1. On the Tools menu in SOPC Builder, click **Download Components**.
2. On the Intellectual Property Solutions web page, type `SOPC Builder ready` ↵ in the box labeled **Search for IP, Development Kits and Reference Designs**.

Functions of SOPC Builder

This section describes the functions of SOPC Builder.

Defining and Generating the System Hardware

SOPC Builder allows you to design the structure of a hardware system. The GUI allows you to add components to a system, configure the components, and specify connectivity.

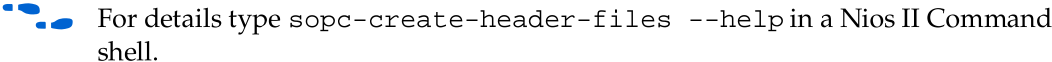
After you add and parameterize components, SOPC Builder generates the system interconnect fabric, and outputs HDL files to your project directory. During system generation, SOPC Builder creates the following items:

- An HDL file for the top-level SOPC Builder system and for each component in the system. The top-level HDL file is named **<*system_name*>v** for Verilog HDL designs and **<*system_name*>.vhd** for VHDL designs.
- Synopsis Design Constraints file (**.sdc**) for timing analysis.
- A Block Symbol File (**.bsf**) representation of the top-level SOPC Builder system for use in Quartus II Block Diagram Files (**.bdf**).
- An example of an instance of the top-level HDL file, **<*SOPC_project_name_inst*>.v** or **<*SOPC_project_name_inst*>.vhd**, which demonstrates how to instantiate the top-level HDL file in your code.
- A data sheet called <*system_name*>**.html** that provides a system overview including the following information:
  - All external connections for the system
  - A memory map showing the address of each Avalon-MM slave with respect to each Avalon-MM master to which it is connected
  - All parameter assignments for each component
- A functional test bench for the SOPC Builder system and ModelSim® simulation project files
- SOPC Builder information file (**.sopcinfo)** that describes all of the components and connections in your system. This file is a complete system description, and is used by downstream tools such as the Nios II tool chain. It also describes the parameterization of each component in the system; consequently, you can parse its contents to get requirements when developing software drivers for SOPC Builder components.
- A Quartus II IP File (**.qip**) that provides the Quartus II software with all required information about your SOPC Builder system. The **.qip** file includes references to the following information:
  - HDL files used in the SOPC Builder system
  - TimeQuest Timing Analyzer Synopsys Design Constraint (**.sdc**) files
  - Component definition files for archiving purposes

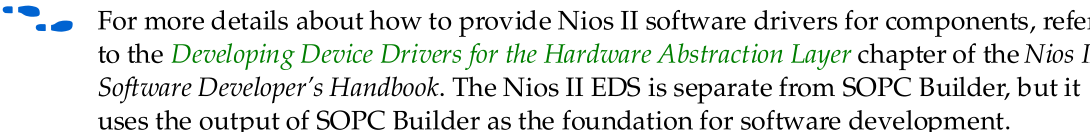
After you generate the SOPC Builder system, you can compile it with the Quartus II software, or you can instantiate it in a larger FPGA design.

Creating a Memory Map for Software Development

When your SOPC Builder system includes a Nios II processor, SOPC Builder generates a header file, **cpu.h**, that provides the base address of each Avalon-MM slave component. In addition, each slave component can provide software drivers and other software functions and libraries for the processor. You can create C header files for your system using the `sopc-create-header-files` utility.



For details type `sopc-create-header-files --help` in a Nios II Command shell.



For more details about how to provide Nios II software drivers for components, refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*. The Nios II EDS is separate from SOPC Builder, but it uses the output of SOPC Builder as the foundation for software development.

Creating a Simulation Model and Test Bench

You can simulate your system after generating it with SOPC Builder. During system generation, SOPC Builder outputs a simulation test bench and a ModelSim setup script that eases the system simulation effort. The test bench does the following:

- Instantiates the SOPC Builder system
- Drives all clocks and resets
- Instantiates simulation models for off-chip devices when available

Visualization of Large SOPC Builder Systems

For large systems, you can use the **Filters** dialog box to customize the display of your system in the connections panel. You can filter the display of your system by interface type, module name, interface type, or using custom tags. For example, you can use filtering to view only components that include an Avalon-MM interface or components that are connected to a particular Nios II processor. For more information, refer to Quartus II online Help.

Operating System Support

SOPC Builder supports all of the operating systems that the Quartus II software supports.

For more information refer to *Quartus II Installation & Licensing for Windows and Linux Workstations.*

© March 2009 Altera Corporation

Talkback Support

Talkback is a Quartus II software feature that provides feedback to Altera on tool and IP feature usage. Altera uses the data to help guide future product planning efforts. Talkback sends Altera information on the components used, interface types, interface properties, parameter names and values, clocking, and software assignments. The Talkback file does not include information about system connectivity, interrupts or the memory map seen by each master in the system. When problems arise in the Quartus II software, Talkback data also helps Altera find and fix the cause.

The Talkback feature is enabled by default. You can disable Talkback if you do not wish to share your tool usage data with Altera.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Component Editor* chapter in volume 4 of the *Quartus II Handbook*
- *Conduit Interface* chapter in the *Avalon Interface Specification*
- *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Nios II Hardware Development Tutorial*
- *SOPC Builder Components* chapter in volume 4 *of the Quartus II Handbook*
- *System Interconnect Fabric for Memory-Mapped Interfaces* chapter in volume 4 of the *Quartus II Handbook*
- *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*

Document Revision History

Table 1–1 shows the revision history for this chapter.

Table 1–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009, v9.0.0	■ Added sopc-create-header-files command ■ Added description of Generate HTML Data Sheet ■ Added instructions for downloading third-party IP. ■ Named top-level HDL system files that SOPC Builder generates. ■ Added paragraph introducing the filtering for visualization of large systems.	Updated to reflect new functionality in the 9.0 release.
November 2008, v8.1.0	■ Expanded description of **.sopcinfo** file ■ Changed page size to 8.5 x 11 inches	—
May 2008, v8.0.0	■ Updated references to Avalon Memory-Mapped and Streaming Interface Specifications and changed to Avalon Interface Specifications. ■ Add Quick Start Guide. ■ Add list of OS support.	The two specifications have been combined into one for all Avalon interfaces.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

 © March 2009 Altera Corporation

2. System Interconnect Fabric for Memory-Mapped Interfaces

QII54003-9.0.0

Introduction

The system interconnect fabric for memory-mapped interfaces is a high-bandwidth interconnect structure for connecting components that use the Avalon® Memory-Mapped (Avalon-MM) interface. The system interconnect fabric consumes minimal logic resources, provides greater flexibility, and higher throughput than a typical shared system bus. It is a cross-connect fabric and not a tristated or time domain multiplexed bus. This chapter describes the functions of system interconnect fabric for memory-mapped interfaces and the implementation of those functions.

High-Level Description

The system interconnect fabric is the collection of interconnect and logic resources that connects Avalon-MM master and slaves on components in a system. SOPC Builder generates the system interconnect fabric to match the needs of the components in a system. The system interconnect fabric implements the connection details of a system. It guarantees that signals are routed correctly between master and slaves, as long as the ports adhere to the rules of the *Avalon Interface Specifications*. This chapter provides information on the following topics:

- "Address Decoding" on page 2–4
- "Datapath Multiplexing" on page 2–5
- "Wait State Insertion" on page 2–5
- "Pipelined Read Transfers" on page 2–6
- "Dynamic Bus Sizing and Native Address Alignment" on page 2–7
- "Arbitration for Multimaster Systems" on page 2–9
- "Burst Adapters" on page 2–13
- "Interrupts" on page 2–14
- "Reset Distribution" on page 2–16

For details about the Avalon-MM interface, refer to the *Avalon Interface Specifications*.

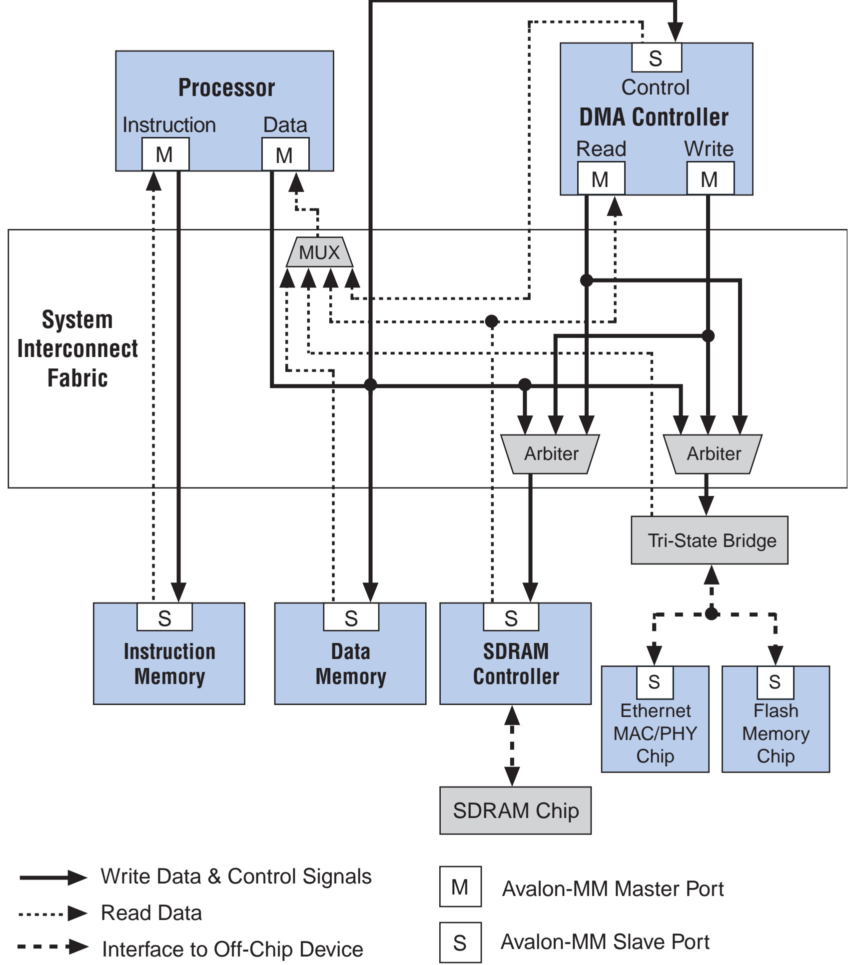
System interconnect fabric for memory-mapped interfaces supports the following items:

- Any number of master and slave components. The master-to-slave relationship can be one-to-one, one-to-many, many-to-one, or many-to-many.
- On-chip components.
- Interfaces to off-chip devices.
- Master and slaves of different data widths.
- Components operating in different clock domains.
- Components using multiple Avalon-MM ports.

Figure 2–1 shows a simplified diagram of the system interconnect fabric in an example memory-mapped system with multiple masters.

All figures in this chapter are simplified to show only the particular function being discussed. In a complete system, the system interconnect fabric might alter the address, data, and control paths beyond what is shown in any one particular figure.

Figure 2–1. System Interconnect Fabric—Example System



SOPC Builder supports components with multiple Avalon-MM interfaces, such as the processor component shown in Figure 2–1. Because SOPC Builder can create system interconnect fabric to connect components with multiple interfaces, you can create complex interfaces that provide more functionality than a single Avalon-MM interface. For example, you can create a component with two different Avalon-MM slaves, each with an associated interrupt interface.

System interconnect fabric can connect any combination of components, as long as each interface conforms to the *Avalon Interface Specifications*. It can, for example, connect a system comprised of only two components with unidirectional dataflow between them. Avalon-MM interfaces are suitable for random address transactions, such as to memories or embedded peripherals.

Generating system interconnect fabric is SOPC Builder's primary purpose. In most cases, you are not required to modify the generated HDL; however, a basic understanding of how HDL works can help you optimize your system. For example, knowledge of the arbitration algorithm can help designers of multimaster systems minimize the impact of arbitration on the system throughput.

Fundamentals of Implementation

System interconnect fabric for memory-mapped interfaces implements a partial crossbar interconnect structure that provides concurrent paths between master and slaves. System interconnect fabric consists of synchronous logic and routing resources inside the FPGA.

For each component interface, system interconnect fabric manages Avalon-MM transfers, interacting with signals on the connected component. Master and slave interfaces can contain different signals and the system interconnect fabric handle any adaptation necessary between them. In the path between master and slaves, the system interconnect fabric might introduce registers for timing synchronization, finite state machines for event sequencing, or nothing at all, depending on the services required by the specific interfaces.

For more information, refer to the *Avalon Memory-Mapped Design Optimizations* chapter in the *Embedded Design Handbook*.

Functions of System Interconnect Fabric

System interconnect fabric logic provides the following functions:

- "Address Decoding" on page 2–4
- "Datapath Multiplexing" on page 2–5
- "Wait State Insertion" on page 2–5
- "Pipelined Read Transfers" on page 2–6
- "Arbitration for Multimaster Systems" on page 2–9
- "Burst Adapters" on page 2–13
- "Interrupts" on page 2–14
- "Reset Distribution" on page 2–16

The behavior of these functions in a specific SOPC Builder system depends on the design of the components in the system and the settings made in SOPC Builder. The remaining sections of this chapter describe how SOPC Builder implements each function.

Address Decoding

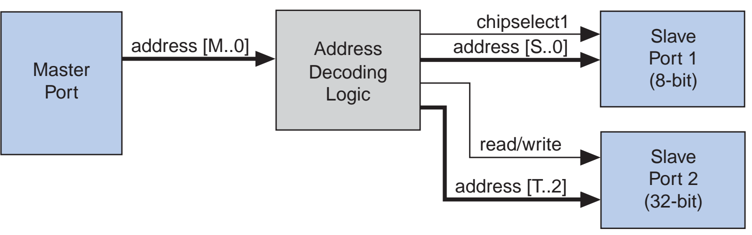
Address decoding logic in the system interconnect fabric forwards an appropriate address and produces a chipselect signal for each slave. Address decoding logic simplifies component design in the following ways:

- The system interconnect fabric selects a slave whenever it is being addressed by a master. Slave components do not need to decode the address to determine when they are selected.
- Slave addresses are properly aligned to the slave interface.
- Changing the system memory map does not involve manually editing HDL.

Figure 2–2 shows a block diagram of the address-decoding logic for one master and two slaves. Separate address-decoding logic is generated for every master in a system.

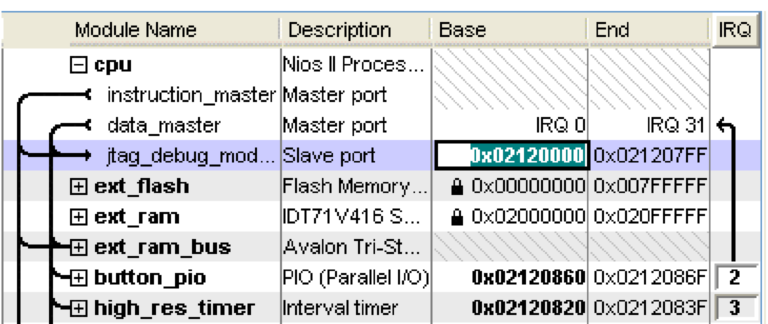
As Figure 2–2 shows, the address decoding logic handles the difference between the master address width (<*M*>) and the individual slave address widths (<*S*> and <*T*>). It also maps only the necessary master address bits to access words in each slave's address space.

Figure 2–2. Block Diagram of Address Decoding Logic



In SOPC Builder, the user-configurable aspects of address decoding logic are controlled by the **Base** setting in the list of active components on the **System Contents** tab, as shown in Figure 2–3.

Figure 2–3. Base Settings in SOPC Builder Control Address Decoding

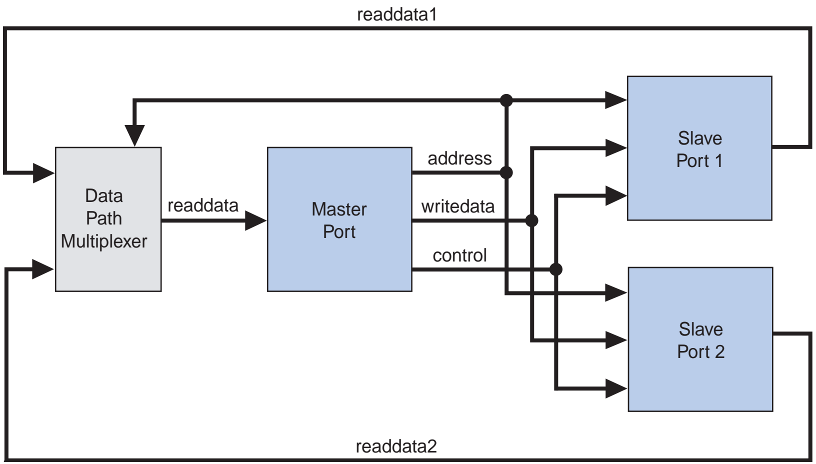


Datapath Multiplexing

Datapath multiplexing logic in the system interconnect fabric drives the `writedata` signal from the granted master to the selected slave, and the `readdata` signal from the selected slave back to the requesting master.

Figure 2–4 shows a block diagram of the datapath multiplexing logic for one master and two slaves. SOPC Builder generates separate datapath multiplexing logic for every master in the system.

Figure 2–4. Block Diagram of Datapath Multiplexing Logic



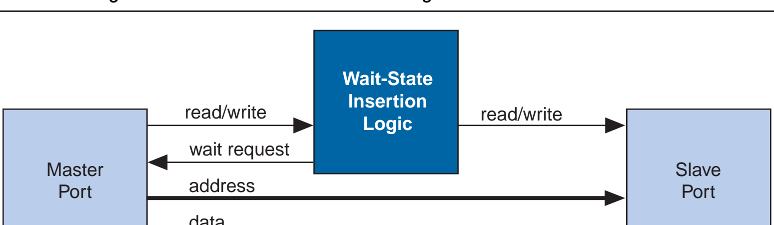
In SOPC Builder, the generation of datapath multiplexing logic is specified using the connections panel on the **System Contents** tab.

Wait State Insertion

Wait states extend the duration of a transfer by one or more cycles. Wait state insertion logic accommodates the timing needs of each slave, and causes the master to wait until the slave can proceed. System interconnect fabric inserts wait states into a transfer when the target slave cannot respond in a single clock cycle. System interconnect fabric also inserts wait states in cases when slave `read_enable` and `write_enable` signals have setup or hold time requirements.

Wait state insertion logic is a small finite-state machine that translates control signal sequencing between the slave side and the master side. Figure 2–5 shows a block diagram of the wait state insertion logic between one master and one slave.

Figure 2–5. Block Diagram of Wait State Insertion Logic



System interconnect fabric can force a master to wait for several reasons in addition to the wait state needs of a slave. For example, arbitration logic in a multimaster system can force a master to wait until it is granted access to a slave.

SOPC Builder generates wait state insertion logic based on the properties of all slaves in the system.

Pipelined Read Transfers

The Avalon-MM interface supports pipelined read transfers, allowing a pipelined master to start multiple read transfers in succession without waiting for the prior transfers to complete. Pipelined transfers allow master-slave pairs to achieve higher throughput, even though the slave requires one or more cycles of latency to return data for each transfer.

SOPC Builder generates system interconnect fabric with pipeline management logic to take advantage of pipelined components wherever possible, based on the pipeline properties of each master-slave pair in the system. Regardless of the pipeline latency of a target slave, SOPC Builder guarantees that read data arrives at each master in the order requested. Because master and slaves often have mismatched pipeline latency, system interconnect fabric often contains logic to reconcile the differences. Many cases of pipeline latency are possible, as shown in Table 2–1.

Table 2–1. Various Cases of Pipeline Latency in a Master-Slave Pair

Master	Slave	Pipeline Management Logic Structure
No pipeline	No pipeline	The system interconnect fabric does not instantiate logic to handle pipeline latency.
No pipeline	Pipelined with fixed or variable latency	The system interconnect fabric forces the master to wait through any slave-side latency cycles. This master-slave pair gains no benefits of pipelining, because the master waits for each transfer to complete before beginning a new transfer. However, while the master is waiting, the slave can accept transfers from a different master.
Pipelined	No pipeline	The system interconnect fabric carries out the transfer as if neither master nor slave were pipelined, causing the master to wait until the slave returns data.
Pipelined	Pipelined with fixed latency	The system interconnect fabric allows the master to capture data at the exact clock cycle when data from the slave is valid. This process enables the master-slave pair to achieve maximum throughput performance.
Pipelined	Pipelined with variable latency	This is the simplest pipelined case, in which the slave asserts a signal when its `readdata` is valid, and the master captures the data. This case enables this master-slave pair to achieve maximum throughput.

SOPC Builder generates logic to handle pipeline latency based on the properties of the master and slaves in the system. When configuring a system in SOPC Builder, there are no settings that directly control the pipeline management logic in the system interconnect fabric.

Dynamic Bus Sizing and Native Address Alignment

SOPC Builder generates system interconnect fabric to accommodate master and slaves with unmatched data widths. Address alignment affects how slave data is aligned in a master's address space, in the case that the master and slave data widths are different. Address alignment is a property of each slave, and can be different for each slave in a system. A slave can declare itself to use one of the following:

- Dynamic bus sizing
- Native address alignment

The following sections explain the implications of the address alignment property slave devices.

Dynamic Bus Sizing

Dynamic bus sizing hides the details of interfacing a narrow component device to a wider master, and vice versa. When an <*N*>-bit master accesses a slave with dynamic bus sizing, the master operates exclusively on full <*N*>-bit words of data, without awareness of the slave data width.

When using dynamic bus sizing, the slave data width in units of bytes must be a power of two.

Dynamic bus sizing provides the following benefits:

- Eliminates the need to create address-alignment hardware manually.
- Reduces design complexity of the master component.
- Enables any master to access any memory device, regardless of the data width.

In the case of dynamic bus sizing, the system interconnect fabric includes a small finite state machine that reconciles the difference between master and slave data widths. The behavior is different depending on whether the master data width is wider or narrower than the slave.

Wider Master

In the case of a wider master, the dynamic bus-sizing logic accepts a single, wide transfer on the master side, and then performs multiple narrow transfers on the slave side. For a data-width ratio of <*N*>:1, the dynamic bus-sizing logic generates up to <*N*> slave transfers for each master transfer. The master waits while multiple slave-side transfers complete; the master transfer ends when all slave-side transfers end.

Dynamic bus-sizing logic uses the master-side byte-enable signals to generate appropriate slave transfers. The dynamic bus-sizing logic performs as many slave-side transfers as necessary to write or read the specified byte lanes.

Narrower Master

In the case of a narrower master, one transfer on the master side generates one transfer on the slave side. In this case, multiple master word addresses map to a single offset in the slave memory space. The dynamic bus-sizing logic maps each master address to a subset of byte lanes in the appropriate slave offset. All bytes of the slave memory are accessible in the master address space.

Table 2–2 demonstrates the case of a 32-bit master accessing a 64-bit slave with dynamic bus sizing. In the table, offset refers to the offset into the slave memory space.

Table 2–2. 32-Bit Master View of 64-Bit Slave with Dynamic Bus Sizing

32-bit Address	Data
0×00000000 (word 0)	$OFFSET[0]_{31..0}$
0×00000004 (word 1)	$OFFSET[0]_{63..32}$
0×00000008 (word 2)	$OFFSET[1]_{31..0}$
0×0000000C (word 3)	$OFFSET[1]_{63..32}$

In the case of a read transfer, the dynamic bus-sizing logic multiplexes the appropriate byte lanes of the slave data to the narrow master. In the case of a write transfer, the dynamic bus-sizing logic uses slave-side byte-enable signals to write only to the appropriate byte lanes.

Altera recommends that you select dynamic bus sizing whenever possible. Dynamic bus sizing offers more flexibility when the master and slave components in your system have different widths.

Native Address Alignment

Table 2–3 demonstrates native address alignment and dynamic bus sizing for a 32-bit master connected to a 16-bit slave (a 2:1 ratio). In this example, the slave is mapped to base address <*BASE*> in the master's address space. In Table 2–3, `OFFSET` refers to the offset into the 16-bit slave address space.

Table 2–3. 32-Bit Master View of 16-Bit Slave Data

32-bit Master Address	Data with Native Alignment	Data with Dynamic Bus Sizing
BASE + 0x0 (word 0)	0×0000:OFFSET[0]	OFFSET[1]:OFFSET[0]
BASE + 0x4 (word 1)	0×0000:OFFSET[1]	OFFSET[3]:OFFSET[2]
BASE + 0x8 (word 2)	0×0000:OFFSET[2]	OFFSET[5]:OFFSET[4]
BASE + 0xC (word 3)	0×0000:OFFSET[3]	OFFSET[7]:OFFSET[6]
...	...	...
BASE + 4*N* (word *N*)	0×0000:OFFSET[N]	OFFSET[2N+1]:OFFSET[2N]

SOPC Builder generates appropriate address-alignment logic based on the properties of the master and slaves in the system. When configuring a system in SOPC Builder, there are no settings that directly control the address alignment in the system interconnect fabric.

Arbitration for Multimaster Systems

System interconnect fabric supports systems with multiple master components. In a system with multiple masters, such as the system pictured in Figure 2–1 on page 2–2, the system interconnect fabric provides shared access to slaves using a technique called slave-side arbitration. Slave-side arbitration moves the arbitration logic close to the slave, such that the algorithm that determines which master gains access to a specific slave in the event that multiple masters attempt to access the same slave at the same time.

The multimaster architecture used by system interconnect fabric offers the following benefits:

- Eliminates having to create arbitration hardware manually.
- Allows multiple masters to transfer data simultaneously. Unlike traditional host-side arbitration architectures where each master must wait until it is granted access to the shared bus, multiple Avalon-MM masters can simultaneously perform transfers with independent slaves. Arbitration logic stalls a master only when multiple masters attempt to access the same slave during the same cycle.
- Eliminates unnecessary master-slave connections. The connection between a master and a slave exists only if it is specified in SOPC Builder. If a master never initiates transfers to a specific slave, no connection is necessary, and therefore SOPC Builder does not waste logic resources to connect the two ports.
- Provides configurable arbitration settings, and arbitration for each slave is specified independently. For example, you can grant one master more arbitration shares than others, allowing it to gain more access cycles to the slave. The arbitration share settings are defined for each slave independently.
- Simplifies master component design. The details of arbitration are encapsulated inside the system interconnect fabric. Each Avalon-MM master connects to the system interconnect fabric as if it is the only master in the system. As a result, you can reuse a component in single-master and multimaster systems without requiring design changes to the component.

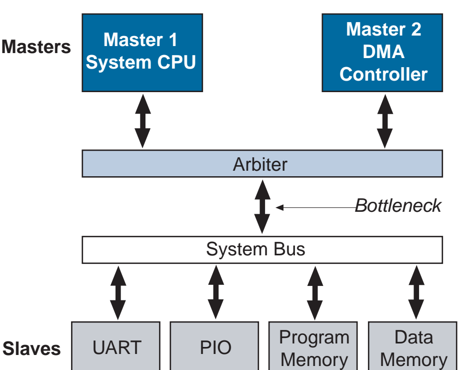
Traditional Shared Bus Architectures

This section discusses the architecture of the system interconnect fabric generated by SOPC Builder for multimaster systems. As a frame of reference for the discussion of multiple masters and arbitration, this section describes traditional bus architectures.

In traditional bus architectures, one or more bus masters and bus slaves connect to a shared bus, consisting of wires on a printed circuit board or on-chip routing. A single arbiter controls the bus (that is, the path between bus masters and bus slaves), so that multiple bus masters do not simultaneously drive the bus. Each bus master requests control of the bus from the arbiter, and the arbiter grants access to a single master at a time. Once a master has control of the bus, the master performs transfers with any bus slave. When multiple masters attempt to access the bus at the same time, the arbiter allocates the bus resources to a single master, forcing all other masters to wait.

Figure 2–6 illustrates the bus architecture for a traditional processor system. Access to the shared system bus becomes the bottleneck for throughput: only one master has access to the bus at a time, which means that other masters are forced to wait and only one slave can transfer data at a time.

Figure 2–6. Bus Architecture in a Traditional Microprocessor System



Slave-Side Arbitration

The system interconnect fabric uses multimaster architecture to eliminate the bottleneck for access to a shared bus. Multiple masters can be active at the same time, simultaneously transferring data with independent slaves. For example, Figure 2–1 on page 2–2 demonstrates a system with two masters (a CPU and a DMA controller) sharing a slave (an SDRAM controller). Arbitration is performed at the SDRAM slave; the arbiter dictates which master gains access to the slave if both masters initiate a transfer with the slave in the same cycle.

Figure 2–7 focuses on the two masters and the shared slave and shows additional detail of the data, address, and control paths. The arbiter logic multiplexes all address, data, and control signals from a master to a shared slave.

Figure 2–7. Detailed View of Multimaster Connections

Arbiter Details

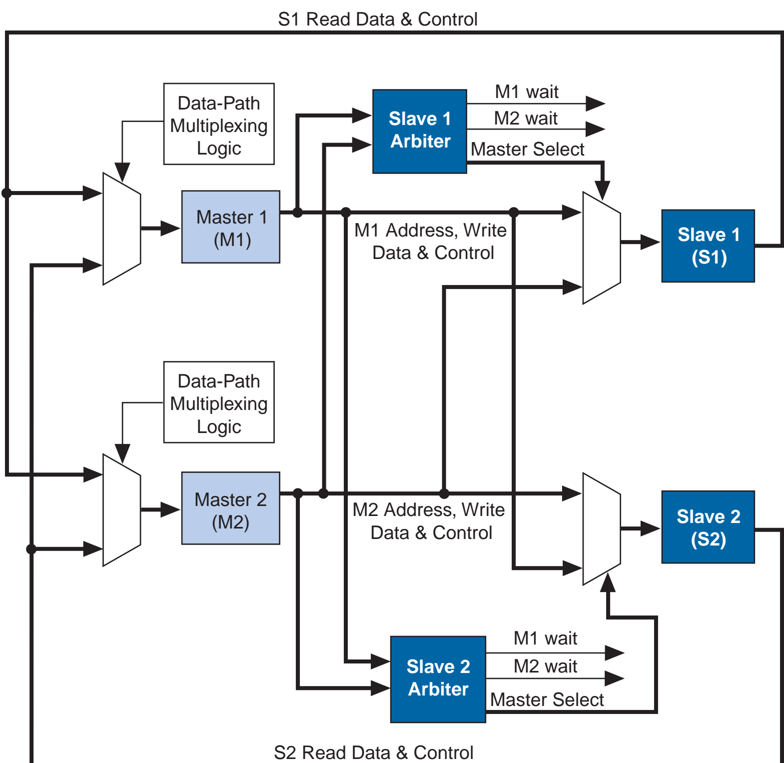
SOPC Builder generates an arbiter for every slave, based on arbitration parameters specified in SOPC Builder. The arbiter logic performs the following functions for its slave:

- Evaluates the address and control signals from each master and determines which master, if any, gains access to the slave next.

- Grants access to the chosen master and forces all other requesting masters to wait.
- Uses multiplexers to connect address, control, and datapaths between the multiple masters and the slave.

Figure 2–8 shows the arbiter logic in an example multimaster system with two masters, each connected to two slaves.

Figure 2–8. Block Diagram of Arbiter Logic



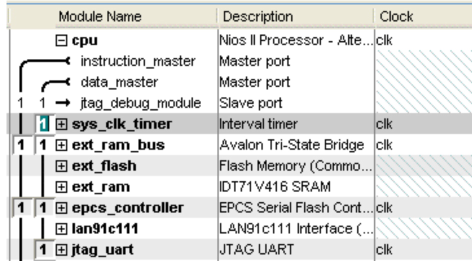
Arbitration Rules

This section describes the rules by which the arbiter grants access to masters when they contend.

Setting Arbitration Parameters in SOPC Builder

You specify the arbitration shares for each master using the connection panel on the **System Contents** tab of SOPC Builder, as shown in Figure 2–9.

Figure 2–9. Arbitration Settings on the System Contents Tab



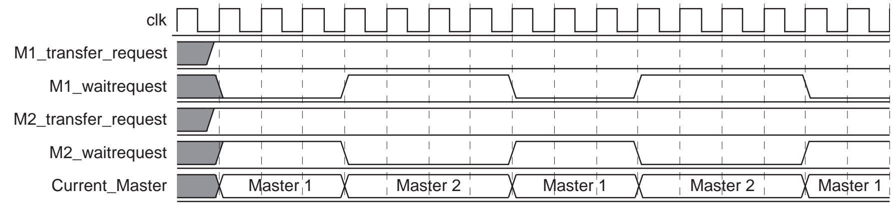
The arbitration settings are hidden by default. To see them, on the View menu, click **Show Arbitration**.

Fairness-Based Shares

Arbiter logic uses a fairness-based arbitration scheme. In a fairness-based arbitration scheme, each master pair has an integer value of transfer *shares* with respect to a slave. One share represents permission to perform one transfer.

For example, assume that two masters continuously attempt to perform back-to-back transfers to a slave. Master 1 is assigned three shares and Master 2 is assigned four shares. In this case, the arbiter grants Master 1 access for three transfers, then Master 2 for four transfers. This cycle repeats indefinitely. Figure 2–10 demonstrates this case, showing each master's transfer request output, wait request input (which is driven by the arbiter logic), and the current master with control of the slave.

Figure 2–10. Arbitration of Continuous Transfer Requests from Two Masters



If a master stops requesting transfers before it exhausts its shares, it forfeits all its remaining shares, and the arbiter grants access to another requesting master. Refer to Figure 2–11. After completing one transfer, Master 2 stops requesting for one clock cycle. As a result, the arbiter grants access back to Master 1, which gets a replenished supply of shares.

Figure 2–11. Arbitration of Two Masters with a Gap in Transfer Requests

Round-Robin Scheduling

When multiple masters contend for access to a slave, the arbiter grants shares in round-robin order. Round-robin scheduling drives a request interface according to space available and data available credit interfaces. At every slave transfer, only requesting masters are included in the arbitration.

Burst Transfers

Avalon-MM burst transfers grant a master uninterrupted access to a slave for a specified number of transfers. The master specifies the number of transfers when it initiates the burst. Once a burst begins between a master-slave pair, arbiter logic does not allow any other master to access the slave until the burst completes. For burst masters, the size of the burst determines the number of cycles that the master has access to the slave, and the selected arbitration shares have no effect.

Burst Adapters

System interconnect fabric provides burst adaptation logic to accommodate the burst capabilities of each port in the system, including ports that do not support burst transfers. Burst adaptation logic consists of a finite state machine that translates the sequencing of address and control signals between the slave side and the master side.

The maximum burst length for each port is determined by the component design and is independent of other ports in the system. Therefore, a particular master might be capable of initiating a burst longer than a slave’s maximum supported burst length. In this case, the burst management logic translates the master burst into smaller slave bursts, or into individual slave transfers if the slave does not support bursts. Until the master completes the burst, the arbiter logic prevents other masters from accessing the target slave.

For example, if a master initiates a burst of 16 transfers to a slave with maximum burst length of 8, the burst adapter logic initiates two bursts of length 8 to the slave. If the master initiates a burst of 14, the burst adapter logic segments the burst transfer into a burst of 8 words followed by a burst of 6 words, because the slave can only handle a maximum burst length of 8. If a master initiates a burst of 16 transfers to a slave that does not support bursts, the burst management logic initiates 16 separate transfers to the slave.

The burst adapter inserts one idle cycle at the start of each burst. System throughput is maximized when burst sizes are as large as possible.

In the case of a non-linewrap burst master connected to a slave with the `linewrapBursts` property set to `TRUE`, it is not always possible to issue the maximum-sized burst to the slave. In cases where a burst would cross a slave burst boundary, the burst adapter must issue the appropriate smaller bursts according to the master request. For example, if a non-linewrap burst master, data width=32 issues a burst of 8 at byte address 0xC to a linewrap burst slave, data width=32, the burst adapter issues a burst read of 5 at byte address 0xC followed by a burst read of size 3 at address 0x20. This example assumes a maximum burst size of 8 for both the master and slave. Table 2–3 provides some examples that show how bursts are handled between master and slaves with and without linewrapping. (Linewrap bursts are common for SDRAM components.) In these examples the following conditions are true:

- The master and slave have the same data width.
- Masters with the `linewrapBursts` property set to `TRUE` must also set `alwaysBurstMaxBurst` to `TRUE` due to a limitation in the burst adapter.

Table 2–4. Burst Behavior for Masters and Slaves with and without Linewrapping

Master Max Burst Size	**8**								**4**			
Slave Max Burst Size	**8**				**4**				**8**			
Master Linewrap Bursts	**True**		**False**		**True**		**False**		**True**		**False**	
Slave Linewrap Bursts	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**	**T**	**F**
Master bursts	8@0	8@0	2@0	2@0	8@0	8@0	2@0	2@0	4@0	4@0	2@0	2@0
Slave receives	8@0	8@0	2@0	2@0	4@0, 4@4	8@0	2@0	2@0	4@0	4@0	2@0	2@0
Master bursts	8@3	8@3	6@3	6@3	8@3	4@0 4@4	6@3	6@3	4@7	4@7	4@7	4@3
Slave Receives	8@3	5@3, 3@0	5@3, 1@8	6@3	1@3, 4@4, 3@0	1@3, 4@4, 3@0	1@3, 4@4, 1@8	4@3, 2@7	1@7, 3@4	1@7, 3@4	1@7, 3@8	4@3

For more information about the `linewrapBursts` property, refer to the *Avalon Memory-Mapped Slave Interfaces* chapter in the *Avalon Interface Specifications.*

Interrupts

In systems where components have interrupt request (IRQ) sender interfaces, the system interconnect fabric includes interrupt controller logic. A separate interrupt controller is generated for each interrupt receiver. The interrupt controller aggregates IRQ signals from all interrupt senders, and maps them to user-specified values on the receiver inputs.

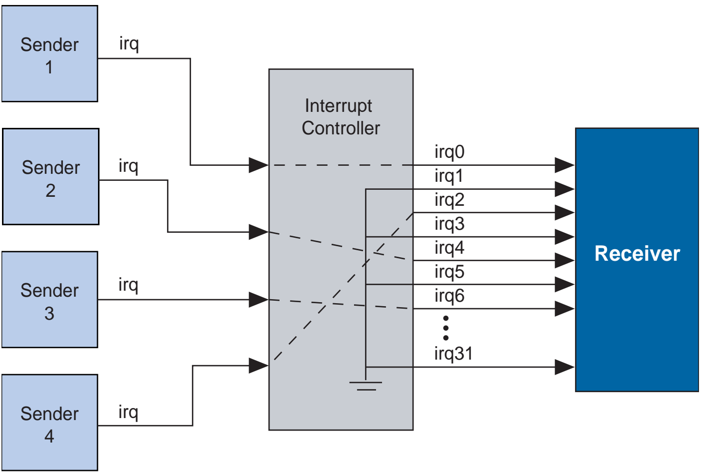
For further information, refer to the *Interrupt Interfaces* chapter in the *Avalon Interface Specifications*.

Individual Requests IRQ Scheme

In the individual requests IRQ scheme, the system interconnect fabric passes IRQs directly from the sender to the receiver, without making any assumptions about IRQ priority. In the event that multiple senders assert their IRQs simultaneously, the receiver logic (presumably under software control) determines which IRQ has highest priority, then responds appropriately.

Using individual requests, the interrupt controller can handle up to 32 IRQ inputs. The interrupt controller generates a 32-bit signal `irq[31:0]` to the receiver, and simply maps slave IRQ signals to the bits of `irq[31:0]`. Any unassigned bits of `irq[31:0]` are disabled. Figure 2–12 shows an example of the interrupt controller mapping the IRQs on four senders to `irq[31:0]` on a receiver.

Figure 2–12. IRQ Mapping Using Software Priority

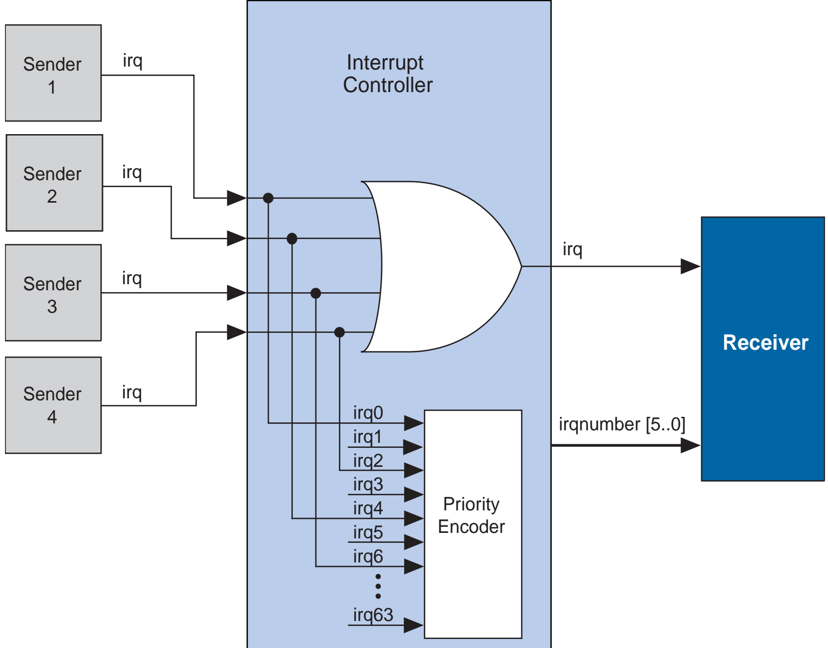


Priority Encoded Interrupt Scheme

In the priority encoded interrupt scheme, in the event that multiple slaves assert their IRQs simultaneously, the system interconnect fabric provides the interrupt receiver with a 1-bit interrupt signal, and the number of the highest priority active interrupt. An IRQ of lesser priority is undetectable until all IRQs of higher priority have been serviced.

Using priority encoded interrupts, the interrupt controller can handle up to 64 slave IRQ signals. The interrupt controller generates a 1-bit `irq` signal to the receiver, signifying that one or more senders have generated an IRQ. The controller also generates a 6-bit `irqnumber` signal, which outputs the encoded value of the highest pending IRQ. See Figure 2–13.

Figure 2–13. IRQ Mapping Using Hardware Priority



Assigning IRQs in SOPC Builder

You specify IRQ settings on the **System Contents** tab of SOPC Builder. After adding all components to the system, you make IRQ settings for all interrupt senders, with respect to each interrupt receiver. For each slave, you can either specify an IRQ number, or specify not to connect the IRQ.

Reset Distribution

SOPC Builder generates the logic used in the system interconnect fabric, which drives the reset pulse to all the logic. The system interconnect fabric distributes the reset signal conditioned for each clock domain. The duration of the reset signal is at least one clock period.

The system interconnect fabric asserts the system-wide reset in the following conditions:

- The global reset input to the SOPC Builder system is asserted.
- Any component asserts its `resetrequest` signal.

The global reset and reset requests are `OR`ed together. This signal is then synchronized to each clock domain associated to an Avalon-MM port, which causes the asynchronous resets to be de-asserted synchronously.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Avalon Memory-Mapped Bridges* chapter in volume 4 of the *Quartus II Handbook*
- *Avalon Memory-Mapped Design Optimizations* chapter in the *Embedded Design Handbook*

Document Revision History

Table 2–5 shows the revision history for this chapter.

Table 2–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009, v9.0.0	■ Added table showing the behavior of the burst adapter for master and slaves with and without `linewrapBursts` set to `TRUE`.	Clarification of burst behavior.
November 2008, v8.1.0	■ Added discussion of a non-bursting Avalon-MM master connected to a Avalon-MM slave with `linewrapBursts = TRUE`. Removed discussion on minimum arbitration shares; this feature is no longer supported. ■ Changed page size to 8.5 x 11 inches	Minor update to reflect software changes.
May 2008, v8.0.0	■ Updated references to Avalon Memory-Mapped and Streaming Interface Specifications and changed to Avalon Interface Specifications. ■ Moved clock-crossing bridge section from this chapter to chapter 11.	The two specifications have been combined into one for all Avalon interfaces.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

3. System Interconnect Fabric for Streaming Interfaces

QII54019-9.0.0

Introduction

Avalon® Streaming interconnect fabric connects high-bandwidth, low latency components that use the Avalon Streaming (Avalon-ST) interface. This interconnect fabric creates datapaths for unidirectional traffic including multichannel streams, packets, and DSP data. This chapter describes the Avalon-ST interconnect fabric and its use in connecting components with Avalon-ST interfaces. Descriptions of specific adapters and their use in streaming systems can be found in the following sections:

- "Adapters" on page 3–3
- "Multiplexer Examples" on page 3–5

High-Level Description

Avalon-ST interconnect fabric is logic generated by SOPC Builder. Using SOPC Builder, you specify how Avalon-ST source and sink ports connect. SOPC Builder then creates a high performance point-to-point interconnect between the two components. The Avalon-ST interconnect is flexible and can be used to implement on-chip interfaces for industry standard telecommunications and data communications cores, such as Ethernet IEEE 802.3 MAC and SPI 4.2. In all cases, bus widths, packets, and error conditions are custom-defined.

Figure 3–1 illustrates the simplest system example that generates an interconnect between the source and sink. This source-sink pair includes only the `data` and `valid` signals.

Figure 3–1. Interconnect for a Simple Avalon Streaming Source-Sink Pair

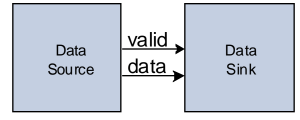
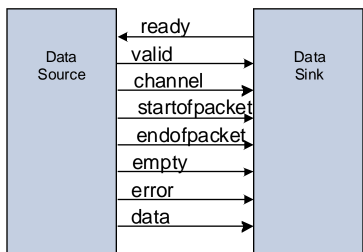


Figure 3–2 illustrates a more extensive interface that includes signals indicating the start and end of packets, channel numbers, error conditions, and back pressure.

Figure 3–2. Avalon Streaming Interface for Packet Data



All data transfers using Avalon-ST interconnect occur synchronously to the rising edge of the associated clock interface. All outputs from the source interface, including the data, channel, and error signals, must be registered on the rising edge of the clock. Registers are not required for inputs at the sink interface. Registering signals only at the source provides for high frequency operation while eliminating back-to-back registration with no intervening logic. There is no inherent maximum performance of the interconnect. Throughput for a system depends on the components and how they are connected.

For details about the Avalon-ST interface protocol, refer to the *Avalon Interface Specifications*.

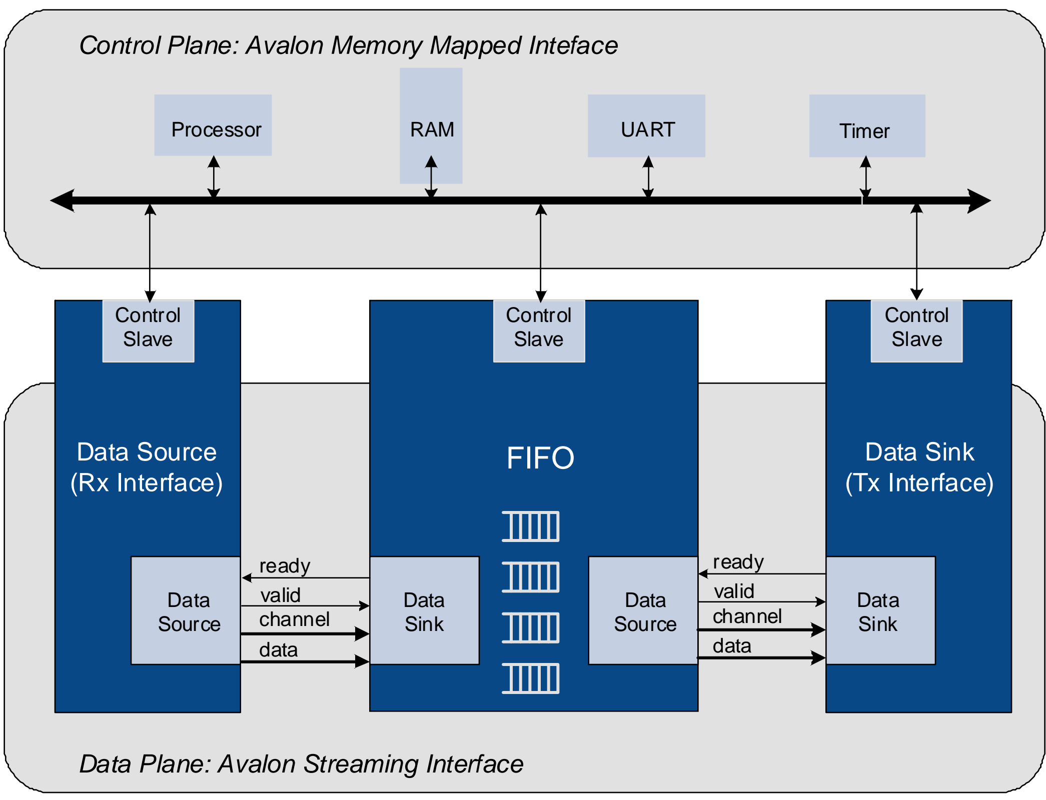
Avalon Streaming and Avalon Memory-Mapped Interfaces

The Avalon-ST and Avalon Memory-Mapped (Avalon-MM) interfaces are complementary. High bandwidth components with streaming data typically use Avalon-ST interfaces for the high throughput datapath. These components can also use Avalon-MM connection interfaces to provide an access point for control. In contrast to the Avalon-MM interconnect, which can be used to create a wide variety of topologies, the Avalon-ST interconnect fabric always creates a point-to-point between a single data source and data sink, as Figure 3–3 illustrates. There are two connection pairs in this figure:

- The data source in the Rx Interface transfers data to the data sink in the FIFO.
- The data source in the FIFO transfers data to the Tx Interface data sink.

In Figure 3–3, the Avalon-MM interface allows a processor to access the data source, FIFO or data sink to provide system control.

Figure 3–3. Use of the Avalon Memory-Mapped and Streaming Interfaces



Adapters

Adapters are configurable SOPC Builder components that are part of the streaming interconnect fabric. They are used to connect source and sink interfaces that are not exactly the same without affecting the semantics of the data. SOPC Builder includes the following four adapters:

- Data Format Adapter
- Timing Adapter
- Channel Adapter
- Error Adapter

You can add Avalon-ST adapters between two components with mismatched interfaces. The adapter allows you to connect a data source to a data sink of differing byte sizes. If you connect mismatched Avalon-ST sources and sinks in SOPC Builder without inserting adapters, SOPC Builder generates error messages. Inserting adapters into the system does not change the types of components that SOPC Builder allows you to connect. The **Insert Avalon-ST Adapters** command on the System menu attempts to correct these errors automatically, if possible, by inserting the appropriate adapter types.

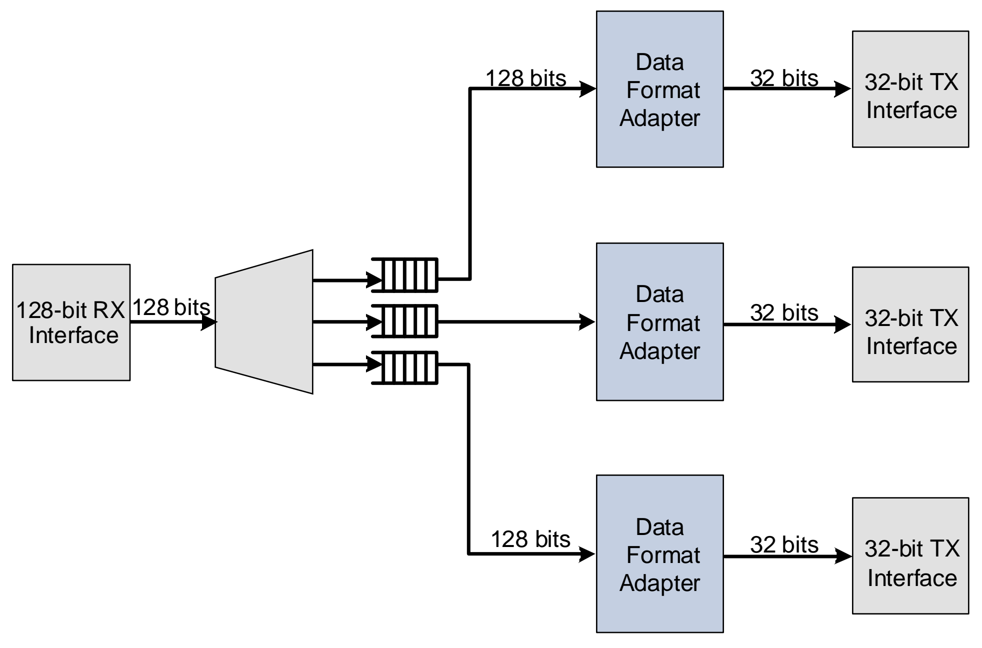
For complete information about these adapters, refer to the *Avalon Streaming Interconnect Components* chapter in volume 4 of the *Quartus II Handbook*.

The following sections provide an overview of these adapters.

Data Format Adapter

The data format adapter allows you to connect interfaces that have different values for the parameters defining the data signal. One of the most common uses of this adapter is to convert data streams of different widths. Figure 3–4 shows an adapter that allows a connection between a 128-bit input data stream and three 32-bit output data streams.

Figure 3–4. Avalon Streaming Interconnect Fabric with Data Format Adapter



Timing Adapter

The timing adapter allows you to connect component interfaces that require a different number of cycles before driving or receiving data. This adapter inserts a FIFO between the source and sink to buffer data or pipeline stages to delay the back pressure signals. The timing adapter can also be used to connect interfaces that support the `ready` signal and those that do not.

Channel Adapter

The channel adapter provides adaptations between interfaces that have different support for the channel signal or channel-related parameters. For example, if the source channel is narrower than the sink channel, you can use this adapter to connect them. The high-order bits of the sink channel are connected to zero. You can also use this adapter to connect a source with a wider channel to a sink with a narrower channel. If the source provides data for a channel that the sink cannot receive, the data is not transferred.

Error Adapter

The error adapter ensures that per-bit error information provided by the source interface is correctly connected to the sink interface's input error signal. Matching error conditions handled by the source and sink are connected. If the source has an error condition that is not supported by the sink, the signal is left unconnected; the adapter provides a simulation error message and an error indication if this error is ever asserted. If the sink has an error condition that is not supported by the source, the sink's input is tied to zero.

Multiplexer Examples

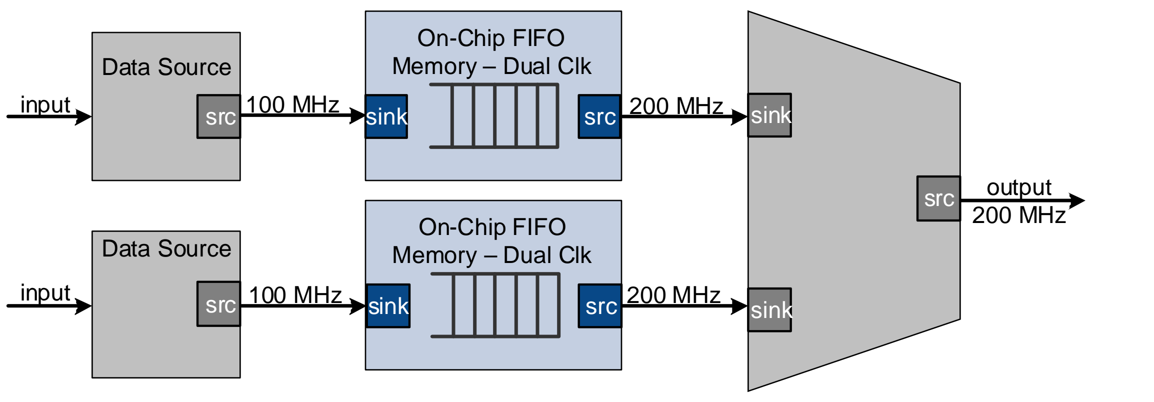
You can combine these adapters with streaming components to create datapaths whose input and output streams have different properties. The following sections provide examples of datapaths constructed using SOPC Builder in which the output stream is higher performance than the input stream:

- The first example shows an output with double the throughput of each interface with a corresponding doubling of the clock frequency.
- The second example doubles the data width.
- The third example boosts the frequency of a stream by 10% by multiplexing input data from two sources.

Example to Double Clock Frequency

Figure 3–5 illustrates a datapath that uses the dual clock version of the on-chip FIFO memory and Avalon-ST channel multiplexer to merge the 100 MHz input from two streaming data sources into a single 200 MHz streaming output. As Figure 3–5 illustrates, this example increases throughput by increasing the frequency and combining inputs.

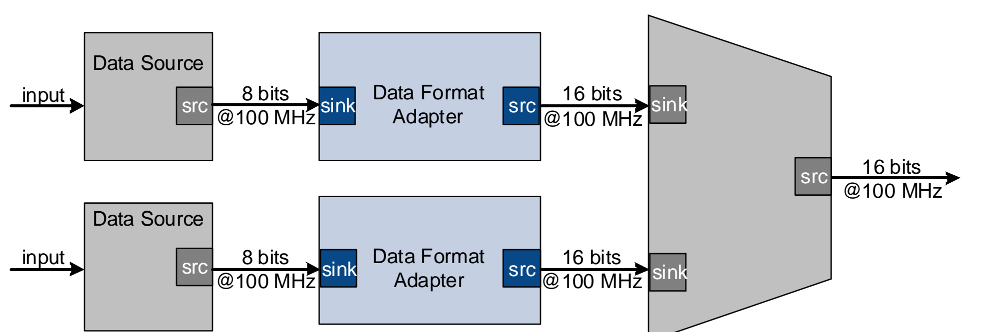
Figure 3–5. Datapath that Doubles the Clock Frequency



Example to Double Data Width and Maintain Frequency

Figure 3–6 illustrates a datapath that uses the data format adapter and Avalon-ST channel multiplexer to convert two, 8-bit inputs running at 100 MHz to a single 16-bit output at 100 MHz.

Figure 3–6. Datapath to Double Data Width and Maintain Original Frequency

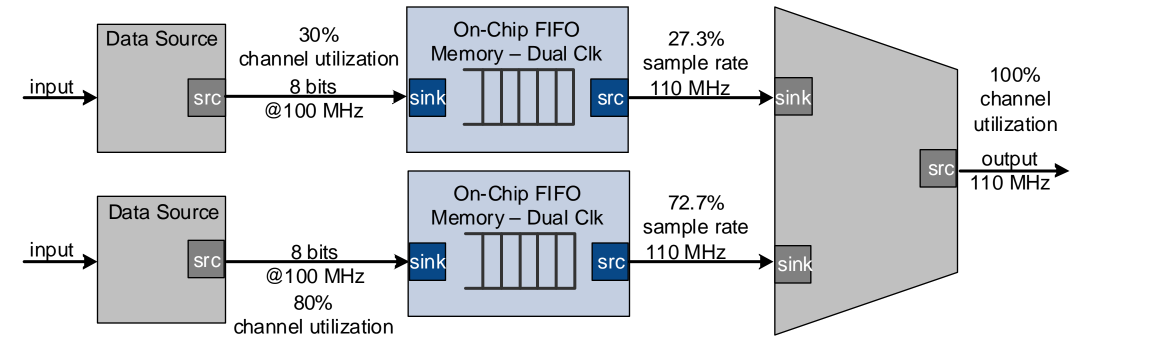


Example to Boost the Frequency

Figure 3–7 illustrates a datapath that uses the dual clock version of the on-chip FIFO memory to boost the frequency of input data from 100 MHz to 110 MHz by sampling two input streams at differential rates. In this example, the on-chip FIFO memory has an input clock frequency of 100 MHz and an output clock frequency of 110 MHz. The channel multiplexer runs at 110 MHz and samples one input stream 27.3 percent of the time and the second 72.7 percent of the time.

You do not need to know what the typical and maximum input channel utilizations are before attempting this. For example, if the first channel hits 50% utilization, the output stream exceeds 100% utilization.

Figure 3–7. Datapath to Boost the Clock Frequency



Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Avalon Streaming Interconnect Components* chapter in volume 4 of the *Quartus II Handbook*

Document Revision History

Table 3–1 shows the revision history for this chapter.

Table 3–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009, v9.0.0	No changes from previous release.	—
November 2008, v8.1.0	■ Added information on error adapter. ■ Changed page size to 8.5 x 11 inches	—
May 2008, v8.0.0	Updated references to Avalon Memory-Mapped and Streaming Interface Specifications and changed to Avalon Interface Specifications.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

4. SOPC Builder Components

QII54004-9.0.0

An SOPC Builder *component* is a hardware design block available within SOPC Builder that can be instantiated in an SOPC Builder system. This chapter defines SOPC Builder components, with emphasis on the structure of custom components.

A component includes the following:

- The HDL description of the component's hardware.
- A description of the interface to the component hardware, such as the names and types of I/O signals.
- A description of any parameters that specify the structure of the component logic and component.
- A GUI for configuring an instance of the component in SOPC Builder.
- Scripts and other information SOPC Builder requires to generate the HDL files for the component and integrate the component instance into the SOPC Builder system.
- Other component-related information, such as reference to software drivers, necessary for development steps downstream of SOPC Builder.

This chapter discusses the design flow for new and classic custom-defined SOPC Builder components, in the following sections:

- "Component Providers" on page 4–1
- "Component Hardware Structure" on page 4–2
- "Exported Connection Points—Conduit Interfaces" on page 4–4
- "SOPC Builder Component Search Path" on page 4–4
- "Component Structure" on page 4–7
- "Classic Components in SOPC Builder" on page 4–8

Component Providers

SOPC Builder components can be obtained from many providers, including the following:

- The components automatically installed with the Quartus® II software.
- Third-party IP developers can provide IP blocks as SOPC Builder-ready components, including software drivers and documentation. A list of third-party components can be found in SOPC Builder by clicking **IP MegaStore** on the Tools menu.
- Altera development kits, such as the Nios® II Development Kit, can provide SOPC Builder components as features.
- You can use the SOPC Builder component editor to convert your own HDL files into custom components.

The GUI interfaces for classic components run slower in newer versions of SOPC Builder when you add or modify your component settings. These components are marked by a gray dot in the **System Contents** tab. You have better performance when you upgrade to the Hardware Component Description File (**_hw.tcl**) component format in newer versions of SOPC Builder. These components are marked by a green dot.

For more information about the **_hw.tcl** file, refer to the *Component Editor* chapter in volume 4 of the *Quartus II Handbook*.

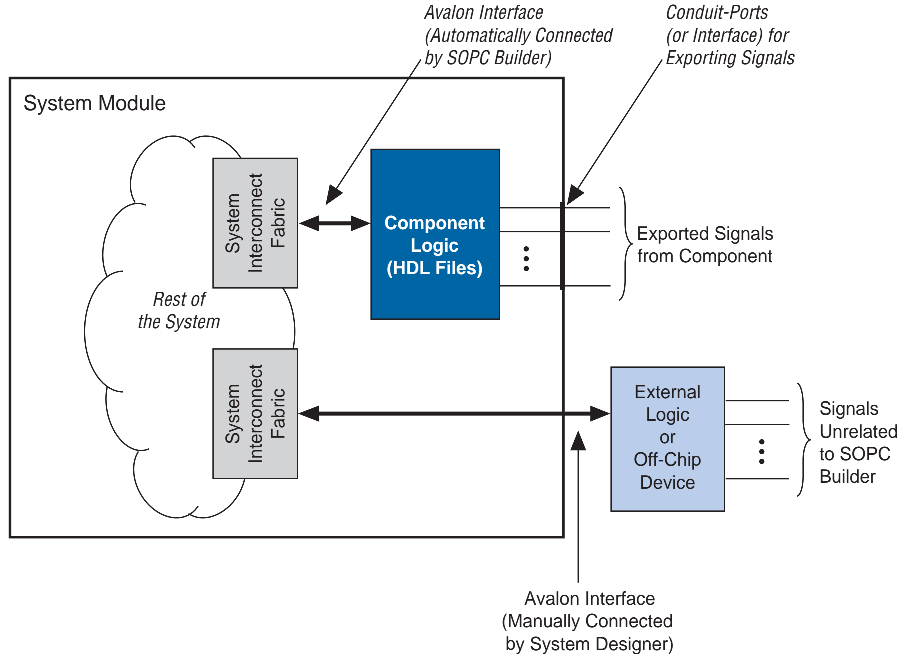
Component Hardware Structure

There are the following types of components in an SOPC Builder system, based on where the associated component logic resides:

- Components that include their associated logic inside the SOPC Builder system
- Components that interface to logic outside the SOPC Builder system

Figure 4–1 shows an example of both types of components.

Figure 4–1. Component Logic Inside and Outside the SOPC Builder System



© March 2009 Altera Corporation

Components Inside the SOPC Builder System

For components that are instantiated inside the SOPC Builder system, the component defines its logic in an associated HDL file. During system generation, SOPC Builder instantiates the component and connects it to the rest of the system. The component can include exported signals in conduit interfaces. Conduit interfaces become ports on the system, so they can be connected to logic outside the SOPC Builder system in the board-level schematic.

For more information about conduit interfaces, refer to the *Conduit Interfaces* chapter in the *Avalon Interface Specifications*.

In general, components connect to the system interconnect fabric using the Avalon® Memory-Mapped (Avalon-MM) interface or the Avalon Streaming (Avalon-ST) interface. A single component can provide more than one Avalon port. For example, a component might provide an Avalon-ST source port for high-throughput data, in addition to an Avalon-MM slave for control.

Static HDL Components

You can define SOPC Builder components whose parameters are all assigned values during the initial editing session. Examples of parameters whose values are typically known at instantiation time are address and data widths and FIFO depths. If all of a component's parameters are assigned when it is instantiated, the HDL for the component is *static*. SOPC Builder automatically generates the top-level HDL wrapper file to apply parameter values to your component.

Dynamic HDL Components

You can also create SOPC Builder components whose parameters are defined by a generation callback. Examples of parameters that might be assigned during generation callback are baud rate and output directory. When you create components that include parameters defined using a generation callback, you must provide a custom generation callback routine to create the top-level wrapper for your component.

For more information about defining your own generation program, refer to the *Generation Callback* section in the *Component Interface Tcl Reference* chapter in volume 4 of the *Quartus II Handbook.*

Components Outside the SOPC Builder System

For components that interface to external logic or off-chip devices with Avalon-compatible signals outside the SOPC Builder system, the component files describe only the interface to the external logic. During system generation, SOPC Builder exports an interface for the component in the top-level SOPC Builder system. You must manually connect the signals at the top-level of SOPC Builder to pins or logic defined outside the system that already has Avalon-compatible signals.

Exported Connection Points—Conduit Interfaces

Conduit interfaces are brought to the top level of the system as additional ports. Exported signals are usually either application-specific signals or the Avalon interface signals.

Application-specific signals are exported to the top level of the system by the conduit interface(s) defined in the **_hw.tcl** file. These are I/O signals in a component's HDL logic that are not part of any Avalon interfaces and connect to an external device, for example DDR SDRAM memory, or logic defined outside of the SOPC Builder system. You use conduit interfaces to connect application-specific signals of the external device and the SOPC Builder system.

You can also export the Avalon interfaces to manually connect them to external devices or logic defined outside a system with Avalon-compatible signals. This method allows a direct connection to the Avalon interface from any device that has Avalon-compatible signals. You can also export the Avalon interface in either an HDL file using conduit interfaces, or in the **_hw.tcl** file without an HDL file.

You export the Avalon interface signals as an HDL file with simple wire connections in the HDL description. The Avalon interface port signals are directly connected to external I/O signals in the HDL description. The conduit interface in the **_hw.tcl** file exports the external I/O signals to the top level of the system.

In the **_hw.tcl** file, no HDL files are specified and only the Avalon signals and interface ports are declared in the file.

SOPC Builder Component Search Path

Each time SOPC Builder starts, it searches for component files. The components that SOPC Builder finds are displayed in the list of available components on the SOPC Builder **System Contents** tab. When you launch SOPC Builder certain directories are searched for two kinds of files:

- **_hw.tcl** files. Each **_hw.tcl** file defines a single component.
- IP Index (**.ipx**) files. Each file indexes a collection of available components.

In general, **.ipx** files facilitate faster startup for SOPC Builder and other tools because fewer files need to be read and analyzed.

Some directories are searched recursively; others only to a specific depth. In the following list of search locations, a recursive descent is annotated by **. The * signifies any file. When a directory is recursively searched, the search stops at any directory containing a **_hw.tcl** or **.ipx** file; subdirectories are not searched.

- `$$PROJECT_DIR/*`
- `$$PROJECT_DIR/ip/**/*`
- `$QUARTUS_ROOTDIR/../ip/**/*`

In SOPC Builder, you can extend the default search path by including additional directories by clicking **Options**, then clicking **IP Search Path** and **Add**. These additional paths apply to all projects; that is, the paths are global to the current version of SOPC Builder.

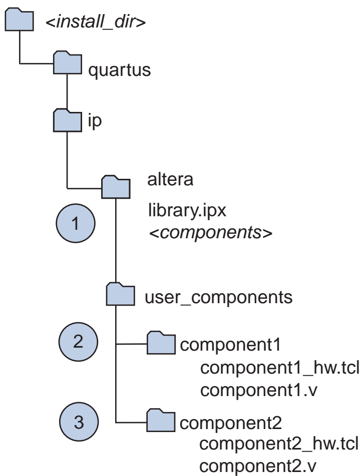
Installing Additional Components

There are two additional ways to make your components available to SOPC Builder projects. The following sections describe these methods.

Copy to the IP Root Directory

The simplest strategy is to copy your components into the standard IP directory provided by Altera. Figure 4–2 illustrates this approach.

Figure 4–2. User Library Included In Subdirectory of $IP_ROOTDIR



In Figure 4–2, the circled numbers identify three steps of the algorithm that SOPC follows during initialization. These steps are explained in the following paragraphs.

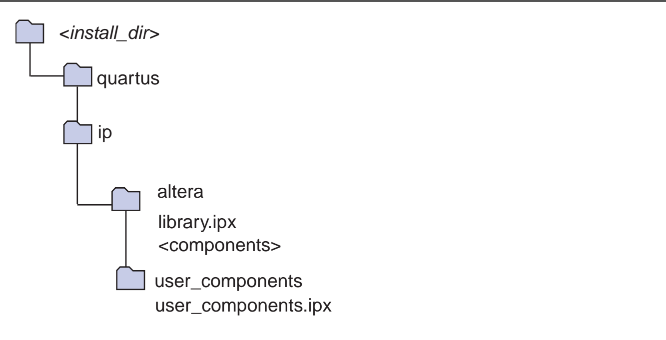
1. SOPC Builder recursively searches the *<install_dir>***/ip/** directory by default. It finds the file in the `altera` subdirectory, which tells it about all of the Altera components. **library.ipx** includes listings for all components found in its subdirectories. The recursive search stops when SOPC Builder finds this **.ipx** file.
2. As part of its recursive search, SOPC Builder also looks in the adjacent **user_components** directory. One level down SOPC Builder finds the **component1** directory, which contains **component1_hw.tcl**. SOPC Builder finds that component stops the recursive descent.
3. SOPC Builder then searches in the adjacent **component2** directory, which includes **component2_hw.tcl**. If SOPC Builder finds that component, the recursive descent stops.

If you save your **.ipx** file in the *<install_dir>***/ip/** directory, SOPC Builder finds your **.ipx** file and stops. SOPC Builder does not conduct the search just described.

Reference Components in an .ipx File

A second approach is to specify your IP directory in a **user_components.ipx** file under *<install_dir>***/ip** path. Figure 4–3 illustrates this approach.

Figure 4–3. Specifying A User .ipx directory



The **user_components.ipx** file includes a single line of code redirecting SOPC Builder to the location of the user library. Example 4–1 shows the code for this redirection.

Example 4–1. Redirect to User Library

```
<library>
  <path path="c:/<user_install_dir>/user_ip/**/*" />
<library>
```

For both of these approaches, if you install a new version of the Quartus II software, you must also update the installation to include your libraries.

Understanding IPX File Syntax

An **.ipx** file is an XML file whose top-level element is *<library>* with a *<path>* subelements are *<path>* and *<component>*. Altera recommends that you only add or edit the *<path>* subelement.

A *<path>* element contains a single attribute, also called `path` and may reference a directory with a wildcard, (*), or reference a single file. Two asterisks designate any number of subdirectories. A single asterisk designates a match to a single file or directory. In searching down the designated path, the following three types of files are identified:

- **.ipx**—additional index files
- **_hw.tcl**—SOPC Builder component definitions
- **_sw.tcl**—Nios II board support package (BSP) software component definitions

A *<component>* element contains several attributes to define a component. If you provide all the required details for each component in an **.ipx** file, the start-up time for SOPC Builder is less than if SOPC Builder must discover the files in a directory. Example 4–2 shows two *<component>* elements. Note that the paths for file names are specified relative to the **.ipx** file.

Example 4–2. Component Elements

```
<library>
  <component
    name="An SOPC Component"
    displayName="SOPC Component"
    version="2.1"
    file="./components/sopc_component/sc_hw.tcl"
   />
  <component
    name="legacy_component"
    displayName="Legacy Component (Classic Edition!)"
    version="0.9"
    file="./components/legacy/old_component/class.ptf"
   />
<library>
```

Upgrading from Earlier Versions

If you specified a custom search path in SOPC Builder prior to v8.1 using the **IP Search Path** option, or by adding it to the `$SOPC_BUILDER_PATH`, SOPC Builder automatically adds those directories to the **user_components.ipx** file in your home directory. This file is saved in *<home_dir>***/altera.quartus/ip/8.1/ip_search_path/user_components.ipx.** Go to the **IP Search Path** option in the **Options** dialog box to see the directories listed here**.**

Component Structure

Most components are defined with a **_hw.tcl** file, a text file written in the Tcl scripting language that describes the components in to SOPC Builder. You can add a component to SOPC Builder by either writing a Tcl description or you can use the component editor to generate an automatic Tcl description of it. This section describes the structure of Tcl components and how they are stored.

For details about the SOPC Builder component editor, refer to the *Component Editor* chapter in volume 4 of the *Quartus II Handbook*. For details about the SOPC Builder Tcl commands, refer to the *Component Interface Tcl Reference* chapter in volume 4 of the *Quartus II Handbook*.

Component Description File (_hw.tcl)

A Tcl component consists of:

- A component description file, which is a Tcl file with file name of the form *<entity name>***_hw.tcl.**
- Verilog HDL or VHDL files that define the top-level module of the custom component (optional).

The **_hw.tcl** file defines everything that SOPC Builder requires about the name and location of component design files.

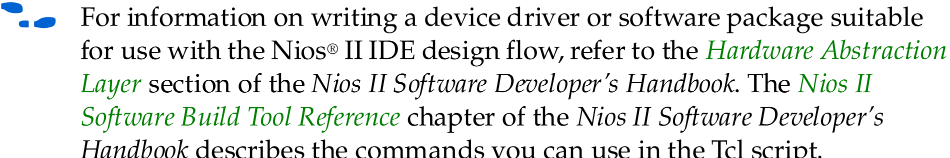
The SOPC Builder component editor saves components in the **_hw.tcl** format. You can use these Tcl files as a template for editing components by hand. When you edit a previously saved **_hw.tcl** file, SOPC Builder automatically saves the earlier version as **_hw.tcl~**.

For more information about the information that you can include in the **_hw.tcl** file, refer to the *Component Interface Tcl Reference* chapter in volume 4 of the *Quartus II Handbook*.

Component File Organization

A typical component uses the following directory structure. The precise names of the directories are not significant.

- *<component_directory>***/**
  - *<hdl>***/**— a directory that contains the component HDL design files and the **_hw.tcl** file
  - *<component name>***_hw.tcl**—the component description file
  - *<component name>***.v or .vhd**—the HDL file that contains the top-level module
  - *<component_name>***_sw.tcl**—the software driver configuration file. This file specifies the paths for the **.c** and **.h** files associated with the component.
- You are not required to create a special sub-directory for component HDL files. However, you are required to follow the naming conventions given here.
  - *<component_dir>* /
  - *<name>***_hw.tcl**
  - *<name>***.v** or **.vhd**
  - *<name>***_sw.tcl**
- *<software>***/**—a directory that contains software drivers or libraries related to the component, if any. Altera recommends that the software directory be subdirectory of the directory that contains the **_hw.tcl** file.



For information on writing a device driver or software package suitable for use with the Nios® II IDE design flow, refer to the *Hardware Abstraction Layer* section of the *Nios II Software Developer's Handbook*. The *Nios II Software Build Tool Reference* chapter of the *Nios II Software Developer's Handbook* describes the commands you can use in the Tcl script.

Classic Components in SOPC Builder

If you use classic components created with an earlier version of SOPC Builder, read through this section to familiarize yourself with the differences. This document uses the term *classic components* to refer to class.ptf-based components created with a previous version of the Quartus II software. If you do not use classic components, skip this section.

Classic components are compatible with newer versions of SOPC Builder, but be aware of the following caveats:

- Classic components configured with the **More Options** tab in SOPC Builder, such as complex IP components provided by third-party IP developers, are not supported in the Quartus II software in version 7.1 and beyond. If your component has a bind program, you cannot use the component without recreating it with the component editor or with Tcl scripting.
- To make changes to a classic component with the component editor, you must first upgrade the component by editing the classic component and saving it in the **_hw.tcl** component format in the component editor.

Referenced Documents

This chapter references the following documents:

- *Component Interface Tcl Reference* chapter in volume 4 of the *Quartus II Handbook*
- *Component Editor* chapter in volume 4 of the *Quartus II Handbook*
- *Conduit Interfaces* chapter in the *Avalon Interface Specifications*
- *Embedded Peripherals* section in volume 5 of the *Quartus II Handbook*
- *Hardware Abstraction Layer* section of the *Nios II Software Developer's Handbook*
- *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*

Document Revision History

Table 4–1 shows the revision history for this chapter.

Table 4–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009, v9.0.0	■ Added 2 paragraphs introducing custom generations for dynamic components.	Updated component descriptions.
November 2008, v8.1.0	■ Revised section on component search paths. ■ Added meaning of green and gray dots next to components on the **System Contents** tab. ■ Changed page size to 8.5 x 11 inches	Revised to reflect changes to the component search path in 8.1.
May 2008, v8.0.0	■ Added paragraph about IP Search Path.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

5. Using SOPC Builder with the Quartus II Software

QII54023-9.0.0

Introduction

This chapter describes the Quartus® II software tools that interface with SOPC Builder, including the following:

- "Quartus II IP File"
- "Quartus II Incremental Compilation" on page 5–1
- "TimeQuest Timing Analyzer" on page 5–2

Quartus II IP File

The Quartus II IP File (**.qip**) generated by SOPC Builder provides the Quartus II software with all required information about your SOPC Builder system. SOPC Builder creates the **.qip** during system generation and adds a reference to it in the Quartus II Settings File (**.qsf)**.

The **.qip** file includes references to the following information:

- HDL files used in the SOPC Builder system
- TimeQuest Timing Analyzer Synopsys Design Constraint (**.sdc**) files
- Component definition files for archiving purposes

The **.qip** file is based on Tcl scripting syntax and is similar to the **.qsf** file. The information required to process most components is included in the system's single **.qip** file. Some complex components provide their own **.qip** file, in which case the system's **.qip** file references the component **.qip** file.

The **.qip** file is normally added to your project automatically by SOPC Builder. If it does not get added automatically you can add the file in the same way that you add other source files to your project. You can also have a **.qip** file for each component in your design. When you generate a design, each **.qip** is pulled into the main **.qip** file for your system by reference.

Quartus II Incremental Compilation

SOPC Builder supports the Quartus II incremental compilation feature, which allows you to separately compile isolated portions, or partitions, of a design. From within the Quartus II software, you can designate an entire SOPC Builder system as a design partition, or you can designate individual SOPC Builder components as design partitions.

Changing the parameters of a component and regenerating your system only prompts other partitions within the same system to recompile if the HDL in that partition depends on the changed parameters. The HDL you generate for the Nios® II processor is optimized as related to components to which the Nios II processor is connected.

For more information about incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

TimeQuest Timing Analyzer

Altera recommends the TimeQuest Timing Analyzer in the Quartus II software for analysis of all new designs. SOPC Builder automatically generates a TimeQuest **.sdc** constraints file for SOPC Builder systems and components. In most cases, you use the TimeQuest constraints to declare false paths for signals that cross clock domains within a component, so that the TimeQuest Timing Analyzer does not perform normal setup and hold analysis for them. You can add **.sdc** files for custom components, using **Add Files** command on **HDL Files** tab in the Component Editor. Turn on the **Synth** option and turn off the **Synth** option.

The Classic Timing Analyzer was primary in earlier versions of the Quartus II software. However, Altera now recommends that you constrain designs before compilation, because the TimeQuest Timing Analyzer reports any unconstrained paths by default during the compilation process.

Refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* for further description of the TimeQuest Timing Analyzer. Refer to the *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* for a description of the benefits of using the TimeQuest Timing Analyzer rather than the Classic Timing Analyzer. Refer to *TimeQuest Example: Basic SDC Example* on www.altera.com for a working example of using the TimeQuest Timing Analyzer. Refer to *TimeQuest Design Examples* on www.altera.com for further details about how to constrain different types of circuits for the TimeQuest Timing Analyzer.

Analyzing PLLs

You must constrain PLL clocks for proper analysis by the TimeQuest Timing Analyzer. You can define clocks generated by PLLs using one of the following methods:

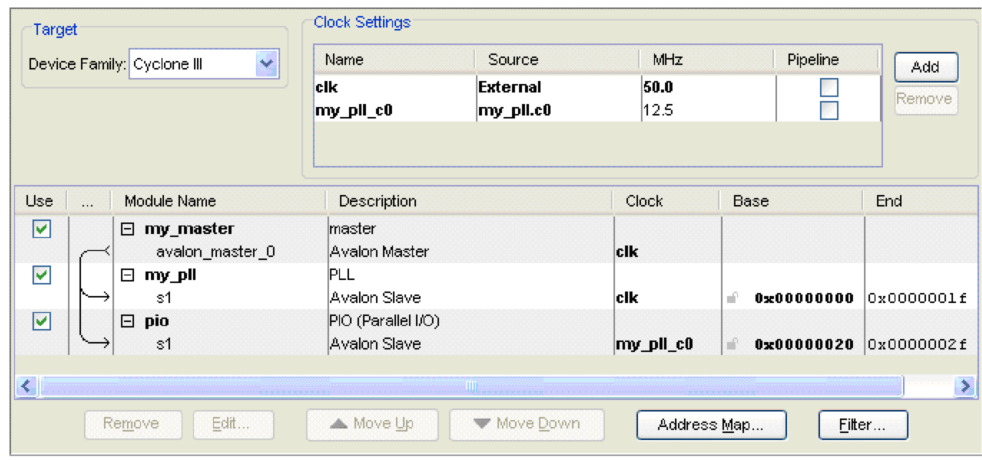
- Use the `derive_pll_clocks` command to derive clocks for all PLL outputs in the design. This is the best method.
- Use the `create_generated_clock` command to designate each clock output.
- Use the `-create_base_clocks` option of the `derive_pll_clock` assignments to designate the base clock feeding the PLL.

The following example focuses on the use of the `derive_pll_clocks` assignment, because this method automatically defines clock frequencies and phase shifts.

`derive_pll_clocks` generates clocks for all PLLs in the Quartus II hardware project, not just for the PLLs in the SOPC Builder system.

The SOPC system shown in Figure 5–1 illustrates the use of the `derive_pll_clocks` assignment in the case of a single clock input and one PLL using a single output.

Figure 5–1. Example SOPC System

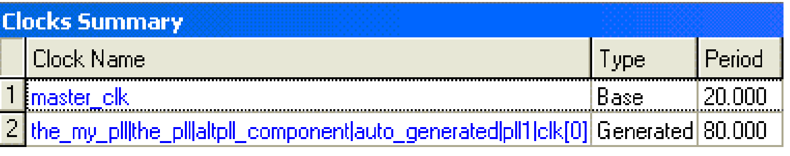


After running the following commands in the TimeQuest Timing Analyzer, two clocks are generated:

```
create_clock -name master_clk -period 20 [get_ports {clk}]
derive_pll_clocks
```

The TimeQuest Timing Analyzer analyzes and reports performance of the constrained clocks in the Clocks Summary report. This displays a report as shown in Figure 5–2.

Figure 5–2. Clocks Summary Report



Clocks Summary

	Clock Name	Type	Period
1	master_clk	Base	20.000
2	the_my_pll|the_pll|altpll_component|auto_generated|pll1|clk[0]	Generated	80.000

`master_clk` is defined by the `create_clock` command, and `the_my_pll` clock is derived from the `derive_pll_clocks` command.

Analyzing Slow Asynchronous I/O Paths

If you use slow asynchronous I/O in an SOPC Builder system, such as PIO and UART peripherals, you do not have to analyze these paths because they are asynchronous to the clock that is used to capture or output data. In this case you must designate false paths to produce an accurate analysis.

For outputs, set a false path between the launch clock and the output. For inputs, a false path should be set between the input and the latching clock. For bidirectional signals, set a false path from the launching clock to the bidirectional pin and also from the bidirectional pin to the latching clock. Launch and latch clocks are typically the clocks associated with the SOPC Builder module that includes the I/O.

For the system described in the PLL section, the following command sets false paths for the PLL outputs:

```
set_false_path -to [get_ports {*_pio[*]}]
```

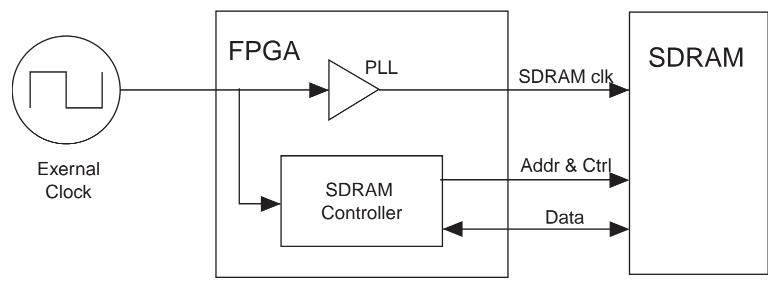
Because design contains a 4-bit PIO, `filter *_pio[*]` includes the following I/O pins.

- `out_port_from_the_pio[0]`
- `out_port_from_the_pio[1]`
- `out_port_from_the_pio[2]`
- `out_port_from_the_pio[3]`

Analyzing Single Data Rate SDRAM and SSRAM

Single data SDRAM interfaces in SOPC Builder typically use the type of circuit shown in Figure 5–3. You can use a PLL to fine tune the phase shift to the external memory to meet I/O timing requirements.

Figure 5–3. Typical Single Data Rate SDRAM Circuit



To constrain this interface, you must create a clock that is recognized by the external SDRAM; then you must set the I/O timing relative to that clock.

Example 5–1 shows how to constrain a PLL output clock and set a Tcl variable for that clock.

Example 5–1. Constraining PLL Output Clock

```
create_clock -period 20.000 -name ext_clk [get_ports {clk}]
derive_pll_clocks
set sdram_clk\my_pll_inst|altpll_component|auto_generated|pll1|clk[0]
```

You can then use the `create_generated_clock` command to define a clock as recognized by the external memory. This generated clock automatically adds delays associated with routing to the clock output pin and the delay of the pin itself. You must also account for some board delay due to the PCB trace between the FPGA and SDRAM by using the `offset` option.

The following command shows the creation of the `sdram_clk_pin` generated clock derived from the output pin `sdram_clk` clock. A 0.5 ns offset accounts for PCB routing delay.

```
create_generated_clock -name sdram_clk_pin -source $sdram_clk \
-offset 0.5 [get_ports {sdram_clk}]
```

There may be some uncertainty associated with the PCB delay not accounted for in this command. The uncertainty can be included in the I/O constraints that are specific to input or output and minimum or maximum delays.

The I/O constraints must be defined in relation to the data sheet for the external memory. Figure 5–4 shows part of a data sheet for an SDRAM device with the worst case input and output timing highlighted for a CAS latency of 3.

Figure 5–4. AC Characteristics from SDRAM Device Data sheet

AC Characteristics Parameter		Symbol	-6		-7		Units	Notes
			Min	Max	Min	Max		
Access time from CLK (pos. edge)	CL = 3	^{t}AC (3)		5.5		5.5	ns	
	CL = 2	^{t}AC (2)		7.5		8	ns	
	CL = 1	^{t}AC (1)		17		17	ns	
Address hold time		^{t}AH	1		1		ns	
Address setup time		^{t}AS	1.5		2		ns	
CLK high-level width		^{t}CH	2.5		2.75		ns	
CLK low-level width		^{t}CL	2.5		2.75		ns	
Clock cycle time	CL = 3	^{t}CK (3)	6		7		ns	23
	CL = 2	^{t}CK (2)	10		10		ns	23
	CL = 1	^{t}CK (1)	20		20		ns	23
CKE hold time		^{t}CKH	1		1		ns	
CKE setup time		^{t}CKS	1.5		2		ns	
CS#, RAS#, CAS#, WE#, DQM hold time		^{t}CMH	1		1		ns	
CS#, RAS#, CAS#, WE#, DQM setup time		^{t}CMS	1.5		2		ns	
Data-in hold time		^{t}DH	1		1		ns	
Data-in setup time		^{t}DS	1.5		2		ns	
Data-out High-Z time	CL = 3	^{t}HZ (3)		5.5		5.5	ns	10
	CL = 2	^{t}HZ (2)		7.5		8	ns	10
	CL = 1	^{t}HZ (1)		17		17	ns	10
Data-out Low-Z time		^{t}LZ	1		1		ns	
Data-out hold time		^{t}OH	2		2.5		ns	

The mapping of external memory timing to FPGA I/O delays is shown in Table 5–1. This also shows whether the minimum or maximum PCB routing delay should be used, which must be added to the FPGA delay constraints.

Table 5–1. External Memory Timing

Memory Timing	FPGA Timing	PCB Routing
Max clock to out	Max input delay	Max
Min clock to out	Min input delay	Min
Min setup	Max output delay	Max
Min hold	Min output delay (-ve)	Min

Note to Table 5–1:

(1) The constraint for minimum output delay is actually 0 – Min hold.

You can use the set_input_delay and set_output_delay commands to set the I/O constraints. In the following examples, a common PCB routing delay of 0.5ns ± 0.1 ns is used, which adds a 0.4 ns or 0.6 ns delay to the paths. Example 5–2 illustrates the use of these commands.

Example 5–2. set_input_delay and set_output_delay commands

```
set_input_delay -clock sdram_clk_pin -max [expr 5.5 + 0.6] <ports>
set_input_delay -clock sdram_clk_pin -min [expr 2.5 + 0.4] <ports>
set_output_delay -clock sdram_clk_pin -max [expr 2.0 + 0.6] <ports>
set_output_delay -clock sdram_clk_pin -min [expr 1 – (1.0 + 0.4)]<ports>
```

In this example, <*ports*> represent a list of I/O ports for the relevant constraints as shown in Example 5–3.

Example 5–3. <ports>

```
set_output_delay -clock sdram_clk_pin -max [expr 2.0 + 1.2] \
[get_ports {cas_n ras_n cs_n we_n addr[*]}]
```

You can use multiple set_input_delay and set_output_delay commands to set different delays for different I/O.

Analyzing Tristate Bridges and Asynchronous Devices

This section discusses the timing constraints associated with the Avalon tristate bridge and asynchronous external devices, such as the CFI Flash and user tristate components. These components typically have slower performance requirements compared with the FPGA, and SOPC Builder generates logic within the interface to control timing across multiple clock cycles. You define the tristate component's timing parameters by entering data for setup, wait, and hold times.

For the interface types previously discussed, the timing is controlled by a state machine that is generated based on setup, wait, and hold settings you specify in the component editor. Because data sheet values for the FPGA are used in calculating the timing, the constraints simply ensure the data sheet timing is met. Adding these constraints ensures that issues associated with data sheet misinterpretation and fitting problems that affect I/O timing are captured.

The TimeQuest Timing Analyzer uses constraints that are based upon the timing of the external device.

For further information on how to convert older FPGA-centric constraints into system-centric constraints, refer to *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Analyzing DDR and DDR2 Memories

When using DDR, DDR2, or DDR3 memory with Cyclone® III, Stratix® III, and Stratix IV families, you must use the corresponding High-Performance Controller MegaCore® function. You can use the MegaWizard™ Plug-In Manager interface to parameterize these functions and generate timing constraints in the form of **.sdc** files. You must ensure that the constraints file associated with the MegaCore function is included in the project for timing analysis. You can add an **.sdc** file to the project by clicking **Add/Remove Files in Project** on the Project menu in the Quartus II software.

As these MegaCore functions make use of the `derive_pll_clocks` command, conflicts may occur if your **.sdc** file also uses these constraints.

For more design examples, refer to TimeQuest Design Examples on www.altera.com. Also, *AN: 433 Constraining and Analyzing Source-Synchronous Interfaces* describes source synchronous constraints for the TimeQuest Timing Analyzer.

Referenced Documents

This chapter references the following documents:

- *AN 433: Constraining and Analyzing Source-Synchronous Interfaces*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Switching to the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- TimeQuest Design Examples
- TimeQuest Example: Basic SDC Example

Document Revision History

Table 5–2 shows the revision history for this chapter.

Table 5–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009, v9.0.0	■ No changes to content from previous release.	—
November 2008, v8.1.0	■ No changes to content from previous release. ■ Changed page size to 8.5 x 11 inches	—
May 2008, v8.0.0	Initial release.	Information moved from other chapters and consolidated here.

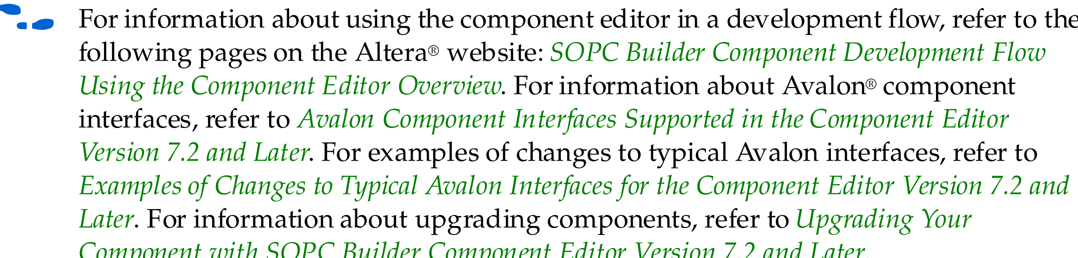
6. Component Editor

QII54005-9.0.0

Introduction

This chapter describes the SOPC Builder component editor. The component editor provides a GUI to support the creation and editing of the Hardware Component Description File (**_hw.tcl**) file that describes a component to SOPC Builder. You use the component editor to do the following:

- Specify the Verilog HDL or VHDL files that describe the modules in your component hardware.
- Conversely, create an HDL template for a component by first defining its interface using the **HDL Files** tab of the component editor.
- Specify the signals for each of the component's interfaces, and define the behavior of each interface signal.
- Specify relationships between interfaces, such as determining which clock interface is used by a slave interface.
- Declare any parameters that alter the component structure or functionality, and define a user interface to let users parameterize instances of the component.



For information about using the component editor in a development flow, refer to the following pages on the Altera® website: *SOPC Builder Component Development Flow Using the Component Editor Overview*. For information about Avalon® component interfaces, refer to *Avalon Component Interfaces Supported in the Component Editor Version 7.2 and Later*. For examples of changes to typical Avalon interfaces, refer to *Examples of Changes to Typical Avalon Interfaces for the Component Editor Version 7.2 and Later*. For information about upgrading components, refer to *Upgrading Your Component with SOPC Builder Component Editor Version 7.2 and Later*.

For information about the use of the component editor, see the following sections:

- "Starting the Component Editor" on page 6–2.
- "HDL Files Tab" on page 6–2.
- "Signals Tab" on page 6–3.
- "Interfaces Tab" on page 6–6.
- "Component Wizard Tab" on page 6–6.
- "Saving a Component" on page 6–8.
- "Editing a Component" on page 6–8.
- "Component GUI" on page 6–8.

For more information about components, refer to the *Component Interface Tcl Reference* chapter in volume 4 of the *Quartus II Handbook*, For more information about the Avalon-MM interface, refer to the *Avalon Interface Specifications*.

Component Hardware Structure

The component editor creates components with the following characteristics:

- A component has one or more interfaces. Typically, an *interface* means an Avalon® Memory-Mapped (Avalon-MM) master or slave or an Avalon Streaming (Avalon-ST) source or sink. You can also specify exported component signals that appear at the top-level of the SOPC Builder system, which can be connected to logic outside the SOPC Builder system. The component editor lets you build a component with any combination of Avalon interfaces, which include:
  - Avalon-MM master and slave
  - Avalon-ST source and sink
  - Avalon-MM tristate slave
  - Interrupt sender and receiver
  - Clock input and output
  - Nios II custom instruction master and slave interfaces
  - Conduit (for exporting signals to the top level)
- Each interface is comprised of one or more signals.
- The component can represent logic that is instantiated inside the SOPC Builder system, or can represent logic outside the system with an interface to it on the generated system.

Starting the Component Editor

To start the component editor in SOPC Builder, on the File menu, click **New Component.** When the component editor starts, the **Introduction** tab displays, which describes how to use the component editor.

The component editor presents several tabs that group related settings. A message window at the bottom of the component editor displays warning and error messages.

Each tab in the component editor provides on-screen information that describes how to use the tab. Click the triangle labeled **About** at the top-left of each tab to view these instructions. You can also refer to Quartus® II online Help for additional information about the component editor.

You navigate through the tabs from left to right as you progress through the component creation process.

HDL Files Tab

The **HDL Files** tab allows you to create an SOPC Builder component from existing Verilog HDL or VHDL files, or to create an HDL template in either Verilog HDL or VHDL for a SOPC Builder component by first specifying its interfaces. The following sections describe both the bottom-up and top-down approaches to component design.

Bottom-Up Design

You can use the **HDL Files** tab to specify Verilog HDL or VHDL files that describe the component logic. Files are provided to downstream tools such as the Quartus II software and ModelSim® in the same order as they appear in the table.

You can also use the component editor to define the interface to components outside the SOPC Builder system. In this case, you do not provide HDL files. Instead, you use the component editor to interactively define the hardware interface.

After you specify an HDL file, the component editor analyzes the file by invoking the Quartus II Analysis and Elaboration module. The component editor analyzes signals and parameters declared for all modules in the top-level file. If the file is successfully analyzed, the component editor's **Signals** tab lists all design modules in the **Top Level Module** list. If your HDL contains more than one module, you must select the appropriate top-level module from the **Top Level Module** list.

All files are managed in a single table, with options for **Synth** and **Sim**. You can select the **Top** option to select the top-level file for synthesis. When the top-level module is changed, the component editor performs best-effort signal matching against the existing port definitions. If a port is absent from the module, it is removed from the port list. You can use the up and down arrows to specify the HDL file analysis order.

By default, all files are added with both **Synth** and **Sim** options turned on. To add a simulation-only file, turn off the **Synth** option for that file. Files that turn on the **Sim** option are passed to ModelSim® for simulation. To add a synthesis-only file, turn off the **Sim** file option. You can add the **.sdc** file for your component using the **Synth** option. Only files that you mark for **Synth** are added to the **.qip** file for your project.

The component editor determines the signals on the component when only the top-level module or entity is added to the table, but all of the files required for the component must be added for the component to compile in Quartus II software or work in simulation.

Top-Down Design

The **Create HDL Template** button on the **HDL Files** tab allows you to create an HDL template for a component if you have not provided a HDL description for it. Clicking the **Create HDL Template** button shows you the component HDL and lets you choose between Verilog HDL and VHDL. Altera recommends that you define your signals, interfaces, parameters and basic component information, including the component name, before creating the HDL template by clicking **Save**. The component editor writes <***component_name***>**.v** or <***component_name***>**.vhd** to your project directory.

After you have component the component's HDL code, you can add other files that are required to define your component, including the **_hw.tcl** file, and synthesis and simulation files using the **Add** button on the **HDL Files** tab.

Signals Tab

You use the **Signals** tab to specify the purpose of each signal on the top-level component module. If you specified a file on the **HDL Files** tab, the signals on the top-level module appear on the **Signals** tab.

The **Interface** list also allows creation of a new interface so that you can assign a signal to a different interface without first switching to the **Interfaces** tab. Each signal must belong to an interface and be assigned a legal signal type for that interface. In addition to Avalon Memory-Mapped and Streaming interfaces, components typically have clock interfaces, interrupt interfaces, and perhaps a conduit interface for exported signals.

Naming Signals for Automatic Type and Interface Recognition

The component editor recognizes signal types and interfaces based on the names of signals in the source HDL file, if they conform to the following naming conventions:

Signal associated with a specific interface—<*interface type*>_<*interface name*>_<*signal type*>[_n]

For any value of <*interface_name*> the component editor automatically creates an interface by that name, if necessary, and assigns the signal to it. The <*signal_type*> must match one of the valid signal types for the type of interface. Refer to the *Avalon Interface Specifications* for the signal types available for each interface type. You can *append* _n to indicate an active-low signal. Table 6–1 lists the valid values for <*interface_type*>.

Table 6–1. Valid Values for <Interface Type>

Value	Meaning
avs	Avalon-MM slave
avm	Avalon-MM master
ats	Avalon-MM tristate slave
aso	Avalon-ST source
asi	Avalon-ST sink
cso	Clock output
csi	Clock input
coe	Conduit
inr	Interrupt receiver
ins	Interrupt sender
ncm	Nios II custom instruction master
ncs	Nios II custom instruction slave

Example 6–1 shows a Verilog HDL module declaration with signal names that infer two Avalon-MM slaves.

Example 6–1. Verilog HDL Module With Automatically Recognized Signal Names

```
module my_slave_irq_component (

   // Signals for Avalon-MM slave port "s1" with irq

   csi_clockreset_clk; //clockreset clock interface
   csi_clockreset_reset_n;//clockreset clock interface

   avs_s1_address;//s1 slave interface
   avs_s1_read; //s1 slave interface
   avs_s1_write;  //s1 slave interface
   avs_s1_writedata; //s1 slave interface
   avs_s1_readdata; //s1 slave interface
   ins_irq0_irq; //irq0 interrupt sender interface
   );

   input csi_clockreset_clk;
   input csi_clockreset_reset_n;
   input [7:0]avs_s1_address;
   input avs_s1_read;
   input avs_s1_write;
   input [31:0]avs_s1_writedata;
   output [31:0]avs_s1_readdata;
   output ins_irq0_irq;


/* Insert your logic here */

endmodule
```

Templates for Interfaces to External Logic

If the component does not use an HDL file to interface to external logic that is Avalon compatible, you can manually add the signals that comprise the interface to the external logic or use the **Create HDL Template** to generate an HDL template for the component. You connect these signals outside of the SOPC Builder system. If your component uses an Avalon interface to interface outside of SOPC Builder, you can use the Templates menu in the component editor to add typical interface signals to your signal list. There are templates for the following interfaces:

- Avalon-MM Slave
- Avalon-MM Slave with Interrupt
- Avalon-MM Master
- Avalon-MM Master with Interrupt
- Avalon-ST Source
- Avalon-ST Sink

After adding a typical Avalon interface using a template, you can add or delete signals to customize the interface.

Interfaces Tab

The **Interfaces** tab allows you to configure the interfaces on your component and specify a name for each interface. The interface name identifies the interface and appears in the SOPC Builder connection panel. The interface name is also used to uniquely identify any signals that are ports on the top-level SOPC Builder system.

The **Interfaces** tab allows you to configure the type and properties of each interface. For example, an Avalon-MM slave interface has timing parameters that you must set appropriately. The **Interfaces** tab displays waveforms that illustrate the timing that you specified. If you update the timing parameters, the waveforms automatically update to illustrate the new timing. The waveforms are available for the following interface types:

- Avalon Memory-Mapped
- Avalon Memory-Mapped tristate
- Avalon Streaming
- Interrupts

If you convert a component from a **class.ptf** to a **_hw.tcl** file, you may require three interfaces: a clock input, the Avalon slave, and an interrupt sender. A parameter in the interrupt sender must be set to reference the Avalon slave.

Component Wizard Tab

The **Component Wizard** tab provides options that affect the presentation of your new component.

Identifying Information

You can specify information that identifies the component as follows:

- Folder—Specifies the location of the component, determined by the location of the top-level HDL file.
- Class Name—Specifies the name used internally to store the component in the component library. The class name is stored in the **.sopc** file. Use the class name when saving a system that contains an instance of this component. It is also the name you use for the component type when you create a system using a **.tcl** script. If you change the class name of a component, existing **.sopc** files that use the component may break.

  SOPC builder uses the class name and version to find components. If two components with the same class name and version are available to SOPC builder at the same time, the behavior of SOPC builder is undefined.

- Display Name—Specifies the user-visible name for this component in SOPC Builder.
- Version—Specifies the version number of the component.
- Group—Specifies which group in SOPC Builder displays your component in the list of available components. If you enter a previously unused group name, SOPC Builder creates a new group by that name.

- Description—Allows you to describe the component.
- Created By—Allows you to specify the author of the component.
- Icon—Allows you to place an image in the title bar of your component, in place of the MegaCore logo. The icon can be a **.jpg**, **.gif**, or **.png** file. The directory for the icon is relative to the directory that contains the **_hw.tcl** file.
- Data sheet URL—Allows you to specify a URL for the datasheet. You can use this property to specify a file on the internet or in your company's file system. The specified file can be in either **.html** or **.pdf** format. To specify an internet file, begin your path with **http://**, for example: **http://mydomain.com/datasheets/my_memory_controller.html**. To specify a file in your company's file system, you begin you path with **file:///** for Linux and **file:////** for Windows, for example: **file:////company_server/datasheets/my_memory_controller.pdf**. For handwritten **_hw.tcl** files, you can specify a relative path using the following Tcl command: `set_module_property DATASHEET_URL [get_module_property MODULE_DIRECTORY]/<relative_path_to_hw.tcl>`
- Parameters—Allows you to specify the parameters for creating the component, as described in the next section.

Parameters

The **Parameters** table allows you to specify the user-configurable parameters for the component.

If the top-level module of the component HDL declares any parameters (*parameters* for Verilog HD or *generics* for VHDL), those parameters appear in the **Parameters** table. The parameters are presented to you when you create or edit an instance of your component. Using the **Parameters** table, you can specify whether or not each parameter is user-editable.

The following rules apply to HDL parameters exposed via the component GUI:

- Editable parameters cannot contain computed expressions.
- If a parameter <*N*> defines the width of a signal, the signal width must be of the form <*N*-1>:0.
- When a VHDL component is used in a Verilog HDL SOPC Builder system, or vice versa, numeric parameters must be 32-bit decimal integers. When passing other numeric parameter types, unpredictable results occur.

Click **Preview the Wizard** at any time to see how the component GUI appears.

Refer to *Component Interface Tcl Reference* chapter in the *Quartus II Handbook* for detailed information about creating and displaying parameters using Tcl scripts.

Saving a Component

You can save the component by clicking **Finish** on any of the tabs, or by clicking **Save** on the File menu. Based on the settings you specify in the component editor, the component editor creates a component description file with the file name *<class-name>***_hw.tcl**. The component editor saves the file in the same directory as the HDL file that describes the component's hardware interface. If you did not specify an HDL file, you can save the component description file to any location you choose.

You can relocate component files later. For example, you could move component files into a subdirectory and store it in a central network location so that other users can instantiate the component in their systems. The **_hw.tcl** file contains relative paths to the other files, so if you move the **_hw.tcl** file you should move all the HDL and other files associated with it.

Altera recommends that you store **_hw.tcl** files for a project is in the **ip/<*class-name*>** directory for the project. You should store the HDL and other files in the same directory as the **_hw.tcl** file.

Editing a Component

After you save a component and exit the component editor, you can edit it in SOPC Builder. To edit a component, right-click it in the list of available components on the **System Contents** tab and click **Edit Component**.

You cannot edit components that were created outside of the component editor, such as Altera-provided components.

If you edit the HDL for a component and change the interface to the top-level module, you need to edit the component to reflect the changes you made to the HDL.

Software Assignments

You can use Tcl commands to create software assignments.You can register any software assignment that you want, as arbitrary key-value pairs. Example 6–2 shows a typical Tcl API script:

Example 6–2. Typical Software Assignment with Tcl API Scripting

```
set_module_assignment name value
set_interface_assignment name value
```

The result is that the assignments go into the **.sopcinfo** file, available for use for downstream components.

Component GUI

To edit component instance parameters, select a component in the **System Contents** tab of the SOPC Builder window and click **Edit**.

Referenced Documents

This chapter references the following documents:

- *Avalon Component Interfaces Supported in the Component Editor Version 7.2*
- *Avalon Interface Specifications*
- *Component Interface Tcl Reference* chapter in volume 4 of the *Quartus II Handbook*
- *Examples of Changes to Typical Avalon Interfaces for the Component Editor Version 7.2 and Later*
- *Nios II Software Developer's Handbook*
- *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*
- *SOPC Builder Component Development Flow Using the Component Editor Overview*
- *Upgrading Your Component with SOPC Builder Component Editor Version 7.2 and Later*

Document Revision History

Table 6–2 shows the revision history for this chapter.

Table 6–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009, v9.0.0	■ Revised description of the **Create HDL Template** functionality and the Templates menu. ■ Interfaces tab now includes waveforms that illustrate timing parameters. ■ Added reference to *Component Interface Tcl Reference* chapter for detailed information about defining and displaying GUI parameters. ■ Added data sheet URL to Component Wizard tab.	Updated to reflect new functionality.
November 2008, v8.1.0	■ Added information about new HDL template feature ■ Changed page size to 8.5 x 11 inches	—
May 2008, v8.0.0	Extensive edits to this chapter, including: ■ Chapter renumbered. ■ Added new section on software assignments.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

7. Component Interface Tcl Reference

QII54022-9.0.0

Introduction

You define SOPC Builder components by declaring their properties and behaviors in a Hardware Component Description File (**_hw.tcl**). Each **_hw.tcl** file represents one component instance which you can add to an SOPC Builder system. You can also share the components that you design with other designers. For your component to have maximum flexibility, you should consider what aspects of its behavior can be parameterized so that other users can change the default parameterization to address different design requirements.

An SOPC Builder component is usually composed of the following four types of files:

- **_hw.tcl** file—describes the SOPC Builder related characteristics, such as interface behaviors. This file is required.
- HDL files—define the component's functionality as hardware. These files are optional.
- **_sw.tcl**—used by the software build tools to compile the component driver code. This file is optional.
- Component driver files—defines the component register map and driver software to allow software to control the component. These files are optional.

This chapter discusses the following topics:

- "Information in a Hardware Component Description File" on page 7–1
- "Component Phases" on page 7–2
- "Writing a Hardware Component Description File" on page 7–2
- "Overriding Default Behaviors" on page 7–9
- "Hardware Tcl Command Reference" on page 7–12

Information in a Hardware Component Description File

A typical **_hw.tcl** file contains the following information:

- Basic component information—includes the component's name, version, and description, a link to its documentation, and pointers to HDL implementation files for synthesis and simulation.
- Parameter Declarations—Parameters are values that the user of your component can set that affect how the component is implemented, such as the size of a memory. Properties of each parameter include the parameter's name, whether or not it is visible, and, if visible, the text to display when describing it. When the SOPC Builder system is generated, the parameters can be applied to the component as Verilog HDL parameters or VHDL generics.

- Interface Properties—The interfaces of a component define how to connect it to the rest of the system and determine how other components in the system interact with it. When you add interfaces to a component, you declare which signals make up each interface. You also define interface properties, such as wait states for an Avalon® Memory-Mapped (Avalon-MM) interface.

Component Phases

The following section describes the distinct phases in the development of an SOPC Builder component.

- Main Program—SOPC Builder first discovers a component and adds it to the component library. The **_hw.tcl** file is executed and the Tcl statements provide non-instance-specific information to SOPC Builder. During this phase, some component interfaces may be incompletely described and ports may have a width of 0 or -1 to indicate that they are variable.
- Validation—Validation allows the component to generate error, warning, or informational messages. Validation occurs when an instance of a component is created, when its parameters are changed, or when some other property of the system is changed.
- Elaboration—Elaboration occurs as SOPC Builder queries a component for its interface information. Elaboration typically occurs immediately after validation and before generation. Interfaces defined in the main program can be enabled or disabled during elaboration. Depending on the validation callback code, elaboration and validation may alternate a few times. Elaboration and validation always occur before generation. Once elaboration is complete, the component must be completely described. For example, all port widths must have positive values.
- Generation—Generation creates all the information that the Quartus® II software and HDL simulator require. The required files typically include VHDL or Verilog HDL files, simulation models, timing constraints, and other information.
- Editor—After an instance of your component has been added to an SOPC Builder system, allows the user of your component to edit the GUI that displays the parameterization. You can change the appearance of the default editor to make it easier to use.

Writing a Hardware Component Description File

This section provides detailed information about **_hw.tcl** files and describes the default behavior of a component in all five phases. The following example uses a simple UART with some simple parameterization.

Providing Basic Information

A typical **_hw.tcl** file first declares basic information, such as the name, location, and the files it includes. Example 7–1 provides sample Tcl code for basic component information.

The Tcl scripts shown in each example are each part of the single **_hw.tcl** that defines a component. `String` arguments must only be enclosed in quotes if they include embedded spaces.

An excellent source of information about Tcl syntax is the Tcl Developer Xchange website.

Example 7–1. Basic Information for _hw.tcl File

```
# The name and version of the component
set_module_property NAME example_uart
set_module_property VERSION 1.0

# The name of the component to display in the library
set_module_property DISPLAY_NAME "Example Component"

# The component's description.
set_module_property DESCRIPTION "An Example Component"

# The component library group that component belongs to
set_module_property GROUP Examples
```

Declaring Parameters

By including configuration parameters in your **_hw.tcl** file, you allow other users of your component to parameterize it differently. Component users can enter integer parameters as decimal, binary, or hexadecimal values. You can specify binary values using the `b'` notation, for example: `b'1111`. You can specify hexadecimal values with either the `0x` or `'h` prefix, for example: `0x100` or `'h100`. Example 7–2 illustrates the use of parameters that can be configured by other users of your component.

Example 7–2. Declaring Parameters

```
# Declare Baud Rate parameter as an integer with a default value of 9600.
add_parameter BAUD_RATE int 9600
# Display this parameter as "Baud Rate" in the Parameter Editor.
set_parameter_property BAUD_RATE DISPLAY_NAME "Baud Rate (bps)"
# We only support three baud rates
set_parameter_property BAUD_RATE ALLOWED_RANGES {9600 19200 38400}
```

You can use the `SYSTEM_INFO` parameter property in conjunction with the `set_parameter_property` command to introduce custom parameters that are part of your component definition. `SYSTEM_INFO` requires the *`<info-type>`* argument that can take on many different values. In some cases, `SYSTEM_INFO` requires more than one argument. For example, when the *`<info-type>`* is `ADDRESS_MAP`, you must specify the Avalon-MM master whose address map you need. Example 7–3 illustrates the use of the `SYSTEM_INFO` parameter.

Example 7–3. Syntax of Tcl Command using the SYSTEM_INFO Parameter

```
set_parameter_property my_parameter SYSTEM_INFO {<info-type> [<arg>]}
```

The following types of system information are defined:

- `CLOCK_RATE` (`Integer` or `String`)—Assigns a positive number which is the clock frequency in Hz to the clock input interface you specify. Assigns 0 if the clock rate is not known.

```
set_parameter_property <my_parameter> SYSTEM_INFO {CLOCK_RATE
<my_clk>}
```

- `CLOCK_DOMAIN` (`Integer`)—Assigns an integer representing the clock domain to the parameter you specify. You can use this command to determine whether multiple interfaces in your module are on the same clock domain. The absolute value of the integer value is arbitrary, but if two interfaces are on the same clock domain, the `CLOCK_DOMAIN` value is guaranteed to be the same and greater than zero.

```
set_parameter_property <my_parameter> SYSTEM_INFO {CLOCK_DOMAIN
<my_clk>}
```

- `RESET_DOMAIN` (`Integer`)—Assigns an integer representing the reset domain to the parameter you specify. You can use this command to determine whether multiple interfaces in your module are on the same reset domain. The absolute value of the integer value is arbitrary, but if two interfaces are on the same reset domain, the `RESET_DOMAIN` value is guaranteed to be the same and greater than zero.

```
set_parameter_property <my_parameter> SYSTEM_INFO {RESET_DOMAIN
<my_reset>}
```

- `ADDRESS_WIDTH` (`Integer`)—Assigns an integer to the parameter that you specify that is the number of bits an Avalon-MM master must drive to address all of its slaves, using byte addresses.

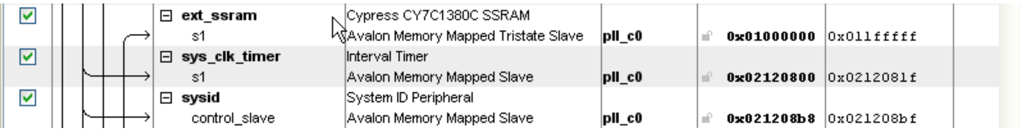
```
set_parameter_property <my_parameter> SYSTEM_INFO (ADDRESS_WIDTH
<my_avalon-mm_master>)
```

- `ADDRESS_MAP` (`String`)—Assigns an XML formatted string describing the address map to the parameter you specify.

```
set_parameter_property <my_parameter> SYSTEM_INFO {ADDRESS_MAP
<my_avalon-mm_master>}
```

  The XML code describing each slave includes, its name, start address, and end address + 1. Figure 7–1 shows a portion of an SOPC Builder system with three Avalon-MM slave devices.

Figure 7–1. SOPC Builder System with Three Avalon-MM Slaves



Example 7–4 shows the XML that describes the address map for the Avalon-MM master that accesses these slaves. The format of the XML string provided may differ from that described here, it may have different white space between the elements and could include additional attributes or elements.

Altera recommends that you use the code provided in the description of “decode_address_map” on page 7–24 to enumerate over the components within an address map, rather than writing your own parser.

Example 7–4. Address Map for an Avalon-MM Master

```
<address-map>
  <slave name='ext_ssram' start='0x01000000' end='0x01200000' />
  <slave name='sys_clk_timer' start='0x02120800' end='0x02120820' />
  <slave name='sysid' start='0x021208B8' end='0x021208C0' />
</address-map>
```

- `MAX_SLAVE_DATA_WIDTH` (`Integer`)—Assigns an integer to the parameter you specify that is the data width of the widest slave connected to the specified Avalon-MM master.

  ```
  set_parameter_property <my_parameter> SYSTEM_INFO
  {MAX_SLAVE_DATA_WIDTH <my_avalon_mm_master>}
  ```

- `INTERRUPTS_USED` (`Integer` or `String`)—Creates a mask indicating which bits of the interrupt receiver vector are connected to an interrupt sender. This mask is assigned to the parameter you specify. You can use this interrupt mask to optimize logic that handles interrupts.

  ```
  set_parameter_property <my_parameter> SYSTEM_INFO (INTERRUPTS_USED
  <my_interrupt_receiver>}
  ```

- `DEVICE_FAMILY` (`String`)—Assigns the family name (not the specific device part number) of the currently selected device to the parameter you specify.

  ```
  set_parameter_property <my_parameter> SYSTEM_INFO {DEVICE_FAMILY}
  ```

- `DEVICE_FEATURES` (`String`)—Creates a list of key/value pairs delineated by spaces indicating whether a particular device feature is available in the currently selected device family. The format of the list is suitable for passing to the Tcl `array set` command. This list is assigned to the parameter you specify. The following features are supported: M512_MEMORY, M4K_MEMORY, M9K_MEMORY, M144K_MEMORY, MRAM_MEMORY, MLAB_MEMORY, ESB, DSP, and EMUL

  ```
  set_parameter_property <my_parameter> SYSTEM_INFO {DEVICE_FEATURES}
  ```

Declaring Interfaces

Most components require a clock input interface. To declare an interface, declare its properties and indicate which signals belong to it. For components with HDL files, `quartus_map` determines each port's direction and width. The interface declaration statement includes the name of the interface, the interface direction, and the clock interface with which it is associated. For interfaces that aren't associated with clocks (such as clock interfaces themselves), omit the associated clock interface, or use the word *asynchronous*.Example 7–5 illustrates interface declaration.

Example 7–5. Declare Interfaces

```
# Declare the clock sink interface, "clock_sink", type=clock, direction=sink

add_interface clock_sink clock sink

# The clock interface has two signals, named "clk" and "reset_n" of types "clk" "reset_n"

add_interface_port clock_sink clk clk input 1
add_interface_port clock_sink reset_n reset_n input 1

# Declare the Avalon slave interface, name=avalon_slave_0, type=avalon, directon=slave,
# associated with the clock_sink clock interface.

add_interface avalon_slave_0 avalon slave clock_sink

# Set a number of properties about the Avalon Slave interface

set_interface_property avalon_slave_0 writeWaitTime 0
set_interface_property avalon_slave_0 addressAlignment DYNAMIC
set_interface_property avalon_slave_0 readWaitTime 1
set_interface_property avalon_slave_0 readLatency 0

# Declare all the signals that belong to my Avalon Slave interface

add_interface_port avalon_slave_0 my_readdata readdata output 8
add_interface_port avalon_slave_0 my_read read input 1
add_interface_port avalon_slave_0 my_write write input 1
add_interface_port avalon_slave_0 my_waitrequest waitrequest output 1
add_interface_port avalon_slave_0 my_address address input 24
add_interface_port avalon_slave_0 my_writedata writedata input 8
```

Adding Files and Guiding Generation

Component description files typically provide all of the information required for generation and downstream tools, identifying the files used by the component such as HDL files and **.sdc** constraint files. You also identify which of the added files is the top-level HDL file and specify which verilog module or VHDL entity within that file is the top-level module for the component. Example 7–6 illustrates the files that are typically required for generation and downstream tools.

Example 7–6. Add Files

```
# Add the HDL file to the component,to be used for synthesis and simulation.

add_file simple_uart.v {SYNTHESIS SIMULATION}

# Add the Timequest file with Quartus timing constraints.

add_file simple_uart.sdc SDC

# Add top-level HDL file that describes the component, name of the top-level module/entity

set_module_property TOP_LEVEL_HDL_FILE simple_uart.v
set_module_property TOP_LEVEL_HDL_MODULE simple_uart
```

Default Behaviors

The **_hw.tcl** file described in the previous section has default behaviors during the validation, elaboration, generation, and editor phases. These default behaviors apply to instances of a component. This section describes the default SOPC Builder behaviors for each of these phases. To override these default behaviors, refer to "Overriding Default Behaviors" on page 7–9.

Validation Phase Behavior

SOPC Builder's default validation checks each parameter value against its ALLOWED_RANGES property. If the values specified are outside the allowed ranges, an error message displays.

The ALLOWED_RANGES property of each parameter is a list of ranges that the parameter can take on, where each range is a single value, or a range of values defined by a start and end value separated by a colon. Table 7–1 shows some examples of values the ALLOWED_RANGES property can take.

Table 7–1. ALLOWED_RANGES Property

ALLOWED_RANGES	Meaning
{a b c}	a or b or c
{1 2 4 8 16}	1, 2, 4, 8, or 16.
1:3	1 through 3, inclusive
{1 2 3 7:10}	1, 2, 3, or 7 through 10 inclusive

Elaboration Phase Behavior

SOPC Builder's default elaboration process calls quartus_map to determine the correct port widths. Because calling quartus_map can be slow, you can set the AFFECTS_ELABORATION property to false for parameters which do not affect port widths, this will prevent re-elaboration when one of these parameters changes.

Generation Phase Behavior

SOPC Builder's default generation does one of the following:

- If the component defines the TOP_LEVEL_HDL_MODULE property, SOPC Builder creates a Verilog HDL or VHDL wrapper module to instantiate the top-level module and applies the parameters as selected by the user of your component. SOPC Builder does not apply parameters in the wrapper if they are not declared in the underlying HDL file.

  or

If the component does not define the TOP_LEVEL_HDL_MODULE property, but instead sets the INSTANTIATE_IN_SYSTEM_MODULE_module property to false, the module is not instantiated inside the SOPC Builder system and a wrapper file is not created. Rather, the interface to the module is exported to the top-level of the SOPC Builder system, and the module must be connected outside the system.

Editor Phase Behavior

SOPC Builder's default editor phase behavior is to use all of the parameter definitions to display the parameterization GUI. The properties of the parameters guide SOPC Builder when it builds the default GUI. Table 7–4 on page 7–20 lists the properties of parameters.

You can place parameters in logical groups and provide images to create a custom GUI for your component. Example 7–7 defines four parameters and illustrates the use of the add_display_item command and the DISPLAY_HINT and ALLOWED_RANGES parameters.

Example 7–7. Defining and Customizing GUI Parameters

```
# provide an icon for the sound group
add_display_item icon Speaker speaker-image speaker.png
add_parameter sound string 0 0
add_parameter volume_control boolean 0 0
add_parameter separate_control string 0 0

# Setup DISPLAY_NAMES for the parameters
set_parameter_property sound DISPLAY_NAME Audio
set_parameter_property volume_control DISPLAY_NAME "Include Volume Control Interface"
set_parameter_property separate_control DISPLAY_NAME "Treble/Bass Controls"

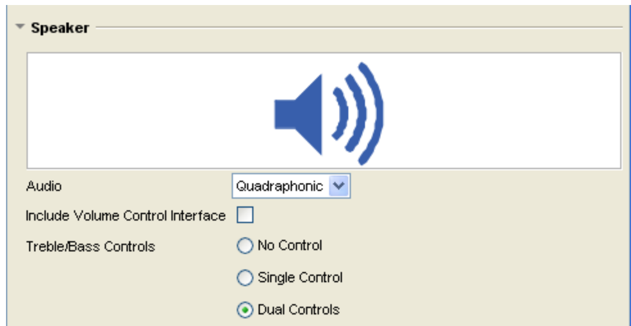
# Display all parameters in the Speaker group
add_display_item parameter Speaker sound
add_display_item parameter Speaker volume_control
add_display_item parameter Speaker separate_control

# There are 4 choices for the sound parameter.
# Strings with internal spaces require double quotes
set_parameter_property sound allowed_ranges {"0:No Audio" 1:Monophonic 2:Stereo
4:Quadraphonic}
set_parameter_property separate_control allowed_ranges {"No Control" "Single Control"
"Dual Controls"}

#Specify how parameters should be displayed
set_parameter_property volume_control DISPLAY_HINT boolean
set_parameter_property separate_control DISPLAY_HINT radio
```

Figure 7–2 shows the GUI that the Tcl commands in Example 7–4 produces.

Figure 7–2. Parameter GUI for Audio Component



Overriding Default Behaviors

You can override each of the default behaviors by using callbacks. This section explains how to write callback procedures for each phase of component development.

Validation Callback

You can use the validation callback to provide validation that extends beyond the default range checking. A validation callback is defined by setting the `VALIDATION_CALLBACK` module property to be the name of the validation callback procedure, as shown in Example 7–8. This validation procedure displays an error if you select a baud rate of 38400 and odd parity.

You can also use the validation callback to set the value of derived parameters. Derived parameters are parameters that are derived from other parameters; their values are not editable and are not saved in the system-on-a-programmable-chip (**.sopc**) file. You indicate that a parameter is derived by setting the parameter's `DERIVED` property to `true`. In Example 7–8 `BAUDRATE_PRESCALE` is a derived parameter whose value is 1/16 of the value of the BAUDRATE parameter.

Example 7–8. Custom Validation Callback Function

```
# Declare the validation callback.

# Declare the validation callback.
set_module_property VALIDATION_CALLBACK my_validation_callback

# Add the BAUDRATE_PRESCALE parameter
add_parameter BAUDRATE_PRESCALE int 600
set_parameter_property BAUDRATE_PRESCALE DERIVED true

# Add the PARITY parameter
add_parameter PARITYd string ODD
set_parameter_property PARITY ALLOWED_RANGES {EVEN ODD}

# The validation callback
proc my_validation_callback {} {
    # Get the current value of parameters we care about
    set br [get_parameter_value BAUD_RATE]
    set p [get_parameter_value PARITY]
    # Display an error for invalid combinations.
    if {($br==38400) && ($p=="ODD")} {
        send_message warning "Odd parity at 38400 bps is not supported."
    }
    # Set the value of our derived parameter
    set bp [expr $br / 16]
    set_parameter_value BAUDRATE_PRESCALE $bp
}
```

Elaboration Callback

You can use an elaboration callback to change interface properties or add new interfaces as a function of parameter values. You define an elaboration callback by setting the ELABORATION_CALLBACK module property to the name of the elaboration callback function, as shown in Example 7–9. You can enable and disable interfaces from the elaboration callback if they are only needed for some parameterizations of the component. Example 7–9 shows how an Avalon-MM slave interface can be included in an instance of the component, based on the USE_STATUS_INTERFACE parameter.

The elaboration callback is not allowed to use parameters marked as AFFECTS_ELABORATION=false. It will not be called if such a parameter is changed.

Example 7–9. Elaboration Callback

```
# Declare the callback.
set_module_property ELABORATION_CALLBACK my_elaboration_callback
# Add the USE_STATUS_INTERFACE parameter
add_parameter USE_STATUS_INTERFACE boolean
# Declare the status slave interface
add_interface status_slave avalon slave clock_sink
set_interface_property status_slave enabled false

# The elaboration callback
proc my_elaboration_callback {} {
    # Get the current value of parameters we care about
    set use_status [get_parameter_value USE_STATUS_INTERFACE]
    # Optionally add an interface
    if { $use_status } {
   set_interface_property status_slave enabled false
   # Set interface properties
   set_interface_property status_slave writeWaitTime 0
   set_interface_property status_slave readWaitTime 1
   # Declare signals
   add_interface_port status_slave st_readdata readdata output 16
   add_interface_port status_slave st_read read input 1
   add_interface_port status_slave st_write write input 1
   add_interface_port status_slave st_waitrequest waitrequest output 1
   add_interface_port status_slave st_address address input 24
   add_interface_port status_slave st_writedata writedata input 16
    }
}
```

Generation Callback

If you define a generation callback, SOPC Builder does not generate an HDL wrapper file to apply parameter values to your component. Instead, it calls the generation callback you defined during the generation phase, allowing the component to programmatically generate its HDL. A generation callback is defined by setting the GENERATION_CALLBACK module property to be the name of the generation callback function, as Example 7–10 illustrates.

Generation callbacks typically retrieve the current value of the component's parameters and the generation properties that guide the generation process, and then generate the HDL files and supporting files in Tcl or by calling an external program. The callback procedure also reports the required files to SOPC Builder with the `add_files` command. Any files added in the generation callback are in addition to the files added in the main body of the **_hw.tcl** file.

The generation callback must write <*output_name.v*> for Verilog or <*output_name.vhd*> for VHDL to the specified <*output_directory*>. This file is a parameterized instance of the component. Other supporting files, such as **.hex** files to initialize memory, may be written to <*output_directory*>. These file names must begin with <*output_name*>. If the supporting files are the same for all parameterizations of the component, you add them from the main program rather than the generation callback. If your system includes multiple instantiations of a component with different parameterizations, you must add the supporting files from the main program to prevent failures.

Example 7–10. Generation Callback Example

```
set_module_property GENERATION_CALLBACK my_generate

# My generation method

proc my_generate {} {
  send_message info "Starting Generation"

  # get generation settings

  set language [get_generation_property HDL_LANGUAGE]
  set outdir [get_generation_property OUTPUT_DIRECTORY ]
  set outputname [get_generation_property OUTPUT_NAME ]

  # get parameter values

  set p1 [get_parameter_value PARAMETER_ONE]
  set csr [get_parameter_value CSR_ENABLED]

  # Do HDL generation with perl

  # add_file creates files relative to the _hw.tcl directory; therefore specify $outdir
  # for synthesis and simulation files

  exec perl my_generate.pl lang=$language dir=$outdir name=$outputname p1=$p1 csr=$csr
  add_file ${outdir}${outputname}.v SYNTHESIS
  add_file ${outdir}${outputname}_sim.v SIMULATION
}
```

Editor Callback

You can use the editor callback procedure to override the parameterization GUI. An editor callback is defined by setting the `EDITOR_CALLBACK` module property to the name of your editor callback procedure, as shown in the Example 7–11. If the editor callback is defined, SOPC Builder calls the editor callback any time the parameterization GUI is displayed, typically when the component is added to a system or updated after it is in the system.

To display your custom GUI, the editor callback must call another program. Typically, an editor callback provides the current parameter values to your program via the command line and collects the new parameter values via `stdout`. The editor callback then uses the `set_parameter_value` command to update SOPC Builder with the new parameter values.

The editor callback returns one of the following three values:

- OK—indicates that the results of the edit should be applied.
- CANCEL—indicates that the system should revert to the state it was in before the editor callback was called.
- ERROR—indicates that the GUI was unable to launch. An appropriate error message should be displayed.

If no value is returned, OK is assumed.

Example 7–11. Editor Callback

```
set_module_property EDITOR_CALLBACK my_editor

# Define Module parameters.

add_parameter PARAMETER_ONE integer 32 "A parameter"
add_parameter CSR_ENABLED boolean true "Enable CSR interface"

# My editor method

proc my_editor {} {

# get parameter values

  set p1 [ get_parameter_value PARAMETER_ONE ]
  set csr [ get_parameter_value CSR_ENABLE ]

  # Display UI, populated with current parameter values.
  # The stdout returned by the UI program includes the new paramter values.

  set result = [exec my_component_ui.exe p1=$p1 csr=$csr]

  # Use the fictional "parse_for_new_value" procedure to parse the returned text for the
  # new parameter values.

  set p1 [parse_for_new_value $result p1]
  set csr [parse_for_new_value $result csr]

# Return the new parameter values to SOPC Builder

  set_parameter_value PARAMETER_ONE $p1
  set_parameter_value CSR_ENABLED $csr
  return OK

}
```

Hardware Tcl Command Reference

This section provides a reference for all hardware Tcl commands, as follows:

- "Module Definition" on page 7–14
- "Parameters" on page 7–19
- "Interfaces and Ports" on page 7–25
- "get_interface_assignment" on page 7–30

The description of each command indicates during which phases it is available: in the main body of the program (main), or during the validation, elaboration, generation, and editor callback phases, or any combination. Table 7–2 summarizes the commands and provides a reference to the full description.

Table 7–2. Command Summary *(Note 1)* (Part 1 of 2)

Command	Full Description
Module Definition	
get_module_properties	page 7–14
get_module_property *<propertyName>*	page 7–15
set_module_property *<propertyName> <propertyValue>*	page 7–16
get_module_ports	page 7–16
get_module_assignment *<moduleName>*	page 7–16
set_module_assignment *<moduleName> [value]*	page 7–17
add_file filename *[<fileProperties> . . .]*	page 7–17
get_files	page 7–18
get_file_property *<filename> <propertyName>*	page 7–18
set_file_property *<filename> <propertyName> <propertyValue>*	page 7–18
send_message *<messageLevel> <messageText>*	page 7–19
Parameters	
add_parameter *<parameterName> <parameterType> [<defaultValue> <description>]*	page 7–22
get_parameters	page 7–22
get_parameter_properties	page 7–19
get_parameter_property *<parameterName> <propertyName>*	page 7–23
set_parameter_property *<parameterName> <propertyName> <value>*	page 7–23
get_parameter_value *<parameterName>*	page 7–23
set_parameter_value *<parameterName> <value>*	page 7–24
decode_address_map *<address_map_XML_string>*	page 7–24
add_display_item *<groupName> <id> <type> [<additionalInfo>]*	page 7–24
Interfaces and Ports	
add_interface *<interfaceName> <interfaceType> <direction> [<associatedClock>]*	page 7–26
get_interfaces	page 7–26
get_interface_properties *<interfaceName>*	page 7–27
get_interface_property *<interfaceName> <propertyName>*	page 7–27
set_interface_property *<interfaceName> <propertyName> <value>*	page 7–27
add_interface_port *<interfaceName> <portName> <portRole> [<direction> <width>]*	page 7–28
get_interface_ports *[<interfaceName>]*	page 7–28
get_port_properties	page 7–29
get_port_property *<portName> <propertyName>*	page 7–29
set_port_property *<portName> <propertyName> [<value>]*	page 7–30
get_interface_assignment *<interfaceName> <name>*	page 7–30
set_interface_assignment *<interfaceName> <name> [<value>]*	page 7–31
Generation	

Table 7–2. Command Summary *(Note 1)* (Part 2 of 2)

Command	Full Description
`get_generation_property <propertyName>`	page 7–32
`get_generation_properties`	page 7–31
`get_project_property <propertyName>`	page 7–32

Note to Table 7–2:

(1) Arguments enclosed in []'s are optional

Module Definition

This section provides information about the commands that you use to define and query a module.

get_module_properties

This command returns the names of all the available module properties as a list of strings. You can use the `get_module_property` and `set_module_property` commands to get and set values of individual properties. The value returned by this command is always the same for a particular version of SOPC Builder.

get_module_properties	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	`get_module_properties`
Returns	list of strings
Arguments	None
Example	`get_module_properties`

Table 7–3 lists the available module properties, their use, and the phases in which they can be set.

Table 7–3. Module Properties (Part 1 of 2)

Property Name	Property Type	Can Be Set	Description
`NAME`	`String`	Main program	The name of the module, such as `my_sopc_component`.
`DISPLAY_NAME`	`String`	Main program	The name to display when referencing the module, such as "My SOPC Component."
`VERSION`	`String`	Main program	The module's version, such as 8.0.
`AUTHOR`	`String`	Main program	The module's author.
`DESCRIPTION`	`String`	Main program	The description of the module, such as "Example SOPC Builder Module."
`GROUP`	`String`	Main program	The component group that the module belongs to, such as "Example Components."
`ICON_PATH`	`String`	Main program	A path to an icon to display in the module's parameter editor.

Table 7–3. Module Properties (Part 2 of 2)

Property Name	Property Type	Can Be Set	Description
DATASHEET_URL	String	Main program	A path to the module's data sheet, for example: http://www.mydomain.com/my_memory_controller.html.
EDITABLE	Boolean	Main program	Indicates if the component is editable in the component editor.
MODULE_TCL_FILE	String	Can only be read, not set	The path to the **_hw.tcl** file. When possible, all other files should be specified relative to the **_hw.tcl** file.
MODULE_DIRECTORY	String	Can only be read, not set	The directory containing the **_hw.tcl** file.
TOP_LEVEL_HDL_FILE	String	Main program	Indicates which of the files added by the `add_file` command contains the module's top-level HDL.
TOP_LEVEL_HDL_MODULE	String	Main program	Indicates the name of the top-level module which must be defined in the module's top-level HDL file.
INSTANTIATE_IN_SYSTEM_MODULE	Boolean	Main program	When `false` the instances of the module are not included in the generated system interconnect fabric. Instead, interfaces to the module are exported out of the top-level of the SOPC Builder system.
VALIDATION_CALLBACK	String	Main program	The name of the validation callback. The default validation is used if this property is not set.
EDITOR_CALLBACK	String	Main program	The name of the editor callback. The default parameterization UI is called if this property is not set.
ELABORATION_CALLBACK	String	Main program	The name of the elaboration callback. The default elaborations used if this property is not set.
GENERATION_CALLBACK	String	Main program	The name of the generation callback. The default generation is used if this property is not set.

get_module_property

This command returns the value of a single module property.

get_module_property		
Callback availability	Main, validation, elaboration, generation, and editor	
Usage	`get_module_property <propertyName>`	
Returns	`String` or `boolean`	
Arguments	`propertyName`	One of the properties listed in Table 7–3 on page 7–14
Example	`set my_name [get_module_property NAME]`	

set_module_property

This command allows you to set the values for module properties.

set_module_property		
Callback availability	Main program	
Usage	`set_module_property` <*propertyName*> <*propertyValue*>	
Returns	None	
Arguments	`propertyName`	One of the properties listed in Table 7–3 on page 7–14
	`propertyValue`	The new value of the property
Example	`set_module_property VERSION 8.0`	

get_module_ports

This command returns a list of the names of all the ports which are currently defined.

get_module_ports		
Callback availability	Main, validation, elaboration, generation, and editor	
Usage	`get_module_ports`	
Returns	List of strings	
Arguments	none	(The ports are implicitly those for the current module.)
Example	`get_module_ports`	

get_module_assignment

This command returns the value of the specified argument. You can use the `get_module_assignment` and `set_module_assignment` and the `get_interface_assignment` and `set_interface_assignment` commands to transfer information about hardware components to embedded software tools and applications.

For more information about specifying information for software tools, refer to *Publishing Component Information to Embedded Software* in the *Nios II Software Developer's Handbook - Studio Edition*.

get_module_assignment		
Callback availability	Main and validation	
Usage	`get_module_assignment` <*name*>	
Returns	`String`	
Arguments	`name`	The name whose value is being retrieved
Example	`get_module_assignment embedded.sw.CMacro.colorSpace`	

set_module_assignment

This command sets the value of the specified argument.

set_module_assignment		
Callback availability	Main and validation	
Usage	`set_module_assignment` *<name> [<value>]*	
Returns	None	
Arguments	`name`	The name whose value is being set
	`value`	The value of the *<name>* argument
Example	`set_module_assignment embedded.sw.CMacro.colorSpace CMYK`	

add_file

This command adds a synthesis, simulation, or TimeQuest constraints file to the module. Files added in the main program cannot be removed. Adding files in the generation callback allows the included files to be a function of the parameter set or to be a result of generation. Files added in callbacks are in addition to any files added in the main program.

add_file		
Callback availability *(1)*	Main, elaboration, and generation	
Usage	`add_file filename` *[<fileProperties> . . .]*	
Returns	None	
Arguments	`filename`	The file name to be added, relative to the directory containing the **_hw.tcl** file
	`fileProperties`	Files support the following 4 properties: ■ `SIMULATION`—File for simulation ■ `SYNTHESIS`—File for synthesis ■ `SDC`—TimeQuest constraints
Example	`add_file my_component.v {SIMULATION SYNTHESIS}`	

Note:

(1) Beginning in version 9.1 of SOPC Builder, the `add_file` command will be restricted to main and generation callbacks.

get_files

This command returns a list of all the files that have been added to the module.

get_files	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	`get_files`
Returns	List of strings
Arguments	None
Example	`set list_of_files [get_files]`

get_file_property

This command returns the value of a single file property. The file name passed as an argument can be a partial as long as it is unique. For example, if the full file name is **/components/my_file.v**, **my_file.v** is sufficient.

get_file_property		
Callback availability	Main, validation, elaboration, generation, and editor	
Usage	`get_file_property` *`<filename> <propertyName>`*	
Returns	None	
Arguments	`filename`	The file name whose properties are being retrieved
	`propertyName`	The filename property whose value is being retrieved
Example	`set forSynthesis [get_file_property my_file.v SYNTHESIS]`	

set_file_property

This command sets the value of a single file property. The file name passed to the function can be a partial file name as long as it is unique. For example, if the full file name is **/components/my_file.v**, **my_file.v** is sufficient. The available properties are described in the `add_files` command.

set_file_property		
Callback availability	Main, elaboration, and generation	
Usage	`set_file_property` *`<filename> <propertyName> <propertyValue>`*	
Returns	None	
Arguments	`filename`	The file name whose properties are being retrieved
	`propertyName`	Name of the file property whose value is being retrieved
	`propertyValue`	Value to set for the file property
Example	`set_file_property my_file.v SYNTHESIS true`	

send_message

This command sends a message to the user of the component.

send_message		
Callback availability	Main, validation, elaboration, generation, and editor	
Usage	send_message *<messageLevel>* *<messageText>*	
Returns	None	
Arguments	messageLevel	The following 4 message levels are supported: ■ Error—provides an error message. The SOPC Builder system cannot be generated while there are error messages. ■ Warning—provides a warning message. ■ Info—provides an informational message. ■ Debug—provides messages when debug mode is enabled.
	messageText	The text of the message
Example	send_message Error "param1 must be greater than param2."	

Parameters

Parameters allow users of your component to affect its operation in the same manner as Verilog HDL parameters or VHDL generics.

get_parameter_properties

This command returns a list of all the available parameter properties as a list of strings. The get_parameter_property and set_parameter_property commands are used to get and set the values of these properties, respectively.

get_parameter_properties	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	get_parameter_properties
Returns	List of parameter property names
Arguments	None
Example	set property_summary [get_parameter_properties]

Table 7–4 describes the properties available to describe the behaviors of each of the parameters you can specify, their use, and when they can be set.

Table 7–4. Parameter Properties (Part 1 of 2)

Property Name	Property Type	Can Be Set	Description
DISPLAY_NAME	String	Main program	The text string to use when displaying the parameter.
ALLOWED_RANGES	String	Main program	Indicates the range or ranges that the parameter value can have. For integers, The ALLOWED_RANGES property is a list of ranges that the parameter can take on, where each range is a single value, or a range of values defined by a start and end value separated by a colon, such as 11:15. This property can also specify legal values and display strings for integers, such as {0:None 1:Monophonic 2:Stereo 4:Quadrophonic} meaning 0,1,2,4 are the legal values. Refer to Example 7–7 and Figure 7–2 for examples illustrating the use of this property.
GROUP	String	Main program	The group in which to display the parameter. This property is deprecated. Use add_display_item instead.
IS_HDL_PARAMETER	Boolean	Main program	When true, the parameter must be passed to the HDL component description. The default value is false false if there is a generation callback. Or, its value is calculated by analyzing the HDL if you have specified a top-level HDL file.
AFFECTS_ELABORATION (1)	Boolean	Main program	Set AFFECTS_ELABORATION to false for parameters that do not affect the external interface of the module. An example of a parameter that does not affect the external interface is isNonVolatileStorage. An example of a parameter that does affect the external interface is width. When the value of a parameter changes, if that parameter has set AFFECTS_ELABORATION=false, the elaboration phase (calling the callback or hardware analysis) is not repeated, improving performance. Because the default value of AFFECTS_ELABORATION is true, the provided HDL file is normally re-analyzed to determine the new port widths and configuration every time a parameter changes.
VISIBLE	Boolean	Main program, validation callback	Indicates whether or not to display the parameter in the parameterization GUI.
ENABLED	Boolean	Main program, validation callback	When false, the parameter is disabled, meaning that it is displayed, but greyed out indicating that it is not editable, on the parameterization GUI.
DERIVED	Boolean	Main program	When true, indicates that the parameter value does not need to be stored, typically because it is set from the validation callback. The default value is false.

Table 7–4. Parameter Properties (Part 2 of 2)

Property Name	Property Type	Can Be Set	Description
DISPLAY_HINT	String	Main program	Provides a hint about how to display a property. The following values are possible: ■ boolean—for integer parameters whose value can be 0 or 1. The parameter displays as a checkbox. ■ radio—displays a parameter with a list of values as radio buttons instead of a drop-down list. ■ hexadecimal—for integer parameters, display and interpret the value as a hexadecimal number, for example: 0x00000010 instead of 16. Refer to Example 7–7 and Figure 7–2 for examples illustrating the use of this property.
SYSTEM_INFO	String	Main program	Allows you to assign information about your system to a parameter that you define. SYSTEM_INFO requires an argument specifying the type of information requested, *<info-type>*. *<info-type>* may also take an argument. The syntax of the Tcl command is: set_parameter_property my_parameter SYSTEM_INFO {*<info-type> [<arg>]*} The following values for *<info-type>* are predefined: ■ CLOCK_RATE ■ CLOCK_DOMAIN ■ RESET_DOMAIN ■ ADDRESS_WIDTH ■ ADDRESS_MAP ■ MAX_SLAVE_DATA_WIDTH ■ INTERRUPTS_USED ■ DEVICE_FAMILY ■ DEVICE_FEATURES For more information about the SYSTEM_INFO and the *<info_type>* argument refer to "Declaring Parameters" on page 7–3.

Note to Table 7–4:

(1) The AFFECTS_ELABORATION property was called AFFECTS_PORT_WIDTHS before version 9.0 of the Quartus II software.

add_parameter

This command adds a parameter to your component.

add_parameter		
Callback availability	Main program	
Usage	add_parameter *<parameterName> <parameterType> [<defaultValue> <description>]*	
Returns	None	
Arguments	parameterName	A name that you, the component author, choose for your parameter
	parameterType	The following 7 types are supported: ■ Integer ■ Natural ■ Positive ■ Boolean ■ Std_logic (VHDL based components only) ■ Std_logic_vector (VHDL based components only) ■ String
	defaultValue	The default length of the parameter is derived from its range.
	description	Explains the use of the parameter
Example	add_parameter seed integer 17 "The seed to use for data generation."	

get_parameters

This command returns the names of all parameters that have been previously defined by add_parameter as a space separated list.

get_parameters	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	get_parameters
Returns	list of strings
Arguments	None
Example	set parameter_summary [get_parameters]

get_parameter_property

This command returns a single parameter property.

get_parameter_property		
Callback availability	Main, validation, elaboration, generation, and editor	
Usage	get_parameter_property *<parameterName> <propertyName>*	
Returns	string or boolean, depending on property refer to Table 7–4	
Arguments	parameterName	The name of the parameter whose property value is being retrieved
	propertyName	One of the properties listed in Table 7–4
Example	get_parameter_property parameter1 GROUP	

set_parameter_property

This command sets a single parameter property.

set_parameter_property		
Callback availability	Main and validation	
Usage	set_parameter_property *<parameterName> <propertyName> <value>*	
Returns	None	
Arguments	parameterName	Specifies the parameter that is being set
	propertyName	Specifies the property of parameterName that is being set, refer to Table 7–4 for a list of properties
	value	Provides the values
Example	set_parameter_property BAUD_RATE ALLOWED_RANGES {9600 19200 38400}	

get_parameter_value

This command returns the current value of a parameter defined previously with the add_parameter command.

get_parameter_value		
Callback availability	Validation, elaboration *(1)*, generation, and editor	
Usage	get_parameter_value *<parameterName>*	
Returns	Depends on type of the parameter	
Arguments	parameterName	Specifies the parameter that is being retrieved
Example	set fifo_width [get_parameter_value fifo_width]	

Note:

(1) If AFFECTS_ELABORATION=false for a given parameter, get_parameter_value is not available for that parameter from the elaboration callback.

set_parameter_value

This command sets a parameter value. Typically, the value of derived parameters is set during the validation callback based on the value of other parameters.

set_parameter_value		
Callback availability	Validation and editor	
Usage	`set_parameter_value <parameterName> <value>`	
Returns	None	
Arguments	`parameterName`	Specifies the parameter that is being set
	`value`	Specifies the value of `parameterName`
Example	`set_parameter_value BAUD_RATE 19200`	

decode_address_map

This is a utility function to convert an XML–formatted address map into a list of Tcl lists. Each inner list is in the correct format for conversion to an array. Using this command to decode the XML representing an Avalon-MM master's address map is easier and ensures that your code will work with future versions of the XML address map.

decode_address_map		
Callback availability	Validation, elaboration, and generation	
Usage	`decode_address_map <address_map_XML_string>`	
Returns	List of Tcl lists, each one suitable for passing to array set	
Arguments	`address_map_XML_string`	An XML string describing the address map of an Avalon-MM master (refer to the `ADDRESS_MAP` type in the "Declaring Parameters" on page 7–3 for more information.

Example	`set address_map_xml [get_parameter_value my_map_param]`
	`set address_map_dec [decode_address_map $address_map_xml]`
	
	`foreach i $address_map_dec {`
	`    array set info $i`
	`    send_message info "Connected to slave $info(name)"`
	`}`	

add_display_item

You can use this command to specify the following two aspects of component display:

- You can create logical groups for a component's parameters. For example, you might want to create separate groups for the component's timing, size, and simulation parameters. A component displays the groups and parameters in the order that you specify them in the **_hw.tcl** file.
- You can specify an image to provide a pictorial representation of a parameter or parameter group.

You create a display group by adding display items to it. You do not need to explicitly create groups before using them.

add_display_item		
Callback availability	Main	
Usage	add_display_item *<groupName> <id> <type> [<additionalInfo>]*	
Returns	None	
Arguments	groupName	Specifies the group to which a display item belongs.
	id	Specifies the parameter or icon to be displayed in a group. Each display item associated with a component must have a different ID.
	type	Specifies the category of the display item. There are currently 2 types: parameter and icon.
	additionalInfo	Provides extra information required for some display items. For icons, it provides the filename of the icon. You can use GIF, JPEG, and PNG file formats.
Examples	add_display_item timing read_latency parameter add_display_item sound speaker icon speaker.jpg	

Interfaces and Ports

You can use the interface and port commands to define interfaces and ports and retrieve their properties.

add_interface

This command adds an interface to your module. As the component author, you choose the name of the interface. By default, interfaces are enabled. You can set the interface property ENABLED to false, to disable a component interface. If an interface is disabled, it is hidden and its ports are automatically terminated to their default values. Signals that you designate as active low by appending a *_n* are terminated to 1. All other signals are terminated to 0.

The properties available for each interface type are different for every interface type. Refer to the *Avalon Interface Specifications*.

add_interface			
Callback availability	Main program and elaboration #		
Usage	add_interface *<interfaceName>* *<interfaceType>* *<direction>* [*<associatedClock>*] *(1)*		
Returns	None		
Arguments	interfaceName	A name that you choose to identify an interface.	
	interfaceType and direction	There are 7 interfaceTypes. The following directions are possible for these interfaceTypes:	
		Interface Type	Direction
		avalon	master, slave *(2)*
		avalon_tristate	slave
		avalon_streaming	source, sink
		interrupt	sender, receiver
		conduit	start
		clock	source, sink
		nios_custom_instruction	slave
	associatedClock	This defines the clock associated with the interface. It is required for all interfaces except clock interfaces.	
Example	add_interface mm_slave avalon slave clock0		

Notes:

(1) For interfaces that are not associated with clocks, such as clock interfaces themselves, the associatedClock is omitted. Another option is to specify the associatedClock argument as asynchronous.

(2) The terms *master*, *source* and *start* are interchangeable. The terms *slave*, *sink* and *end* are interchangeable.

get_interfaces

This command returns the names of all interfaces that have been previously defined by add_interface as space separated list.

get_interfaces	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	get_interfaces
Returns	list of strings
Arguments	None
Example	set all_interfaces [get_interfaces]

get_interface_properties

This command returns the names of all the available interface properties for the specified interface as a space separated list.

get_interface_properties		
Callback availability	Main program and elaboration	
Usage	`get_interface_properties <interfaceName>`	
Returns	list of strings	
Arguments	interfaceName	The name of an interface that you defined
Example	`get_interface_properties mm_slave`	

The properties available for each interface type are different for every interface type. Refer to the *Avalon Interface Specifications*.

get_interface_property

This command returns the value of a single interface property from the specified interface.

get_interface_property		
Callback availability	Main program and elaboration	
Usage	`get_interface_property <interfaceName> <propertyName>`	
Returns	Depends upon the type of the property being returned	
Arguments	`interfaceName`	The name of an interface from which you want to retrieve information
	`propertyName`	The name of the property whose value you want to retrieve
Example	`get_interface_property mm_slave readWaitTime`	

set_interface_property

This command sets a single interface property for an interface.

set_interface_property		
Callback availability	Main, validation, elaboration, generation, and editor	
Usage	`set_interface_property <interfaceName> <propertyName> <value>`	
Returns	Depends upon the type of the property being returned	
Arguments	`interfaceName`	The name of an interface that includes this property
	`propertyName`	The name of the property whose value you want to set
	`value`	The value to set for the specified property
Example	`set_interface_property mm_slave linewrapBursts false`	

add_interface_port

This command adds a port to an interface on your module. As the component author, you determine the name of the port. The port roles that you can set depend on the interface. The port direction and width may be omitted if either:

- If you have defined the direction and width of the port in the top-level HDL file
- If you define the direction and width by setting port properties in the elaboration callback

For ports added in the main program you can pass in a width of -1 to indicate explicitly that the port width will be defined later.

add_interface_port		
Callback availability	Main program and elaboration	
Usage	add_interface_port *<interfaceName> <portName> <portRole> [<direction> <width>]*	
Returns	None	
Arguments	interfaceName	The name of the interface to which the port belongs.
	portName	The name of the port that you, the component author, have chosen.
	portRole	The role of this port within the interfaces. Port roles are referred to as signal types in the *Avalon Specification*. Refer to the *Avalon Interface Specifications* for the signal types available for each interface type.
	direction	The direction can be input, output, bidir, and for VHDL, buffer.
	width	The width of the port in bits.
Example	add_interface_port mm_slave s0_rdata readdata output 32	

get_interface_ports

This command returns the names of all of the ports that have been added to a given interface. If the interface name is omitted, all ports for all interfaces are returned.

get_interface_ports		
Callback availability	Main, validation, elaboration, generation, and editor	
Usage	get_interface_ports *[<interfaceName>]*	
Returns	list of strings	
Arguments	interfaceName	The name of the interface whose ports you want to list. (Optional)
Example	get_interface_ports mm_slave	

get_port_properties

This command returns a list of all available port properties.

<table>
<tr><th colspan="5">get_port_properties</th></tr>
<tr><td>Callback availability</td><td colspan="4">Main, validation, elaboration, generation, and editor</td></tr>
<tr><td>Usage</td><td colspan="4"><code>get_port_properties <portName></code></td></tr>
<tr><td>Returns</td><td colspan="4">list of strings</td></tr>
<tr><td rowspan="6">Arguments</td><td><code>portName</code></td><td colspan="3">The name of the port whose properties are required. The following 4 port properties are supported:</td></tr>
<tr><td>Property</td><td>Type</td><td colspan="2">Description</td></tr>
<tr><td><code>DIRECTION</code></td><td><code>INPUT,OUTPUT</code>
<code>BIDIR,BUFFER</code></td><td colspan="2">The direction of the port from the component’s perspective.</td></tr>
<tr><td><code>WIDTH</code></td><td><code>Integer</code></td><td colspan="2">The width of the port in bits.</td></tr>
<tr><td><code>TERMINATION</code></td><td><code>Boolean</code></td><td colspan="2">When true, instead of connecting the port to the SOPC Builder system, it is left unconnected (<code>OUTPUT</code>, <code>BIDIR</code>, and <code>BUFFER</code>) or set to a fixed value (<code>INPUT</code>). Has no effect for components that implement a generation callback instead of using the default wrapper generation.</td></tr>
<tr><td><code>TERMINATION_WIDE</code></td><td>up to 63 bits</td><td colspan="2">The constant value to drive an input port.</td></tr>
<tr><td>Example</td><td colspan="4"><code>get_port_properties mm_slave</code></td></tr>
</table>

get_port_property

This command returns the value of single port property for the specified port.

<table>
<tr><th colspan="3">get_port_property<</th></tr>
<tr><td>Callback availability</td><td colspan="2">Main, validation, elaboration, generation, and editor</td></tr>
<tr><td>Usage</td><td colspan="2"><code>get_port_property <portName> <propertyName></code></td></tr>
<tr><td>Returns</td><td colspan="2">Depends on the type of the property</td></tr>
<tr><td rowspan="2">Arguments</td><td><code>portName</code></td><td>The name of the port</td></tr>
<tr><td><code>propertyName</code></td><td>One of the 4 supported properties:
■ <code>DIRECTION</code>
■ <code>WIDTH</code>
■ <code>TERMINATION</code>
■ <code>TERMINATION_VALUE</code></td></tr>
<tr><td>Example</td><td colspan="2"><code>get_port_property rdata WIDTH</code></td></tr>
</table>

set_port_property

This command sets a single port property.

set_port_property		
Callback availability	Main program, elaboration, and custom generation	
Usage	`set_port_property <portName> <propertyName> [<value>]`	
Returns	None	
Arguments	`portName`	The name of the port
	`propertyName`	■ One of the 4 supported properties: ■ `DIRECTION` ■ `WIDTH` ■ `TERMINATION` ■ `TERMINATION_VALUE`
	`value`	The value to set
Example	`set_port_property rdata WIDTH 32`	

get_interface_assignment

This command returns the value of the specified name for the specified interface.

get_interface_assignment		
Callback availability	Main and validation	
Usage	`get_interface_assignment <interfaceName> <name>`	
Returns	String	
Arguments	`interfaceName`	The name of the Avalon interface whose assignment is being retrieved
	`name`	The assignment whose value is being retrieved.
Example	`get_interface_assignment s1 embeddedsw.configuration.isFlash 1`	

set_interface_assignment

This command sets the value of the specified assignment for the specified interface.

set_interface_assignment		
Callback availability	Main and validation	
Usage	set_interface_assignment *<interfaceName> <name> [<value>]*	
Returns	None	
Arguments	interfaceName	The name of the Avalon interface whose assignment is being set
	name	The assignment whose value is being set.
	value	The value to assign
Example	set_interface_assignment s1 embeddedsw.configuration.isFlash 1	

Generation

This section covers the commands that set and get generation properties.

get_generation_properties

This command returns the names of all the available generation properties as a space separated list. These properties cannot be changed by the module.

get_generation_properties			
Callback availability	Main, validation, elaboration, generation, and editor		
Usage	get_generation_properties		
Returns	list of strings. The following generation properties are supported:		
	Property	Type	Description
	HDL_LANGUAGE	ENUM	The HDL language to generate. Is either verilog or vhdl (lowercase). If the module cannot generate the specified language, generating in the other language is acceptable.
	OUTPUT_DIRECTORY	File	The location in which files must be generated. The filename components in the directory name are separated with forward slashes.
	OUTPUT_NAME	String	The top-level file name and entity to be generated. If the OUTPUT_NAME is module_0 and the HDL_LANGUAGE is verilog, the file **module_0.v** must be generated and must contain the module, module_0.
Arguments	None		
Example	get_generation_properties		

get_generation_property

This command returns the value of a single generation property.

get_generation_property		
Callback availability	Generation	
Usage	`get_generation_property <propertyName>`	
Returns	`String` or `boolean`, depending on the type	
Arguments	`propertyName`	One of the 3 generation properties: ■ `HDL_LANGUAGE` ■ `OUTPUT_DIRECTORY` ■ `OUTPUT_NAME`
Example	`get_generation_property OUTPUT_DIRECTORY`	

get_project_property

This command returns the value of a single project property.

get_project_property			
Availability	Validation, elaboration, generation, and editor		
Usage	`get_project_property <propertyName>`		
Returns	`String` or `boolean`, depending on the property		
Argument	`propertyName` The following properties are supported:		
	Property	Type	Description
	`QUARTUS_ROOTDIR`	`DIRECTORY`	Value of the `$QUARTUS_ROOTDIR` env variable
	`QUARTUS_PROJECT_DIRECTORY`	`DIRECTORY`	Path to the current project directory
	`QUARTUS_PROJECT_NAME`	`String`	Name of the current Quartus II project
	`DEVICE_FAMILY_NAME`	`Enum`	One of the following current device families: STRATIX, STRATIXII, STRATIXIIGX, ARRIAGX, STRATIXGX, STRATIXIII, STRATIXIV, CYCLONE, CYCLONEII, CYCLONEIII, HARDCOPY, HARDCOPYII,HARDCOPYIII, MAXII, APEX20KE, APEX20KC, APEXII, ACEX1K
	`DEVICE_FAMILY_FEATURES`	`Enum`	The device family supports the following features: M512_MEMORY, M4K_MEMORY, M9K_MEMORY, M144K_MEMORY, MRAM_MEMORY, MLAB_MEMORY, ESB, EPCS, DSP, EMUL, HARDCOPY, LVDS_IO, ADDRESS_STALL, TRANSCEIVER_3G_BLOCK TRANSCEIVER_6G_BLOCK,DSP_SHIFTER_BLOCK
Example	`get_project_property DEVICE_FAMILY_NAME`		

Referenced Document

This chapter references the following document:

- *Avalon Interface Specifications*

Document Revision History

Table 7–6 shows the revision history for this chapter.

Table 7–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009, v9.0.0	■ Added `add_display_item` commands. ■ Added `DISPLAY_HINT`, `IS_HDL_PARAMETER`, `DERIVED`, and `SYSTEM_INFO` parameters to Table 7–4 on page 7–20. Described `SYSTEM_INFO` parameter in detail. ■ Added ENABLED interface property to enable or disable an interface. ■ The `AFFECTS_PORT_WIDTHS` parameter has been renamed `AFFECTS_ELABORATION` to better reflect its function. ■ Added note saying that the `add_file` command will be restricted to the main and generation callbacks starting in version 9.1 of the Quartus II software. ■ Explained that before the elaboration phase, parameters may have values of 0 or -1 that are determined during HDL analysis.	Added several new commands to increase functionality, clarified a few others, and corrected typographic errors.\
November 2008, v8.1	■ Added `get_module_ports`, `get_interface_assignment`, `set_interface_assignment`, `get_module_assignment`, and `set_module_assignment` commands ■ Corrected availability to include more callbacks for several commands ■ Added two additional types for `add_parameter` command: `natural` and `positive` ■ Added brackets for some optional parameters ■ Changed `add_file` command for simulation and synthesis in Example 7–10 to write to `$outdir` ■ `get_project_property` is available in validation callback ■ Changed page size to 8.5 x 11 inches	Added 5 new commands and corrected commands that did not define optional arguments or omitted some callback availability.
June 2008, v8.0.1	■ Reformatted command information in tables.	—
May 2008, v 8.0.0	■ Added new Editing **_hw.tcl** commands and ■ debug commands sections. ■ Changed chapter title from *Building a Component Interface with Tcl Scripting Commands* to *Component Interface Tcl Reference*.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

8. Archiving SOPC Builder Projects

QII54017-9.0.0

Introduction

This chapter identifies the files you must include when archiving an SOPC Builder project. With this information, you can archive the SOPC Builder system. You may want to archive your SOPC Builder system for one of the following reasons:

- To place an SOPC Builder design under source control
- To create a backup
- To bundle a design for transfer to another location

To use this information, you must decide what source control or archiving tool to use, and you must know how to use it. This chapter describes how to find and identify the files that you must include in an archived SOPC Builder design. Refer to "Required Files" on page 8–2.

Limitations

This chapter provides information about archiving SOPC Builder systems, including Nios® II software applications, if any. If your SOPC Builder system does not contain a Nios II processor, you can disregard information about archiving Nios II software applications.

This chapter does not cover archiving SOPC Builder components, for two reasons:

- SOPC Builder components can be recovered, if necessary, from the original Quartus® II and Nios II installations.
- If your SOPC Builder system was developed with an earlier version of the Quartus II software and Nios II Embedded Design Suite (EDS), when you restore it for use with the current version, you normally use the current, installed components.

If your SOPC Builder system was developed with an earlier version of the Quartus II Complete Design Suite and you restore it for use with the current version, the regenerated system is functionally identical to the original system. However, there might be differences in details such as timing performance, component implementation, or HAL implementation.

For details of version changes, refer to the *Quartus II Reference Documentation*.

To ensure that you can regenerate your exact original design, maintain a record of the tool and IP version(s) originally used to develop the design. Retain the original installation files or media in a safe place.

The archival process addressed by this chapter is different than Quartus II project archiving. A Quartus II project archive contains the complete Quartus II project, including the SOPC Builder module. The Quartus II software adds all HDL files to the archive, including HDL files generated by SOPC Builder, although these files are not strictly necessary, if you regenerate the design files afterwards. A Quartus II project archive also archives the Quartus II IP (**.qip**) file.

This chapter is only concerned with archiving the SOPC Builder system, without the generated HDL files.

For more details about archiving Quartus II projects, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook.*

Required Files

This section describes the files required to archive an SOPC Builder system and its associated Nios II software projects (if any). This is the minimum set of files needed to completely recompile an archived system, both the SRAM Object File (**.sof**) and the executable software (**.elf**).

If you have Nios II software projects, archive them together with the SOPC Builder system on which they are based. For more details about archiving Nios II designs, refer to the *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*.

The files listed in Table 8–1 are located in the Quartus II project directory.

Table 8–1. Files Required for an SOPC Builder System

File Description	File Name	Write Permission Required? *(1)*
SOPC Builder system description	*<sopc_builder_system>*.**sopc**	Yes
SOPC Builder classic system description for generation *(1)*	*<sopc_builder_system>*.**ptf**	Yes
SOPC Builder report file	*<sopc_builder_system>*.**sopcinfo**	Yes
All non-generated HDL source files *(2)*	for example: **top_level_schematic.bdf**, **customlogic.v**	No
Quartus II project file	*<project_name>*.**qpf**	No
Quartus II settings file	*<project_name>*.**qsf**	Yes

Notes to Table 8–1:

(1) The *<sopc_builder_system>*.**ptf** file is only required if you intend to edit or view the system in a version of SOPC Builder prior to version 7.1 and must also be writable to generate a system.

(2) Include all HDL source files not generated by SOPC Builder, including HDL source files you create or copy from elsewhere. To identify a file generated by SOPC Builder, open the file and look for the following header: `Legal Notice: (C)<year> Altera Corporation. All rights reserved.`

Many source control tools mark local files `read-only` by default. In this case, you must override this behavior. You do not have to check the files out of source control unless you are modifying the SOPC Builder design or Nios II software project.

Referenced Documents

This chapter references the following documents:

- *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*
- *Using the Nios II Software Build Tools* chapter of the *Nios II Software Developer's Handbook*
- *Quartus II Reference Documentation*

Document Revision History

Table 8–2 shows the revision history for this chapter.

Table 8–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009, v9.0.0	No change from previous release.	—
November 2008, v8.1.0	Changed page size to 8.5” × 11”.	—
May 2008, v8.0.0	Renumbering from Chapter 7 to 8.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

Section II. Building Systems with SOPC Builder

This section uses example designs to show you how to build a system or component. Chapters in this section serve to answer the question, "How do I define systems in SOPC Builder." This chapter refers to design examples that you can download free from www.altera.com. Design file hyperlinks are located with individual chapters linked from the Altera website.

This section includes the following chapters:

- Chapter 9, SOPC Builder Memory Subsystem Development Walkthrough
- Chapter 10, SOPC Builder Component Development Walkthrough

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

9. SOPC Builder Memory Subsystem Development Walkthrough

QII54006-9.0.0

Introduction

Most systems generated with SOPC Builder require memory. For example, embedded processor systems require memory for software, while digital signal processing (DSP) systems require memory for data buffers. Many systems use multiple types of memories. For example, a processor-based DSP system can use off-chip SDRAM to store software, and on-chip RAM for fast access to data buffers. You can use SOPC Builder to integrate almost any type of memory into your system.

This chapter uses design examples to describe how to build a memory subsystem as part of a larger system created with SOPC Builder. This chapter focuses on the following kinds of memory most commonly used in SOPC Builder systems:

- "On-Chip RAM and ROM" on page 9–6
- "EPCS Serial Configuration Device" on page 9–9
- "SDR SDRAM" on page 9–11
- "DDR SDRAM" on page 9–14
- "DDR2 SDRAM" on page 9–14
- "Off-Chip SRAM and Flash Memory" on page 9–15

This chapter assumes that you are familiar with the following task and concepts:

- Creating FPGA designs and making pin assignments with the Quartus® II software. For details, refer to the *Introduction to the Quartus II Software* manual.
- Building simple systems with SOPC Builder. For details, refer to the *Introduction to SOPC Builder* chapter in volume 4 of the *Quartus II Handbook.*
- SOPC Builder components. For details, refer to the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*.
- Basic concepts of the Avalon® interfaces. You do not need extensive knowledge of the Avalon interfaces, such as transfer types or signal timing. However, to create your own custom memory subsystem with external memories, you need to understand the Avalon Memory-Mapped (Avalon-MM) interface. For details, refer to the *System Interconnect Fabric for Memory-Mapped Interfaces* chapter in volume 4 of the *Quartus II Handbook* and the *Avalon Interface Specifications*.

Refer to the *Memory System Design* chapter in the *Embedded Design Handbook* for additional information on the efficient use of memories in SOPC Builder systems.

Example Design

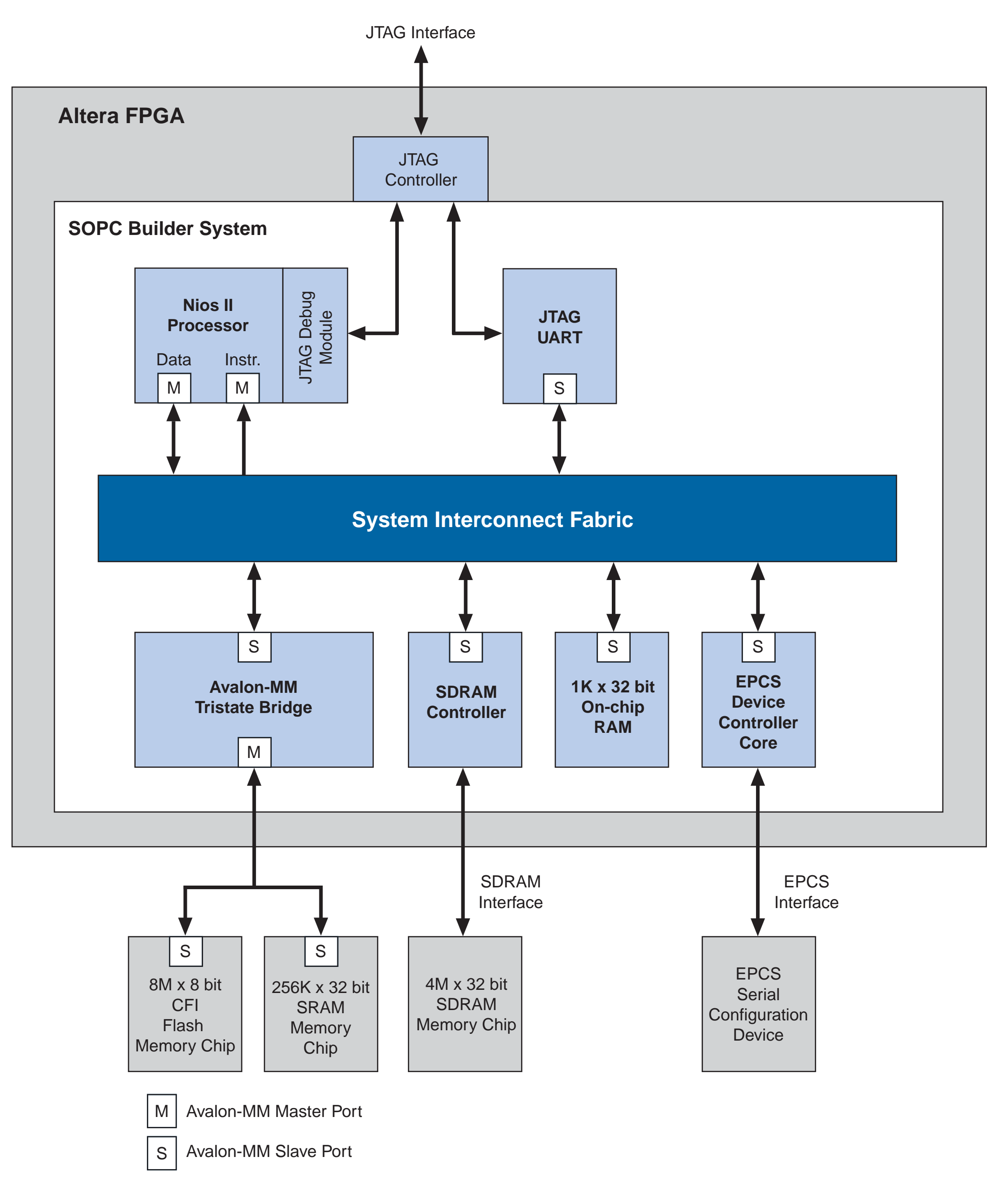
This chapter demonstrates the process for building a system that contains one of each type of memory as shown in Figure 9–1. Each section of the chapter builds on previous sections, culminating in a complete system.

By following the example design in this chapter, you learn how to create a complete customized memory subsystem for your system or design. The memory components in the example design are independent. For a custom system, you only need to instantiate the memories you need. You can also create multiple instantiations of the same type of memory, limited only by on-chip memory resources or FPGA pins to interface with off-chip memory devices.

Example Design Structure

Figure 9–1 shows a block diagram of the example system.

Figure 9–1. Example Design Block Diagram



In Figure 9–1, all blocks shown below the system interconnect fabric comprise the memory subsystem. For demonstration purposes, this system uses a Nios® II processor core to master the memory devices, and a JTAG UART core to communicate with the host PC. However, the memory subsystem could be connected to any master component, located either on-chip or off-chip.

Example Design Starting Point

The example design consists of the following elements:

- A Quartus II project named **quartus2_project**. A Block Design File (**.bdf)** named **toplevel_design**. **toplevel_design** is the top-level design file for **quartus2_project**. **toplevel_design** instantiates the SOPC Builder system, as well as other pins and modules required to complete the design.
- An SOPC Builder system named **sopc_memory_system**. **sopc_memory_system** is a subdesign of **toplevel_design**. **sopc_memory_system** instantiates the memory components and other SOPC Builder components required for a functioning SOPC Builder system.

This discussion assumes that the **quartus2_project** already exists, **sopc_memory_system** has been started in SOPC Builder, and the Nios II core and the JTAG UART core are already instantiated. This example design uses the default settings for the Nios II core and the JTAG UART core; these settings do not affect the rest of the memory subsystem.

Hardware and Software Requirements

To build a memory subsystem similar to the example design in this chapter, you need the following tools:

- Quartus II software version 5.0 or higher—Both Quartus II Web Edition and the fully licensed version support this design flow.
- Nios II Embedded Design Suite (EDS) version 5.0 or higher—Both the evaluation edition and the fully licensed version support this design flow. The Nios II EDS provides the SOPC Builder memory components described in this chapter. It also provides several complete example designs which demonstrate a variety of memory components instantiated in working systems.

The Quartus II Web Edition software and the Nios II EDS, Evaluation Edition are available free for download from the Altera® website. Visit www.altera.com/download. Also, for further reference, see the Design Examples.

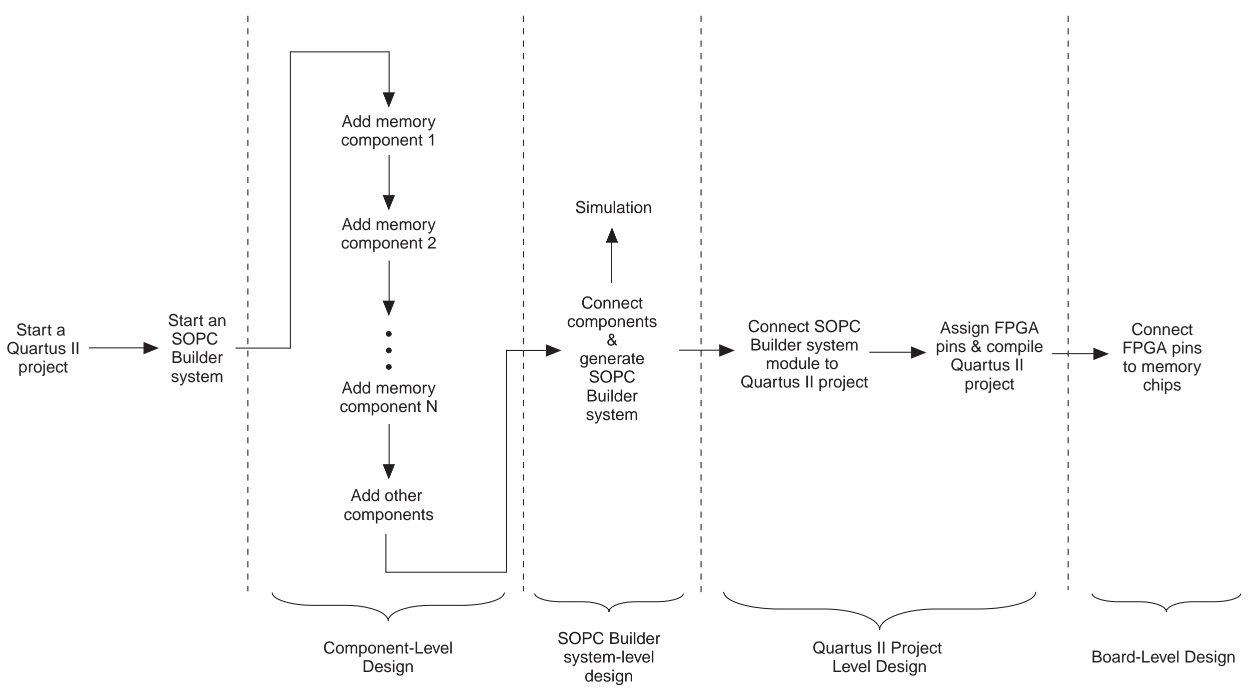
This chapter does not describe downloading and verifying a working system in hardware. Therefore, there are no hardware requirements for the completion of this chapter. However, the example memory subsystem has been tested in hardware.

Design Flow

This section describes the design flow for building memory subsystems with SOPC Builder, which is similar to other SOPC Builder designs. After starting a Quartus II project and an SOPC Builder system, there are five steps to completing the system, as shown in Figure 9–2:

1. Component-level design in SOPC Builder
2. SOPC Builder system-level design
3. Simulation
4. Quartus II project-level design
5. Board-level design

Figure 9–2. Design Flow



Component-Level Design in SOPC Builder

In this step, you specify which memory components to use and configure each component to meet the needs of the system. All memory components are available from the **Memory and Memory Controllers** category in the list of available components in SOPC Builder.

SOPC Builder System-Level Design

In this step, you connect components together and configure the SOPC Builder system as a whole. Like the process of adding non-memory SOPC Builder components, you use the **System Contents** tab to do the following:

- Rename the component instance (optional).

- Connect the memory component to masters in the system. Each memory component must be connected to at least one master.
- Assign a base address.
- Assign a clock domain. A memory component can operate on the same or different clock domain as the master(s) that access it.

Simulation

In this step, you verify the functionality of the SOPC Builder system. For systems with memories, this step depends on simulation models for each of the memory components, in addition to the system testbench generated by SOPC Builder. Refer to “Simulation Considerations” for more information.

Quartus II Project-Level Design

In this step, you integrate the SOPC Builder system with the rest of the Quartus II project, which includes connecting the SOPC Builder system to FPGA pins, connecting wiring the SOPC Builder system to other design blocks (such as other HDL modules) in the Quartus II project.

In the example design in this chapter, the SOPC Builder system comprises the entire FPGA design. There are no other design blocks in the Quartus II project.

Board-Level Design

In this step, you connect the physical FPGA pins to memory devices on the board. If the SOPC Builder system interfaces with off-chip memory devices, you must make board-level design choices.

Simulation Considerations

SOPC Builder can automatically generate a testbench for RTL simulation of the system using ModelSim®. This testbench instantiates the SOPC Builder system and can also instantiate memory models for external memory components. The testbench is plain text HDL, located at the bottom of the top-level SOPC Builder system HDL design file. To explore the contents of the auto-generated testbench, open the top-level HDL file and search on keyword `test_bench`.

Beginning in ModelSim SE 6.2, design optimization is on by default. Optimization may eliminate design nodes which are referenced in your wave display file. In this case, the you cannot display the waveforms. You can ignore this failure if you want to run an optimized simulation. However, if you want to see the simulation signals, you can disable the optimized compile by setting `VoptFlow = 0` in your **modelsim.ini** file. The **modelsim.ini** is stored in the top-level directory of the ModelSim installation.

Generic Memory Models

The memory components described in this chapter, except for the SRAM, provide generic simulation models. Therefore, it is very easy to simulate an SOPC Builder system with memory components immediately after generating the system.

The generic memory models store memory initialization files, such as Data (**.dat**) and Hexadecimal (**.hex**) files, in a directory named *<Quartus II project directory>*/*<SOPC Builder system name>***_sim**. When generating a new system, SOPC Builder creates empty initialization files. You can manually edit these files to provide custom memory initialization contents for simulation.

For designs that include a Nios II processor, you can create memory initialization files using the Nios II software build tools. For more information, refer to *Creating Memory Initialization Files* in the *Nios II Software Developer's Handbook – Studio Edition*.

Vendor-Specific Memory Models

You can also manually connect vendor-specific memory models to the SOPC Builder system. In this case, you must manually edit the testbench and connect the vendor memory model. You might also need to edit the vendor memory model slightly for time delays. The SOPC Builder testbench assumes zero delay.

On-Chip RAM and ROM

Altera FPGAs include on-chip memory blocks that can be used as RAM or ROM in SOPC Builder systems. On-chip memory has the following benefits for SOPC Builder systems:

- On-chip memory has fast access time, compared to off-chip memory.
- SOPC Builder automatically instantiates on-chip memory inside the SOPC Builder system, so you do not have to make any manual connections.
- Certain memory blocks can have initialized contents when the FPGA powers up. This feature is useful, for example, for storing data constants or processor boot code.
- On-chip memories support dual port accesses, allowing two master to access the same memory concurrently.

Component-Level Design for On-Chip Memory

In SOPC Builder you instantiate on-chip memory by clicking **On-chip Memory (RAM or ROM)** from the list of available components. The configuration wizard for the **On-chip Memory (RAM or ROM)** component has the following options: **Memory type**, **Size**, and **Read latency**.

Memory Type

The **Memory type** options define the structure of the on-chip memory:

- **RAM (writable)**—This setting creates a readable and writable memory.
- **ROM (read only)**—This setting creates a read-only memory.

- **Dual-port access—**This setting creates a memory component with two slaves, which allows two masters to access the memory simultaneously.

If two masters access the same address simultaneously in a dual-port memory undefined results will occur. (Concurrent accesses are only a problem for two writes. A read and write to the same location will read out the old data and store the new data.)

- **Block type—**This setting directs the Quartus II software to use a specific type of memory block when fitting the on-chip memory in the FPGA.

The MRAM blocks do not allow the contents to be initialized during power up. The M512s memory type does not support dual-port mode where both ports support both reads and writes.

Because of the constraints on some memory types, it is frequently best to use the **Auto** setting. **Auto** allows the Quartus II software to choose a type and the other settings direct the Quartus II software to select a particular type.

Size

The **Size** options define the size and width of the memory.

- **Data width—**This setting determines the data width of the memory. The available choices are **8**, **16**, **32**, **64**, **128**, **256**, **512**, or **1024** bits. Assign **Data width** to match the width of the master that accesses this memory the most frequently or has the most critical throughput requirements. For example, if you are connecting the on-chip memory to the data master of a Nios II processor, you should set the data width of the on-chip memory to 32 bits, the same as the data-width of the Nios II data master. Otherwise, the access latency could be longer than one cycle because the Avalon interconnect fabric performs width translation.
- **Total memory size—**This setting determines the total size of the on-chip memory block. The total memory size must be less than the available memory in the target FPGA.

Read Latency

On-chip memory components use synchronous, pipelined Avalon-MM slaves. Pipelined access improves f_{MAX} performance, but also adds latency cycles when reading the memory. The **Read latency** option allows you to specify either one or two cycles of read latency required to access data. If the **Dual-port access** setting is turned on, you can specify a different read latency for each slave. When you have dual-port memory in your system you can specify different clock frequencies for the ports. You specify this on the **System Contents** tab in SOPC Builder.

Non-Default Memory Initialization

For ROM memories, you can specify your own initialization file by selecting **Enable non-default initialization file**. This option allows the file you specify to be used to initialize the ROM in place of the default initialization file created by SOPC Builder.

Enable In-System Memory Content Editor Feature

Enables a JTAG interface used to read and write to the RAM while it is operating. You can use this interface to update or read the contents of the memory from your host PC.

For more information refer to *In-System Updating of Memory and Constants* in volume 3 of the *Quartus II Handbook*.

SOPC Builder System-Level Design for On-Chip Memory

There are few SOPC Builder system-level design considerations for on-chip memories. See “SOPC Builder System-Level Design” on page 9–4.

When generating a new system, SOPC Builder creates a blank initialization file in the Quartus II project directory for each on-chip memory that can power up with initialized contents. The name of this file is <*name of memory component*>**.hex**.

Simulation for On-Chip Memory

At system generation time, SOPC Builder generates a simulation model for the on-chip memory. This model is embedded inside the SOPC Builder system, and there are no user-configurable options for the simulation testbench.

You can provide memory initialization contents for simulation in the file <*Quartus II project directory*>/<*SOPC Builder system name*>**_sim/**<*Memory component name*>**.dat**.

Quartus II Project-Level Design for On-Chip Memory

The on-chip memory is embedded inside the SOPC Builder system, and there are no signals to connect to the Quartus II project.

To provide memory initialization contents, you must fill in the file <*name of memory component*>**.hex.** The Quartus II software recognizes this file during design compilation and incorporates the contents into the configuration files for the FPGA.

If your design includes a Nios II processor, you can create memory initialization files using the Nios II software build tools. For more information, refer to *Creating Memory Initialization Files* in the *Nios II Software Developer’s Handbook – Studio.* For the memory to be initialized, you then must compile the hardware in the Quartus II software for the SRAM Object File (**.sof**) to pick up the memory initialization files. All memory types with the exception of MRAMs support this feature.

Board-Level Design for On-Chip Memory

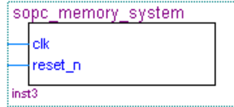
The on-chip memory is embedded inside the SOPC Builder system, and there is nothing to connect at the board level.

Example Design with On-Chip Memory

This section demonstrates adding a 4 KByte on-chip RAM to the example design. This memory uses a single slave interface with a read latency of one cycle.

For demonstration purposes, Figure 9–3 shows the result of generating the SOPC Builder system at this stage. (In a normal design flow, you generate the system only after adding all system components.)

Figure 9–3. SOPC Builder System with On-Chip Memory



Because the on-chip memory is contained entirely within the SOPC Builder system, **sopc_memory_system** has no I/O signals associated with **onchip_ram**. Therefore, you do not need to make any Quartus II project connections or assignments for the on-chip RAM, and there are no board-level considerations.

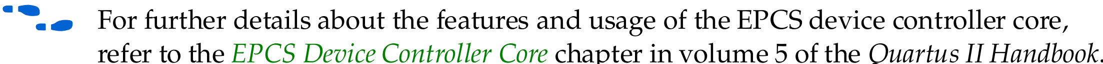
EPCS Serial Configuration Device

Many systems use an Altera EPCS serial configuration device to configure the FPGA. Altera provides the EPCS device controller core, which allows SOPC Builder systems to access the memory contents of the EPCS device.

This feature provides flexible design options:

- The FPGA design can reprogram its own configuration memory, providing a mechanism for remote upgrades.
- The FPGA design can use leftover space in the EPCS as nonvolatile storage.

Physically, the EPCS device is a serial flash memory device, which has slow access time. Altera provides software drivers to control the EPCS core for the Nios II processor only.



For further details about the features and usage of the EPCS device controller core, refer to the *EPCS Device Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

Component-Level Design for an EPCS Device

In SOPC Builder you instantiate an EPCS controller core by adding an **EPCS Serial Flash Controller** component. There are no settings for this component.

For details, refer to the *Nios II Flash Programmer User Guide*.

SOPC Builder System-Level Design for an EPCS Device

There are two SOPC Builder system-level design considerations for EPCS devices:

- Assign a base address.
- Set the IRQ connection to **NC** (no connect). The EPCS controller hardware is capable of generating an IRQ. However, the Nios II driver software does not use this IRQ, and therefore you can leave the IRQ signal disconnected.

There can only be one EPCS controller core per FPGA, and the instance of the core is always named `epcs_controller`.

If you want to store Nios II code in the EPCS memory, point the Nios II reset address at the EPCS controller. Inside the EPCS controller is a bootloader, which Nios II runs after it leaves reset, that copies the code from the EPCS flash into main memory.

Simulation for an EPCS Device

The EPCS controller core provides a limited simulation model:

- Functional simulation does not include the FPGA configuration process, and therefore the EPCS controller does not model the configuration features.
- The simulation model does not support read and write operations to the flash region of the EPCS device.
- A Nios II processor can boot from the EPCS device in simulation. However, the boot loader code is different during simulation. The EPCS controller boot loader code assumes that all other memory simulation models are initialized, and therefore the boot load process is unnecessary. During simulation, the boot loader simply forces the Nios II processor to jump to start, skipping the boot load process.

Verification in the hardware is the best way to test features related to the EPCS device.

Quartus II Project-Level Design for an EPCS Device

If you use a device from Cyclone III, Stratix III, or Stratix IV families, you must connect the EPCS pins manually.

For earlier device families, however, the Quartus II software automatically connects the EPCS controller core in the SOPC Builder system to the dedicated configuration pins on the FPGA. This connection is invisible to you. Therefore, there are no EPCS-related signals to connect in the Quartus II project.

Board-Level Design for an EPCS Device

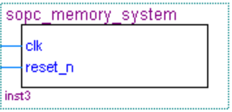
You must connect the EPCS device to the FPGA as described in the Altera *Configuration Handbook*. No other connections are necessary.

Example Design with an EPCS Device

This section demonstrates adding an EPCS device controller core to the example design.

For demonstration purposes only, Figure 9–4 shows the result of generating the SOPC Builder system at this stage.

Figure 9–4. SOPC Builder System with EPCS Device



Because the Quartus II software automatically connects the EPCS controller core to the FPGA pins, the SOPC Builder system has no I/O signals associated with **epcs_controller**. Therefore, you do not need to make any connections or assignments between the Quartus II project and the EPCS controller core.

This chapter does not cover the details of configuration using the EPCS device. For further information, refer to the Altera *Configuration Handbook*.

SDR SDRAM

Altera provides a free SDR SDRAM controller core, which allows you to use inexpensive SDRAM as bulk RAM in your FPGA designs. The SDR SDRAM controller core is necessary, because Avalon-MM signals cannot describe the complex interface on an SDRAM device. The SDR SDRAM controller acts as a bridge between the system interconnect fabric and the pins on an SDRAM device. The SDR SDRAM controller can operate in excess of 100 MHz.

SDR SDRAM is a single data rate SDR SDRAM. Synchronous design allows precise cycle control. With the use of system clock, I/O transactions are possible on every clock cycle. Operating over a range of frequencies, programmable latencies allow the same device to be useful for a variety of high bandwidth, high performance memory system applications.

For further details about the features and usage of the SDR SDRAM controller core, refer to the *SDR-SDRAM Controller Core with Avalon Interface* chapter in volume 5 of the *Quartus II Handbook*.

Component-Level Design for SDRAM

The choice of SDRAM device(s) and the configuration of the device(s) on the board heavily influence the component-level design for the SDRAM controller. Typically, the component-level design task involves parameterizing the SDRAM controller core to match the SDRAM device(s) on the board. You must specify the structure (address width, data width, number of devices, number of banks, and so on) and the timing specifications of the device(s) on the board.

For complete details about configuration options for the SDRAM controller core, refer to the *SDRAM Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

SOPC Builder System-Level Design for SDRAM

You can select the SDRAM controller in the SOPC Builder **System Contents** tab. Like the on-chip memory, there are few SOPC Builder system-level design considerations for SDRAM. Refer to “SOPC Builder System-Level Design” on page 9–4.

Simulation for SDRAM

At system generation time, SOPC Builder can generate a generic SDRAM simulation model and include the model in the system testbench. To use the generic SDRAM simulation model, you must turn on a setting in the SDRAM controller configuration wizard. You can provide memory initialization contents for simulation in the file *<Quartus II project directory>*/*<SOPC Builder system name>***_sim/***<Memory component name>***.dat.**

Alternatively, you can provide a specific vendor memory model for the SDRAM. In this case, you must manually wire up the vendor memory model in the system testbench.

For further details, refer to "Simulation Considerations" on page 9–5 and the *SDRAM Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

Quartus II Project-Level Design for SDRAM

SOPC Builder generates a SOPC Builder system with top-level I/O signals associated with the SDRAM controller. In the Quartus II project, you must connect these I/O signals to FPGA pins, which connect to the SDRAM device on the board. In addition, you might have to accommodate clock skew issues.

Connecting and Assigning the SDRAM-Related Pins

After generating the system with SOPC Builder, you can find the names and directions of the I/O signals in the top-level HDL file for the SOPC Builder system. The file has the name

<Quartus II project directory>/*<SOPC Builder system name>***.v** or *<Quartus II project directory>*/*<SOPC Builder system name>***.vhd**. You must connect these signals in the top-level Quartus II design file.

You must assign a pin location for each I/O signal in the top-level Quartus II design to match the target board. Depending on the performance requirements for the design, you might have to assign FPGA pins carefully to achieve the required performance.

Accommodating Clock Skew

As SDRAM frequency increases, so does the possibility that you must accommodate skew between the SDRAM clock and I/O signals. This issue affects all synchronous memory devices, including SDRAM. To accommodate clock skew, you can instantiate an ALTPLL megafunction in the top-level Quartus II design to create a phase-locked loop (PLL) clock output. You use a phase-shifted PLL output to drive the SDRAM clock and reduce clock-skew issues. The exact settings for the ALTPLL megafunction depend on your target hardware. You must experiment to tune the phase shift to match the board.

For details, refer to the *ALTPLL Megafunction User Guide*.

Board-Level Design for SDRAM

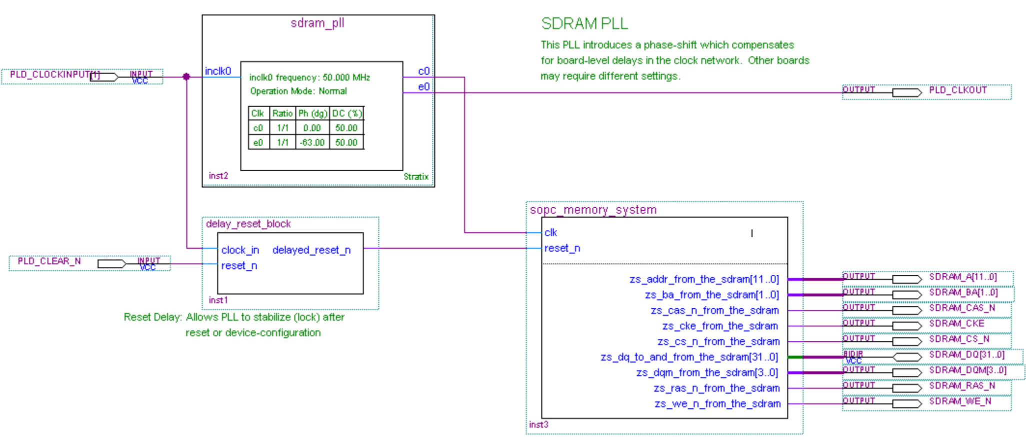
Memory requirements largely dictate the board-level configuration of the SDRAM device or devices. The SDRAM controller core can accommodate various configurations of SDRAM on the board, including multiple banks and multiple devices.

Example Design with SDR SDRAM

This section demonstrates adding a 16-Mbyte SDRAM device to the example design, using the SDRAM Controller configuration wizard. This SDRAM is a single device with 32-bit data.

For demonstration purposes, Figure 9–5 shows the result of generating the SOPC Builder system at this stage, and connecting it in **toplevel_design.bdf**.

Figure 9–5. toplevel_design.bdf with SDRAM



After generating the system, the top-level SOPC Builder system file **sopc_memory_system.v** contains the list of SDRAM-related I/O signals that must be connected to FPGA pins. Example 9–1 shows these pins.

Example 9–1. I/O Signals Connected to FPGA Pins

```
output  [ 11: 0] zs_addr_from_the_sdram;
output  [  1: 0] zs_ba_from_the_sdram;
output           zs_cas_n_from_the_sdram;
output           zs_cke_from_the_sdram;
output           zs_cs_n_from_the_sdram;
inout   [ 31: 0] zs_dq_to_and_from_the_sdram;
output  [  3: 0] zs_dqm_from_the_sdram;
output           zs_ras_n_from_the_sdram;
output           zs_we_n_from_the_sdram;
```

As shown in Figure 9–5, **toplevel_design.bdf** uses an instance of `sdram_pll` to phase shift the SDRAM clock by –63 degrees. (Degrees are relative to clock frequency. If you change the clock speed you must change the phase shift. You should parameterize the PLL with -3.5 ns, because the compensation is for the round-trip delays and clock to I/O delays.)

toplevel_design.bdf also uses a subdesign `delay_reset_block` to insert a delay on the `reset_n` signal for the SOPC Builder system. This delay is necessary to allow the PLL output to stabilize before the SOPC Builder system begins operating.

Figure 9–6 shows pin assignments in the Quartus II Assignment Editor for some of the SDRAM pins. The correct pin assignments depend on the target board.

Figure 9–6. Pin Assignments for SDRAM

	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reserved
188	SDRAM_A[0]	PIN_AE4	7	LVTTL	Column I/O		
189	SDRAM_A[10]	PIN_Y11	7	LVTTL	Column I/O		
190	SDRAM_A[11]	PIN_AB7	7	LVTTL	Column I/O		
191	SDRAM_A[1]	PIN_W12	7	LVTTL	Column I/O	PGM0	
192	SDRAM_A[2]	PIN_AC11	7	LVTTL	Column I/O	nRS	
193	SDRAM_A[3]	PIN_W10	7	LVTTL	Column I/O	RUnLU	
194	SDRAM_A[4]	PIN_AA11	7	LVTTL	Column I/O	PGM1	
195	SDRAM_A[5]	PIN_AC10	7	LVTTL	Column I/O	RDN7	
196	SDRAM_A[6]	PIN_AB11	7	LVTTL	Column I/O	RUP7	
197	SDRAM_A[7]	PIN_AC8	7	LVTTL	Column I/O	FCLK5	
198	SDRAM_A[8]	PIN_AB10	7	LVTTL	Column I/O	FCLK4	
199	SDRAM_A[9]	PIN_V11	7	LVTTL	Column I/O		
200	SDRAM_BA[0]	PIN_AG19	8	LVTTL	Column I/O	DQ6B4	
201	SDRAM_BA[1]	PIN_AF19	8	LVTTL	Column I/O	DQ6B5	
202	SDRAM_CAS_N	PIN_AD18	8	LVTTL	Column I/O	DQ6B2	
203	SDRAM_CKE	PIN_AE18	8	LVTTL	Column I/O	DQ6B1	
204	SDRAM_CS_N	PIN_AG18	8	LVTTL	Column I/O	DQ6B0	
205	SDRAM_DQM[0]	PIN_AE14	7	LVTTL	Column I/O	CLK6n	
206	SDRAM_DQM[1]	PIN_Y13	7	LVTTL	Column I/O	CLK7n	
207	SDRAM_DQM[2]	PIN_AE7	7	LVTTL	Column I/O	DQS1B	
208	SDRAM_DQM[3]	PIN_AG10	7	LVTTL	Column I/O	DQS3B	

DDR SDRAM

You can use double-data rate (DDR) SDRAM devices for a broad range of applications, such as embedded processor systems, image processing, storage, communications, and networking. In addition, the universal adoption of DDR SDRAM in PCs makes DDR SDRAM memory a solution for high-bandwidth applications. DDR SDRAM is a <*2n*> prefetch architecture where the internal data bus is twice the width of the external data bus and data transfers occur on both clock edges. It uses a strobe, `DQS`, which is associated with a group of data pins (`DQ`) for read and write operations. Both the `DQS` and `DQ` ports are bidirectional. Address ports are shared for write and read operations.

Refer to the DDR SDRAM literature on the Altera website for further details on the use of DDR SDRAM memory, including *AN 517: Using High-Performance DDR, DDR2, and DDR3 SDRAM With SOPC Builder*.

DDR2 SDRAM

Double-data rate DDR2 SDRAM is the second generation of double-data rate DDR SDRAM technology, with features such as lower power consumption, higher data bandwidth, enhanced signal quality, and on-die termination. DDR2 SDRAM brings higher memory performance to a broad range of applications, such as PCs, embedded processor systems, image processing, storage, communications, and networking. It is a <*4n*> pre-fetch architecture with two data transfers per clock cycle. The memory uses a strobe (`DQS`) associated with a group of data pins (`DQ`) for read and write operations. Both the `DQ` and `DQS` ports are bidirectional. Address ports are shared for write and read operations.

For more information refer to the *DDR and DDR2 SDRAM Controller Compiler User Guide*, the *DDR2 SDRAM High-Performance Controller User Guide*, and *AN 517: Using High-Performance DDR, DDR2, and DDR3 SDRAM With SOPC Builder*.

Off-Chip SRAM and Flash Memory

SOPC Builder systems can directly access many off-chip RAM and ROM devices, without a controller core to drive the off-chip memory. Avalon-MM signals can describe the interfaces on many standard memories, such as SRAM and flash memory. I/O signals on the SOPC Builder system can connect directly to the memory device.

While off-chip memory usually has slower access time than on-chip memory, off-chip memory provides the following benefits:

- Off-chip memory cost-per-bit is less expensive than on-chip memory resources.
- The size of off-chip memory is bounded only by the 32-bit Avalon-MM address space.
- Off-chip ROM, such as flash memory, can be used for bulk storage of nonvolatile data.
- Multiple off-chip RAM and ROM memories can share address and data pins to conserve FPGA I/O resources at the expense of throughput.

Adding off-chip memories to an SOPC Builder system also requires the **Avalon-MM Tristate Bridge** component.

Component-Level Design for SRAM and Flash Memory

There are several ways to instantiate an interface to an off-chip memory device:

- For common flash interface (CFI) flash memory devices, add the **Flash Memory (Common Flash Interface)** component in SOPC Builder.
- For Altera development boards, Altera provides SOPC Builder components that interface to the specific devices on each development board. For example, the Nios II EDS includes the components **Cypress CY7C1380C SSRAM** and **IDT71V416 SRAM**, which appear on Nios II development boards.

For further details about the features and usage of the SSRAM controller core, refer to the *Nios Development Board Cyclone II Edition Reference Manual* or *Nios Development Board Stratix II Edition*.

For further details about the features and usage of the SDRAM controller core, refer to the *Building Memory Subsystems Using SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*.

These components make it easy for you to create memory systems targeting Altera development boards. However, these components target only the specific memory device on the board; they do not work for different devices.

- For general memory devices, RAM or ROM, you can create a custom interface to the device with the SOPC Builder component editor. Using the component editor, you define the I/O pins on the memory device and the timing requirements of the pins.

In all cases, you must also instantiate the **Avalon-MM Tristate Bridge** component. Multiple off-chip memories can connect to a single tristate bridge, in order to share pins such as the off-chip address bus.

Avalon-MM Tristate Bridge

A tristate bridge connects off-chip devices to the system interconnect fabric. The tristate bridge creates I/O signals for the SOPC Builder system, which you must connect to FPGA pins in the top-level Quartus II project.

The tristate bridge creates address and data pins that can be shared by multiple off-chip devices. This feature lets you conserve FPGA pins when connecting the FPGA to multiple devices with mutually exclusive access.

You must use a tristate bridge in either of the following cases:

- The off-chip device has bidirectional data pins.
- Multiple off-chip devices share the address, data, or both address and data buses.

In SOPC Builder, you instantiate a tristate bridge by instantiating the **Avalon-MM Tristate Bridge** component. The Avalon-MM Tristate Bridge configuration wizard has a single option: To register incoming (to the FPGA) signals or not.

- **Registered**—This setting adds registers to all FPGA input pins associated with the tristate bridge (outputs from the memory device).
- **Not Registered**—This setting does not add registers between the memory device output pins and the system interconnect fabric.

The Avalon-MM tristate bridge automatically adds registers to output signals from the tristate bridge to off-chip devices.

Registering the input and output signals shortens the register-to-register delay from the memory device to the FPGA, resulting in higher system f_{MAX} performance. However, the registers add one additional cycle of latency for Avalon-MM masters accessing memory connected to the tristate bridge in each direction. The registers do not affect the timing (setup, hold, and wait) of the transfers from the perspective of the memory device.

For details about the Avalon-MM tristate interface, refer to the *Avalon Interface Specifications*.

Flash Memory

In SOPC Builder, you instantiate an interface to CFI flash memory by adding a **Flash Memory (Common Flash Interface)** component. If the flash memory is not CFI compliant, you must create a custom interface to the device with the SOPC Builder component editor.

The choice of flash devices and the configuration of the devices on the board help determine the component-level design for the flash memory configuration wizard. Typically, the component-level design task involves parameterizing the flash memory interface to match the devices on the board. Using the Flash Memory (Common Flash Interface) configuration wizard, you must specify the structure (address width and data width) and the timing specifications of the flash memory devices.

For details about features and usage, refer to the *Common Flash Interface Controller Core* chapter in volume 5 of the *Quartus II Handbook*.

For an example of instantiating the Flash Memory (Common Flash Interface) component in an SOPC Builder system, see "Example Design with SRAM and Flash Memory" on page 9–21.

SRAM

To instantiate an interface to off-chip SRAM:

1. Create a new component with the SOPC Builder component editor that defines the interface.
2. Instantiate the new interface component in the SOPC Builder system.

The choice of RAM devices and the configuration of the devices on the board determine how you create the interface component. The component-level design task involves entering parameters into the component editor to match the devices on the board.

For details about using the component editor, refer to the *Component Editor* chapter in volume 4 of the *Quartus II Handbook*.

SOPC Builder System-Level Design for SRAM and Flash Memory

In the SOPC Builder **System Contents** tab, the Avalon-MM tristate bridge has two ports:

- Avalon-MM slave—This port faces the on-chip logic in the SOPC Builder system. You connect this slave to on-chip masters in the system.
- Avalon-MM tristate master—This port faces the off-chip memory devices. You connect this master to the Avalon-MM tristate slaves on the interface components for off-chip memories.

You assign a clock to the Avalon-MM tristate bridge that determines the Avalon-MM clock cycle time for off-chip devices connected to the tristate bridge.

You must assign base addresses to each off-chip memory. The Avalon-MM tristate bridge does not have an address; it passes unmodified addresses from on-chip masters to off-chip slaves.

Simulation for SRAM and Flash Memory

The SOPC Builder output for simulation depends on the type of memory components in the system:

- **Flash Memory (Common Flash Interface)** component—This component provides a generic simulation model. You can provide memory initialization contents for simulation in the file *<Quartus II project directory>*/*<SOPC Builder system name>***_sim/***<Flash memory component name>***.dat**.
- Custom memory interface created with the component editor—In this case, you must manually connect the vendor simulation model to the system testbench. SOPC Builder does not automatically connect simulation models for custom memory components to the SOPC Builder system.
- Altera-provided interfaces to memory devices—Altera provides simulation models for these interface components. You can provide memory initialization contents for simulation in the file *<Quartus II project directory>*/*<SOPC Builder system name>***_sim/***<Memory component name>***.dat**. Alternately, you can provide a specific vendor simulation model for the memory. In this case, you must manually wire up the vendor memory model in the system testbench.

For further details, see "Simulation Considerations" on page 9–5.

Quartus II Project-Level Design for SRAM and Flash Memory

SOPC Builder generates an SOPC Builder system with top-level I/O signals associated with the tristate bridge and the memory interface components. In the Quartus II project, you must connect the I/O signals to FPGA pins, which connect to the memory devices on the board.

After generating the system with SOPC Builder, you can find the names and directions of the I/O signals in the top-level HDL file for the SOPC Builder system. The file has the name *<Quartus II project directory>*/*<SOPC Builder system name>***.v** or *<Quartus II project directory>*/*<SOPC Builder system name>***.vhd**. You must connect these signals in the top-level Quartus II design file.

You must assign a pin location for each I/O signal in the top-level Quartus II design to match the target board. Depending on the performance requirements for the design, you might have to assign FPGA pins carefully to achieve timing.

SOPC Builder inserts synthesis directives in the top-level SOPC Builder system HDL to assist the Quartus II fitter with signals that interface with off-chip devices. Example 9–2 illustrates a directive. Using `FAST_OUTPUT_REGISTER=ON` places the output register in the IO block, reducing the off-chip delay.

For more information about improving IO timing refer to the I/O Specifications section in *The Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* and the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook.*

Example 9–2. Synthesis Directive

```
reg [ 22: 0] tri_state_bridge_address /* synthesis
ALTERA_ATTRIBUTE =  "FAST_OUTPUT_REGISTER=ON" */;
```

Board-Level Design for SRAM and Flash Memory

Memory requirements determine the board-level configuration of the SRAM and flash memory device or devices. You can lay out memory devices in any configuration, as long as the resulting interface can be described with Avalon-MM signals.

Special consideration is required when connecting the Avalon-MM `address` signal to the address pins on the memory devices.

The SOPC Builder system presents the smallest number of address lines required to access the largest off-chip memory, which is usually less than 32 address bits. Not all memory devices connect to all address lines.

Aligning the Least-Significant Address Bits

The Avalon-MM tristate `address` signal always presents a byte address. Each address location in many memory devices contains more than one byte of data. In this case, the memory device must ignore one or more of the least-significant Avalon-MM `address` lines. For example, a 16-bit memory device must ignore Avalon-MM `address[0]` (which is a byte address), and connect Avalon-MM `address[1]` to the least-significant address line.

Table 9–1 shows the relationship between Avalon-MM `address` lines and off-chip address pins for all possible Avalon-MM data widths.

Table 9–1. Connecting the Least-Significant Avalon-MM Address Line (Part 1 of 2)

Avalon-MM Address Line	Address Line Connecting to Memory Device				
	8-bit Memory	16-bit Memory	32-bit Memory	64-bit Memory	128-bit Memory
`address[0]`	A0	No connect	No connect	No connect	No connect
`address[1]`	A1	A0	No connect	No connect	No connect
`address[2]`	A2	A1	A0	No connect	No connect

Table 9–1. Connecting the Least-Significant Avalon-MM Address Line (Part 2 of 2)

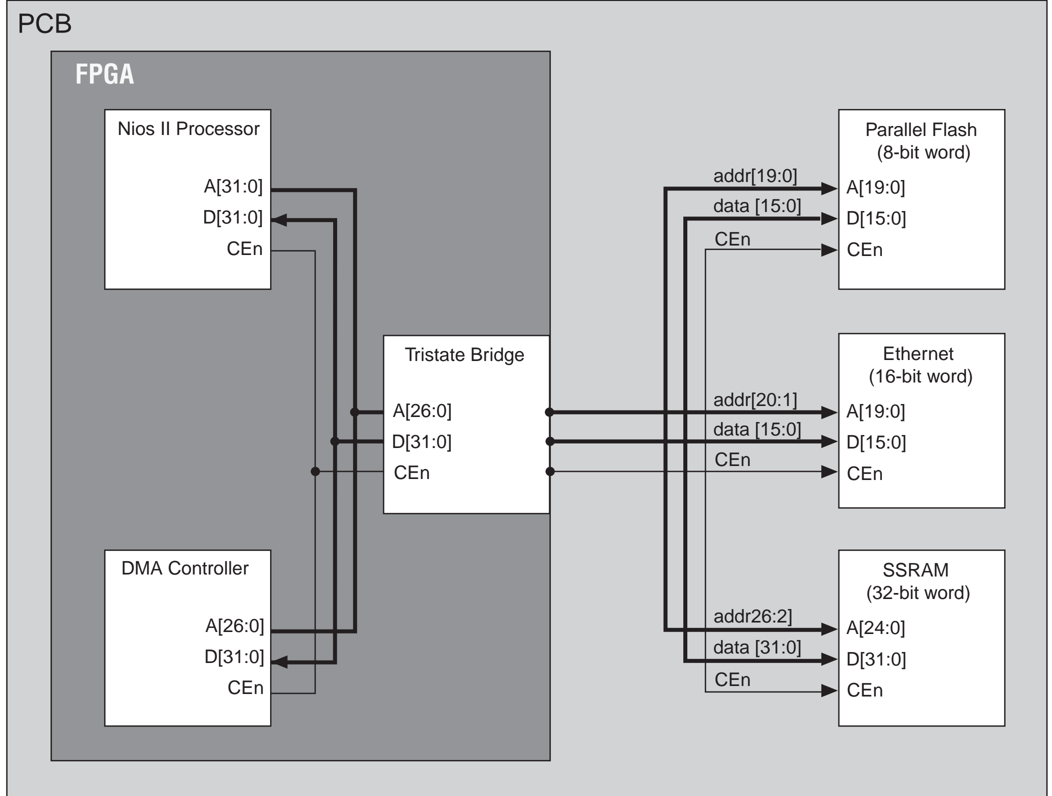
Avalon-MM Address Line	Address Line Connecting to Memory Device				
	8-bit Memory	**16-bit Memory**	**32-bit Memory**	**64-bit Memory**	**128-bit Memory**
`address[3]`	A3	A2	A1	A0	No connect
`address[4]`	A4	A3	A2	A1	A0
`address[5]`	A5	A4	A3	A2	A1
`address[6]`	A6	A5	A4	A3	A2
`address[7]`	A7	A6	A5	A4	A3
`address[8]`	A8	A7	A6	A5	A4
`address[9]`	A9	A8	A7	A6	A5
`address[10]`	A10	A9	A8	A7	A6
...	...	...	...	...	...

You must ensure that the address bits are properly assigned when mixed width components are connecting to the tristate bridge. Failing to ensure that the components are properly aligned may result in a board respin.

Aligning the Most-Significant Address Bits

The Avalon-MM `address` signal contains enough address lines for the largest memory connected to the tristate bridge. Smaller off-chip memories might not use all of the most-significant address lines as Figure 9–7 illustrates.

Figure 9–7. Connecting a Tristate Bridge to Components with Address Widths and Different Word Sizes



Example Design with SRAM and Flash Memory

This section demonstrates adding a 1-MByte SRAM and an 8-MByte flash memory to the example design. These memory devices connect to the system interconnect fabric through an Avalon-MM tristate bridge.

Adding the Avalon-MM Tristate Bridge

In the **Avalon-MM Tristate Bridge** configuration wizard, turn on the **Registered inputs and outputs** option to maximize system f_{MAX}, which increases the read latency by two for both the SRAM and flash memory.

Adding the Flash Memory Interface

The flash memory is 8M × 8-bit, which requires 23 address bits and 8 data bits. Table 9–2 gives the **Flash Memory (Common Flash Interface)** settings for the example design.

Table 9–2. Flash Memory Interface (CFI)

Parameter	Value
Attributes	
Presets	AMD29LV065D12R
Address Width (bits)	23
Data Width (bits)	8
Timing	
Setup	40
Wait	160
Hold	40
Units	ns

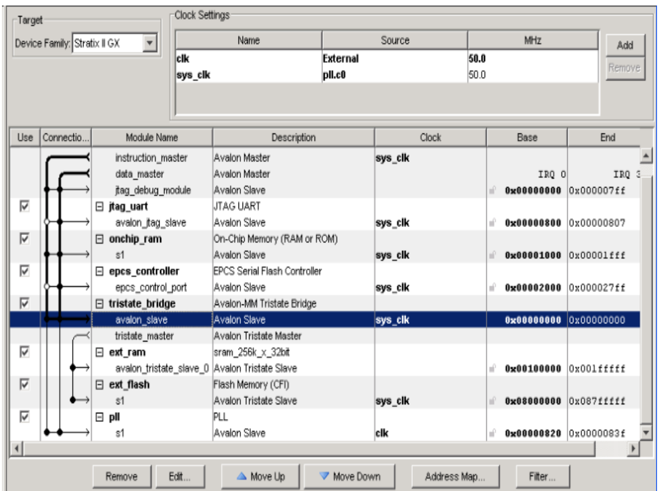
Adding the SRAM Interface

The SRAM device is 256K × 32-bit, which requires 18 word address bits and 32 data bits. The example design uses a custom memory interface created with the SOPC Builder component editor.

SOPC Builder System Contents Tab

Figure 9–8 shows the SOPC Builder system after adding the Tristate bridge and memory interface components, and configuring them appropriately on the **System Contents** tab. Figure 9–8 represents the complete example design in SOPC Builder.

Figure 9–8. SOPC Builder System with SRAM and Flash Memory



After generating the system, the top-level SOPC Builder system file **sopc_memory_system.v** contains the list of I/O signals for SRAM and flash memory that must be connected to FPGA pins, as shown in Example 9–3.

Example 9–3. I/O Signals for SRAM and Flash Memory

```
output          address_to_the_ext_flash[ 23..0];
output          address to_the_ext_ram[ 19..0];
output          be_n_to_the_ext_ram[ 3..0];
output          read_n_to_the_ext_flash;
output          read_n_to_the_ext_ram;
output          read_n_to_the_ext_ram;
output          select_n_to_the_ext_flash;
output          select_n_to_the_ext_ram;
bidirectional   tristate_bridge_data [ 31..0]
output          write_n_to_the_ext_flash;
output          write_n_to_the_ext_ram;
```

The Avalon-MM tristate bridge signals that can be shared are named after the instance of the tristate bridge component, such as `tri_state_bridge_data[31:0]`.

Connecting and Assigning Pins in the Quartus II Project

Figure 9–9 shows the result of generating the SOPC Builder system for the complete example design.

Figure 9–9. Top Level System with SRAM and Flash Memory

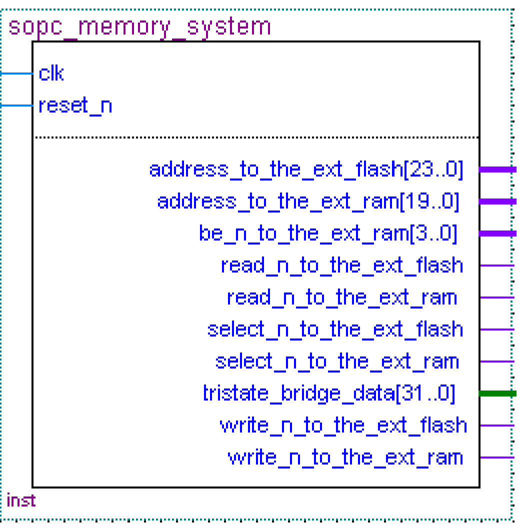


Figure 9–10 shows the pin assignments in the Quartus II Assignment Editor for some of the SRAM and flash memory pins. The correct pin assignments depend on the target board.

Figure 9–10. Pin Assignments for SRAM and Flash Memory

	To	Location	I/O Bank	I/O Standard	General Function	Special Function	Reser
243	SRAM_BE_N[0]	PIN_M18	3	LVTTL	Column I/O		
244	SRAM_BE_N[1]	PIN_F17	3	LVTTL	Column I/O		
245	SRAM_BE_N[2]	PIN_J18	3	LVTTL	Column I/O	RUP3	
246	SRAM_BE_N[3]	PIN_L17	3	LVTTL	Column I/O	CLK15n	
247	SRAM_CS_N	PIN_B24	3	LVTTL	Column I/O	DQ9T4	
248	SRAM_OE_N	PIN_B26	3	LVTTL	Column I/O	DQ9T7	
249	SRAM_WE_N	PIN_C24	3	LVTTL	Column I/O	DQS9T	

Connecting FPGA Pins to Devices on the Board

Table 9–3 shows the mapping between the Avalon-MM address lines and the address pins on the SRAM and flash memory devices.

Table 9–3. FPGA Connections to SRAM and Flash Memory (Part 1 of 2)

Avalon-MM Address Line	Flash Address (8M × 8-bit Data)	SRAM Address (256K × 32-bit data)
`tri_state_bridge_address[0]`	A0	No connect
`tri_state_bridge_address[1]`	A1	No connect
`tri_state_bridge_address[2]`	A2	A0
`tri_state_bridge_address[3]`	A3	A1
`tri_state_bridge_address[4]`	A4	A2
`tri_state_bridge_address[5]`	A5	A3
`tri_state_bridge_address[6]`	A6	A4
`tri_state_bridge_address[7]`	A7	A5

Table 9–3. FPGA Connections to SRAM and Flash Memory (Part 2 of 2)

Avalon-MM Address Line	Flash Address (8M × 8-bit Data)	SRAM Address (256K × 32-bit data)
tri_state_bridge_address[8]	A8	A6
tri_state_bridge_address[9]	A9	A7
tri_state_bridge_address[10]	A10	A8
tri_state_bridge_address[11]	A11	A9
tri_state_bridge_address[12]	A12	A10
tri_state_bridge_address[13]	A13	A11
tri_state_bridge_address[14]	A14	A12
tri_state_bridge_address[15]	A15	A13
tri_state_bridge_address[16]	A16	A16
tri_state_bridge_address[17]	A17	A15
tri_state_bridge_address[18]	A18	A16
tri_state_bridge_address[19]	A19	A17
tri_state_bridge_address[20]	A20	No connect
tri_state_bridge_address[21]	A21	No connect
tri_state_bridge_address[22]	A22	No connect

Referenced Documents

This chapter references the following documents:

- *Altera Configuration Handbook*
- *ALTPLL Megafunction User Guide*
- *AN 517: Using High-Performance DDR, DDR2, and DDR3 SDRAM With SOPC Builder*
- *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*
- *Avalon Interface Specifications*
- *Component Editor* chapter in volume 4 of the *Quartus II Handbook*
- *Common Flash Interface Controller Core* chapter in volume 5 of the *Quartus II Handbook*
- *Configuration Handbook*
- *DDR and DDR2 SDRAM Controller Compiler User Guide*
- *DDR2 SDRAM High-Performance Controller User Guide*
- *EPCS Device Controller Core* chapter in volume 5 of the *Quartus II Handbook*
- *In-System Updating of Memory and Constants* in volume 3 of the *Quartus II Handbook*
- *Introduction to the Quartus II Software manual*
- *Introduction to SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*
- *Memory System Design* chapter in the *Embedded Design Handbook*
- *Nios II Embedded Processor Design Examples*
- *Nios II Flash Programmer User Guide*
- *SDRAM Controller Core* chapter in volume 5 of the *Quartus II Handbook*
- *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*
- *System Interconnect Fabric for Memory-Mapped Interfaces* chapter in volume 4 of the *Quartus II Handbook*
- *The Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*

Document Revision History

Table 9–4 shows the revision history for this chapter.

Table 9–4. *Document Revision History*

Date and Document Version	Changes Made	Summary of Changes
March 2009, v9.0.0	Minor updates to clarify text.	—
November 2008, v8.1.1	■ Removed private comments	—
November 2008, v8.1.0	■ Added text explaining that starting in 6.2, ModelSim turns the VoptFlow option on by default which may optimize away nodes included in preset wave file. ■ Changed page size to 8.5 x 11 inches	—
May 2008, v8.0.0	■ Chapter renumbered from 8 to 9. ■ Added brief new sections referencing DDR-2 and PFLs. ■ Updated references to Avalon Interface Specifications. ■ Updated Figures 9-1, 9-14, 9-15, 9-16, and 9-19 with new art.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

10. SOPC Builder Component Development Walkthrough

QII54007-9.0.0

Introduction

This chapter describes the parts of a custom SOPC Builder component and guides you through the process of creating an example custom component, integrating it into a system, and testing it in hardware.

This chapter is divided into the following sections:

- "Component Development Flow" on page 10–2.
- "Design Example: Checksum Hardware Accelerator" on page 10–4. This design example shows you how to develop a component with both Avalon® Memory-Mapped (Avalon-MM) master and slaves.
- "Sharing Components" on page 10–7. This section shows you how to use components in other systems, or share them with other designers.
- ".sopcinfo Files" on page 10–7.

SOPC Builder Components and the Component Editor

An SOPC Builder component is usually composed of the following four types of files:

- HDL files—define the component's functionality as hardware.
- Hardware Component Description File (**_hw.tcl**) —describes the SOPC Builder related characteristics, such as interface behaviors. This file is created by the component editor.
- C-language files—define the component register map and driver software to allow programs to control the component.
- Software Component Description File (**_sw.tcl**) file—used by the software build tools to use and compile the component driver code.

The component editor guides you through the creation of your component. You can then instantiate the component in an SOPC Builder system and make connections in the same manner as other SOPC Builder components. You can also share your component with other designers.

For information about creating the **_sw.tcl** file, see the *Developing Device Drivers for the Hardware Abstraction Layer* chapter in the *Nios II Software Developer's Handbook*.

Prerequisites

This chapter assumes that you are familiar with the following:

- Building systems with SOPC Builder. For details, refer to the *Introduction to SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*.
- SOPC Builder components. For details, refer to the *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*.
- Basic concepts of the Avalon-MM interface.

Hardware and Software Requirements

To use the design example in this chapter, in addition to the current version of the Quartus II software and Nios II Embedded Design Suite, you must have the following:

- Design files for the example design—A hyperlink to the design files appears next to the chapter, *SOPC Builder Component Development Walkthrough*, on the SOPC Builder literature page.
- Nios development board and an Altera® USB-Blaster™ download cable—You can use either of the following Nios development boards:
  - Stratix® II Edition, RoHS compliant version
  - Cyclone® II Edition

If you do not have a development board, you can follow the hardware development steps. You cannot download the complete system without a working board, but you can simulate the system.

You can download the Quartus II Web Edition software and the Nios II EDS, Evaluation Edition for free from the Altera Download Center at www.altera.com.

Component Development Flow

This section provides an overview of the development process for SOPC Builder components.

Typical Design Steps

A typical development sequence for an SOPC Builder component includes the following items:

1. Specification and definition.
   a. Define the functionality of the component.
   b. Determine component interfaces, such as Avalon Memory-Mapped (Avalon-MM), Avalon Streaming (Avalon-ST), interrupt, or other interfaces.
   c. Determine the component clocking requirements; what interfaces are synchronous to what clock inputs.
   d. If you want a microprocessor to control the component, determine the interface to software, such as the register map.
2. Implement the component in VHDL or Verilog HDL.

3. Import the component into SOPC Builder.

   a. Use the component editor to create a **_hw.tcl** file that describes the component.

   b. Instantiate the component into an SOPC Builder system.

   When importing an HDL file using the component editor, any parameter definitions that are dependent upon other defined parameters cause an error. Example 10–1 illustrates the declaration of a `DEPTH` parameter which is legal Verilog HDL syntax in the Quartus II software, but causes an error in the component editor syntax checker.

Example 10–1. DEPTH Parameter

```
parameter WIDTH = 32;
parameter DEPTH = ((WIDTH == 32) ? 8 : 16);
```

   To avoid this error, use a `localparam` for the dependent parameter instead, as shown in Example 10–2.

Example 10–2. localparam Parameter

```
parameter WIDTH = 32;
localparam DEPTH = ((WIDTH == 32)?8:16);
```

4. Develop the software driver, which can occur in parallel with the hardware implementation. Create the component’s driver, including a C header file that defines the hardware-level register map for software.

   For further details, see the *Nios II Software Developer’s Handbook*.

5. Perform in-system testing, such as the following:

   a. Test register-level accesses to the component in hardware or simulation using a microprocessor, such as the Nios II processor.

   b. Performance benchmarking.

Hardware Design

As with any logic design process, the development of SOPC Builder component hardware begins after the specification phase. Creating the HDL design is often an iterative process, as you write and verify the HDL logic against the specification.

The architecture of a typical component consists of the following functional blocks:

- Task logic—Implements the component's fundamental function. The task logic is design dependent.
- Interface logic—Provides a standard way of providing data to or getting data from the components and of controlling the functioning of the components.

  For further details, refer to the *Avalon Interface Specifications*.

Figure 10–1 shows the top-level blocks of a checksum component, which includes both Avalon-MM master and slaves.

The work flow for developing SOPC Builder hardware, including how to decide upon and implement the register map, is described in the *Using the Nios II Software Build Tools* chapter in the *Nios II Software Developer's Handbook*. Also, guidelines for developing device drivers is described in the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Design Example: Checksum Hardware Accelerator

Altera has provided a checksum hardware accelerator design example to demonstrate the steps to create a component and instantiate it in a system. This design example is available for download from the Altera literature website. Included in the compressed download file is a **readme.pdf** that describes how to create and compile the hardware design, and describes how to use the checksum hardware accelerator in your design.

You can use the checksum algorithm in network applications where data integrity must be inspected by the receiving device. The checksum algorithm accumulates data with end-round-carry summation, which means that the carry bit from the accumulator is added to the least significant bit of the next input. After the data is accumulated, you can use the result to verify the data integrity of the data buffer. Because the checksum algorithm operates over a data buffer, you can implement it more efficiently with a pipelined read master. A pipelined read master continuously posts read transactions minimizing the effects of the memory read latency. The checksum accelerator can read data and calculate the checksum result every clock cycle, which you cannot do with a general purpose processor.

The checksum hardware accelerator requires information from a host processor such as the buffer base address, buffer length, and various control signals. As a result, the hardware accelerator exposes an Avalon-MM slave interface so that a host processor can control the read master operation. The host processor also accesses the checksum result from the slave interface. Each piece of information sent or read by the host processor is accessed separately in the register file implemented with the slave interface. For example, the status and control signals are implemented as separate registers because they contain information used for different purposes and have different access capabilities.

Hardware accelerators can operate in parallel with a host processor; consequently, adding an interrupt sender interface to the hardware accelerator increases system performance. While the accelerator is operating on a buffer, the host processor can perform other tasks such as preparing another buffer for transmission. The interrupt is asserted after the buffer checksum is calculated. The host processor can be interrupted by the hardware accelerator to notify it that a checksum result has been calculated. The host processor can then read the checksum value and clear the interrupt by writing to the status register via the accelerator slave interface.

Figure 10–1. Checksum Component with Avalon-MM Master and Slaves

Software Design

If you want a microprocessor to control your component, you must provide software files that define the software view of the component. At a minimum, you must define the register map for each Avalon-MM slave that is accessible to a processor.

Typically, the header file declares macros to read and write each register in the component, relative to a symbolic base address assigned to the component. Table 10–1 shows the register map of the checksum component for use by the Nios II processor.

Table 10–1. Avalon-MM Slave Port Register Map (Control)

Offset	Name	Rd/Wr/clr	Bits										
			31 10	9	8	7	6	5	4	3	2	1	0
0	Status	Rd/Wclr										Busy	Done
4	Read Address *(1)*	Rd/Wr	Read Address (32-bit word aligned)										
8	N/A	—	Reserved ()										
12	Length (Bytes)	Rd/Wr	Length in Bytes (must be a multiple of 4 for word aligned)										
16	N/A	—	Reserved ()										
20	N/A	—	Reserved ()										
24	Control	Rd/Wr			RC ON				I_E N	GO		Inv	Clr
28	Checksum Results	Rd	16-Bit Checksum Result (upper 16 bits are zeros)										

Note to Table 10–1:

(1) Wr=Writable; Rd=Readable; Wclr=Write 1 to clear

In the example checksum project, you can view an example of a software driver in the directory <*projectdir*>**/ip/checksum_accelerator**, which is the top level folder of the hardware and software for the custom checksum block.

Software drivers abstract hardware details of the component so that software can access the component at a high level. The driver functions provide the software an API to access the hardware. The software requirements vary according to the needs of the component. The most common types of routines initialize the hardware, read data, and write data.

When developing software drivers, you should review the software files provided for other ready-made components. The IP installer provides many components you can use as reference. You can also view the <*Nios II EDS install path*>/**components**/ directory for examples.

For details about writing drivers for the Nios II hardware abstraction layer (HAL), refer to the *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*.

Verifying the Component

You can verify the component in incremental stages, as you complete more of the design. You should first verify the hardware logic as a unit (which might consist of multiple smaller stages of verification) and later verify the component in a system.

System Console

The system console is an interactive Tcl console available from within SOPC Builder that provides you with read and write access to the debugging capabilities that are available in your FPGA logic. You can use the system console to control and query the state of the Nios II processor, issue Avalon transactions, board bring-up, and access either JTAG UARTs or system level debug (SLD) nodes.

 © March 2009 Altera Corporation

For further details, refer to the *System Console User Guide*.

System-Level Verification

After you package a **_hw.tcl** file with the component editor, you can instantiate the component in a system and verify the functionality of the overall SOPC Builder system.

SOPC Builder provides support for system-level verification for HDL simulators such as ModelSim®. SOPC Builder automatically produces a test bench for system-level verification.

You can include a Nios II processor in your system to enhance simulation capabilities during the verification phase. Even if your component has no relationship to the Nios II processor, the auto-generated ModelSim simulation environment provides an easy-to-use starting point.

Sharing Components

When you create a component, component editor saves the **_hw.tcl** file in the same directory as the top-level HDL file. Where appropriate, files referenced by the **_hw.tcl** file are specified relative to the **_hw.tcl** file itself, so the files can easily be moved and copied. To share a component, include it in your IP library.

For more information about including components in an IP library refer to *Finding Components in SOPC Builder* in *Chapter 4: SOPC Builder Components* in volume 4 of the *Quartus II Handbook*.

.sopcinfo Files

Every time SOPC Builder generates a system, a <*mysystem*>**.sopcinfo** file is also generated, which contains the following information:

- SOPC Builder project, including:
  - Name and tool version
  - HDL language
- Each module instantiated in the system, including:
  - Name and version
  - Where interface information was found on the disk, such as signal names and types, interface properties, and clock domain mapping
  - Parameter names and values
- Each connection, including:
  - Component and interface connections
  - Base address, Avalon-MM interfaces, IRQ number interfaces
  - Memory map as seen by each master in the system

The **.sopcinfo** file is a report file only, and cannot be edited with SOPC Builder.

Referenced Documents

This chapter references the following documents:

- *Developing Device Drivers for the Hardware Abstraction Layer* chapter of the *Nios II Software Developer's Handbook*
- *Introduction to SOPC Builder* chapter in volume 4 of the *Quartus II Handbook*
- *SOPC Builder Components* chapter in volume 4 of the *Quartus II Handbook*
- *System Console User Guide*
- *Using the Nios II Software Build Tools* chapter in the *Nios II Software Developer's Handbook*

Document Revision History

Table 10–2 shows the revision history for this chapter.

Table 10–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009, v9.0.0	Corrected direction of transform_data_available and transform_byte_lanes signals in Figure 10–1 on page 10–5.	One correction.
November 2008, v8.1.0	■ Added reference to new search path for IP chapter 4 of this volume. ■ Correction direction of signals in Figure 10–1. ■ Changed page size to 8.5 x 11 inches.	One correction and one change to reflect changes in underlying software.
May 2008, v8.0.0	■ Chapter renumbered from 9 to 10. ■ Removed discussion of the Checksum Design example, which will now be in a readme.pdf file and zipped with the rest of the design files. ■ Deleted references to Avalon Memory-Mapped and Streaming Interface Specifications and changed to Avalon Interface Specifications. ■ New Figure 9-1 and Table 9-1. ■ New section on **.sopcinfo** file.	Deleted example procedure.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

Section III. Interconnect Components

This section provides information on Avalon® Memory-Mapped (Avalon-MM) and Avalon Streaming (Avalon-ST) components that can be added to SOPC Builder systems. The components described in these chapters help you to create and optimize your SOPC Builder system. They are provided for free and can be used without a license in any design targeting an Altera device.

This section includes the following chapters:

- Chapter 11, Avalon Memory-Mapped Bridges
- Chapter 12, Avalon Streaming Interconnect Components

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

11. Avalon Memory-Mapped Bridges

QII54020-9.0.0

Introduction to Bridges

This chapter introduces Avalon® Memory-Mapped (Avalon-MM) bridges, and describes the Avalon-MM bridge components provided by Altera® for use in SOPC Builder systems.

You use bridges to control the topology of the generated SOPC Builder system. Bridges are not end-points for data, but rather affect the way data is transported between components. By inserting Avalon-MM bridges between masters and slaves, you control system topology, which in turn affects the interconnect that SOPC Builder generates. You can also use bridges to separate components in different clock domains and to implement clock domain crossing logic. Manual control of the interconnect can result in higher performance or lower logic utilization or both. Altera provides the following Avalon-MM bridges:

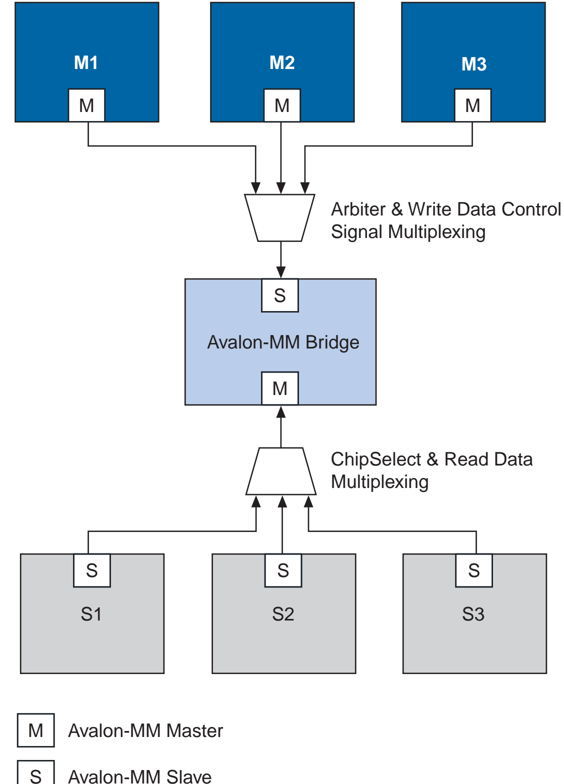
- "Avalon-MM Pipeline Bridge" on page 11–7
- "Clock Crossing Bridge" on page 11–10
- "Avalon-MM DDR Memory Half-Rate Bridge" on page 11–18

For additional information on using bridges to optimize and control the topology of SOPC Builder systems, refer to *Avalon Memory-Mapped Design Optimizations* in the *Embedded Design Handbook*.

Structure of a Bridge

A bridge has one Avalon-MM slave and one Avalon-MM master, as shown in Figure 11–1. In an SOPC Builder system, one or more masters connect to the bridge slave; in turn, the Avalon-MM bridge master connects to one or more slaves. In Figure 11–1, all three masters have logical connections to all three slaves, although physically each master only connects to the bridge.

Figure 11–1. Example of an Avalon-MM Bridge in an SOPC Builder System



Transfers initiated to the bridge's slave propagate to the master in the same order in which they are initiated on the slave.

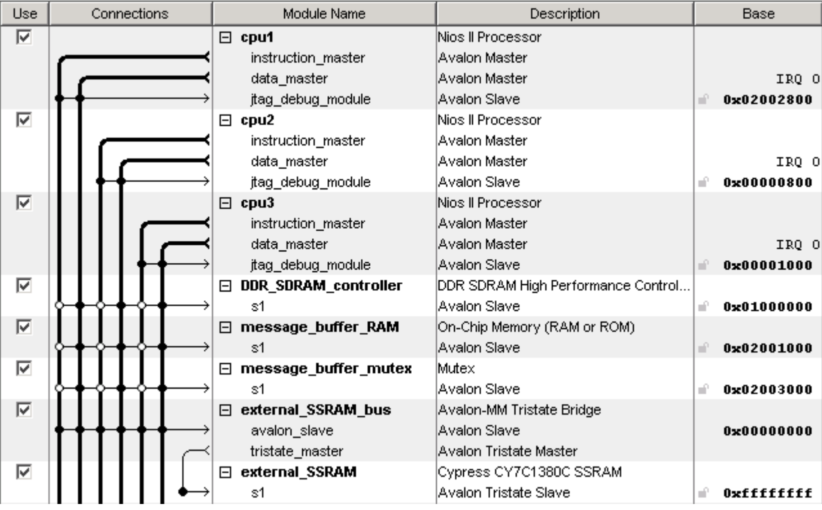
For details on the Avalon-MM interface, refer to the *Avalon Interface Specifications*.

Reasons for Using a Bridge

When you have no bridges between master-slave pairs, SOPC Builder generates a system interconnect fabric with maximum parallelism, such that all masters can drive transactions to all slaves concurrently, as long as each master accesses a different slave. For systems that do not require a large degree of concurrency, the default behavior might not provide optimal performance. With knowledge of the system and application, you can optimize the system interconnect fabric by inserting bridges to control the system topology.

Figure 11–2 and Figure 11–3 show an SOPC system without bridges. This system includes three CPUs, a DDR SDRAM controller, a message buffer RAM, a message buffer mutex, and a tristate bridge to an external SRAM.

Figure 11–2. Example System Without Bridges—SOPC Builder View



Use	Connections	Module Name	Description	Base
☑		⊟ **cpu1**	Nios II Processor	
		instruction_master	Avalon Master	
		data_master	Avalon Master	IRQ 0
		jtag_debug_module	Avalon Slave	0x02002800
☑		⊟ **cpu2**	Nios II Processor	
		instruction_master	Avalon Master	
		data_master	Avalon Master	IRQ 0
		jtag_debug_module	Avalon Slave	0x00000800
☑		⊟ **cpu3**	Nios II Processor	
		instruction_master	Avalon Master	
		data_master	Avalon Master	IRQ 0
		jtag_debug_module	Avalon Slave	0x00001000
☑		⊟ **DDR_SDRAM_controller**	DDR SDRAM High Performance Control...	
		s1	Avalon Slave	0x01000000
☑		⊟ **message_buffer_RAM**	On-Chip Memory (RAM or ROM)	
		s1	Avalon Slave	0x02001000
☑		⊟ **message_buffer_mutex**	Mutex	
		s1	Avalon Slave	0x02003000
☑		⊟ **external_SSRAM_bus**	Avalon-MM Tristate Bridge	
		avalon_slave	Avalon Slave	0x00000000
		tristate_master	Avalon Tristate Master	
☑		⊟ **external_SSRAM**	Cypress CY7C1380C SSRAM	
		s1	Avalon Tristate Slave	0xffffffff

Figure 11–3 illustrates the default system interconnect fabric for the system in Figure 11–2.

Figure 11–3. Example System without Bridges—System Interconnect View

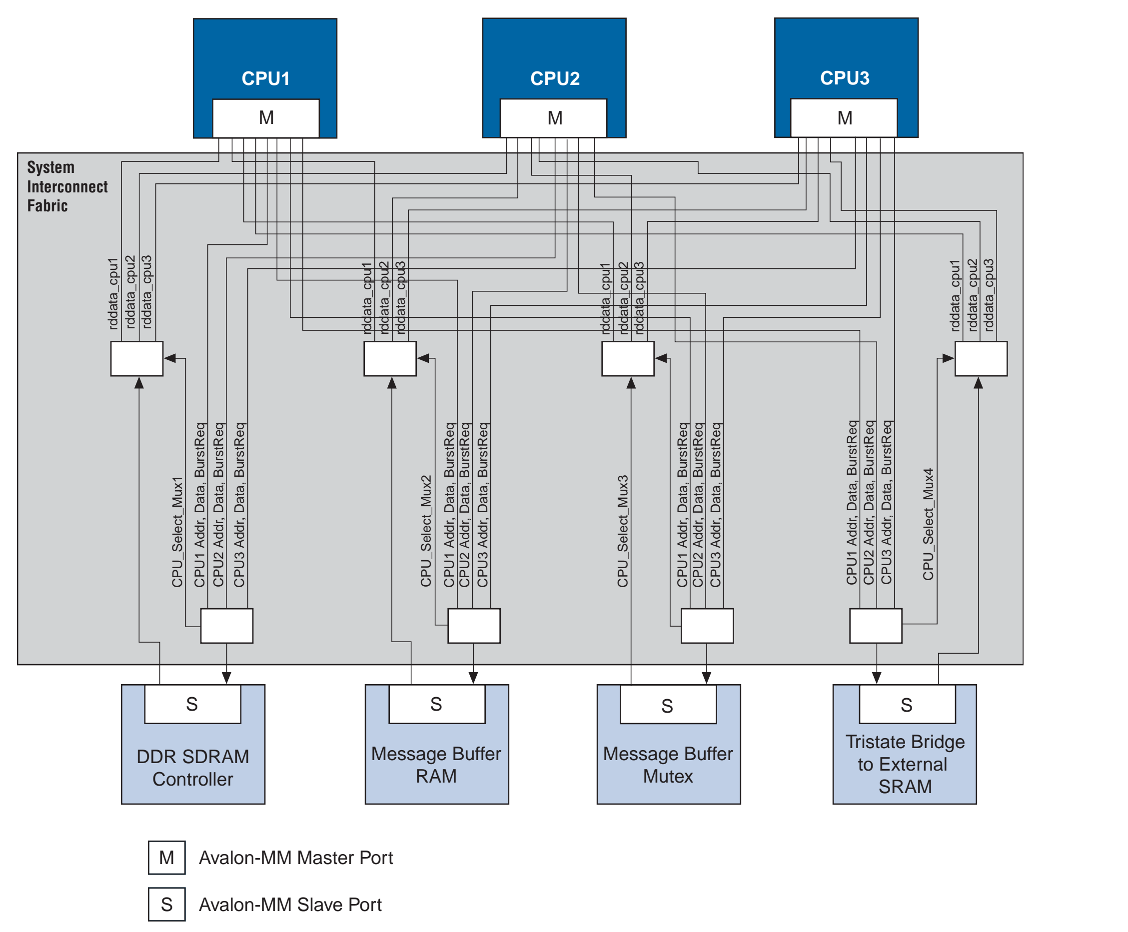


Figure 11–4 and Figure 11–5 show how inserting bridges can affect the generated logic. For example, if the DDR SDRAM controller can run at 166 MHz and the CPUs accessing it can run at 120 MHz, inserting an Avalon-MM clock-crossing bridge between the CPUs and the DDR SDRAM has the following benefits:

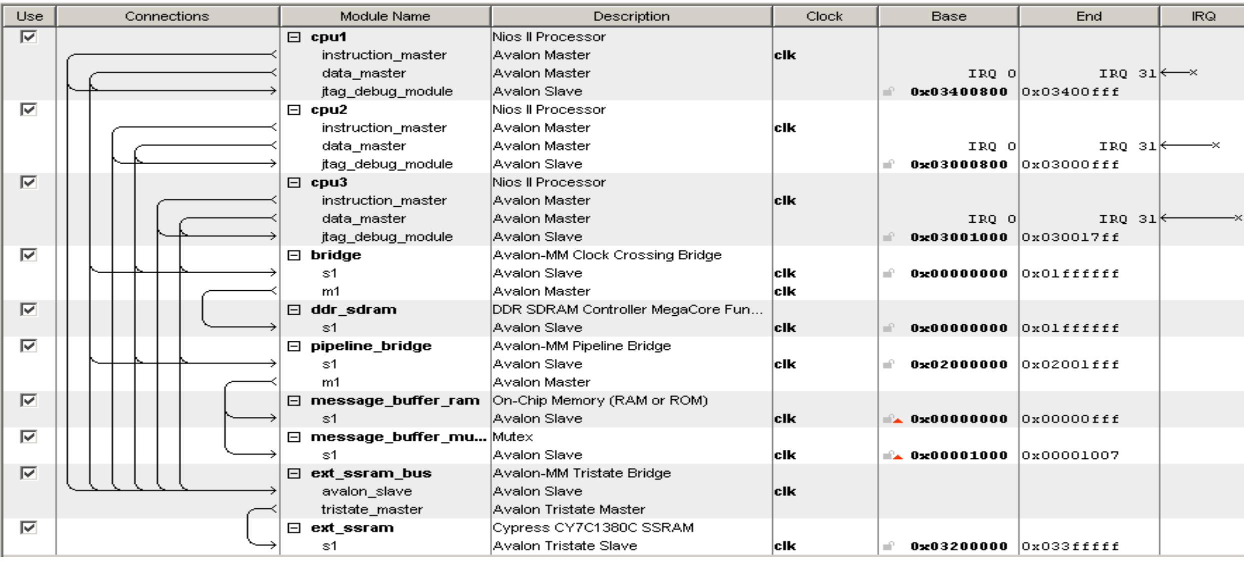
- Allows the CPU and DDR interfaces to run at different frequencies.
- Places system interconnect fabric for the arbitration logic and multiplexer for the DDR SDRAM controller in the slower clock domain.
- Reduces the complexity of the interconnect logic in the faster domain, allowing the system to achieve a higher f_{MAX}.

Inserting the clock-crossing bridge does increase read latency and may not be beneficial unless your system includes more devices that access the memory.

In the system illustrated in Figure 11–4, the message buffer RAM and message buffer mutex must respond quickly to the CPUs, but each response includes only a small amount of data. Placing an Avalon-MM pipeline bridge between the CPUs and the message buffers results in the following benefits:

- Eliminates separate arbiter logic for the message buffer RAM and message buffer mutex, which reduces logic utilization and propagation delay, thus increasing the f_{MAX}.
- Reduces the overall size and complexity of the system interconnect fabric.

Figure 11–4. Example SOPC System with Bridges—SOPC Builder View

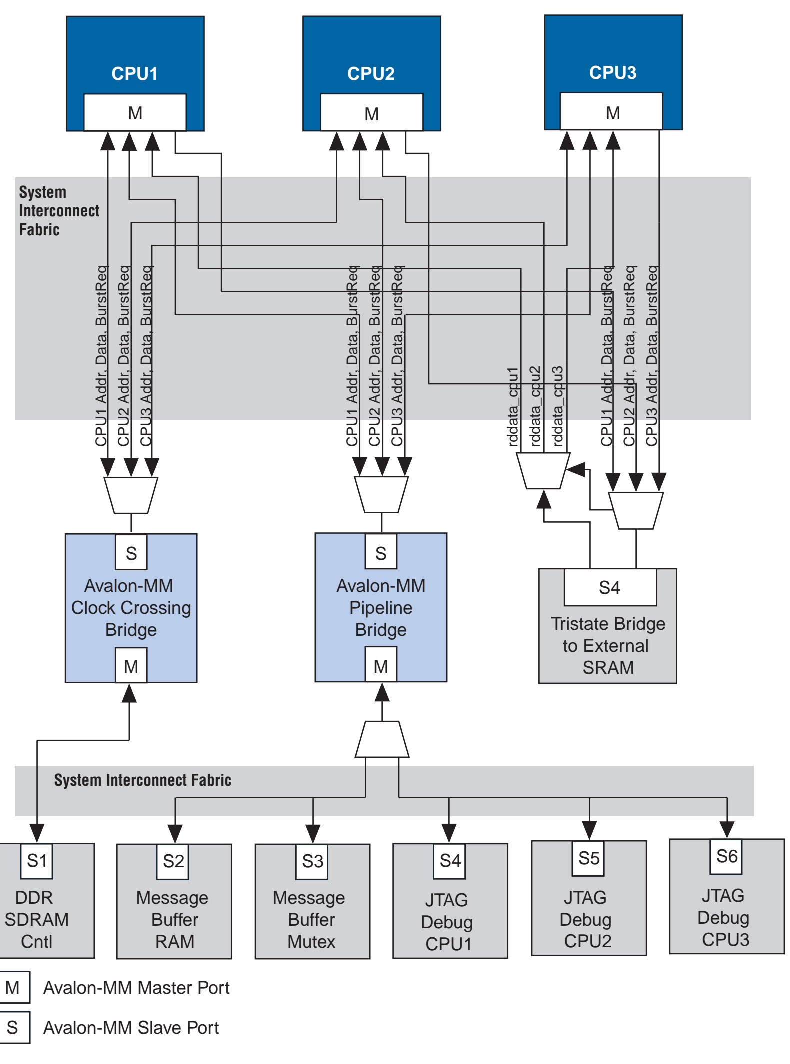


Use	Connections	Module Name	Description	Clock	Base	End	IRQ
☑		cpu1	Nios II Processor				
		instruction_master	Avalon Master	clk			
		data_master	Avalon Master		IRQ 0	IRQ 31	
		jtag_debug_module	Avalon Slave		0x03400800	0x03400fff	
☑		cpu2	Nios II Processor				
		instruction_master	Avalon Master	clk			
		data_master	Avalon Master		IRQ 0	IRQ 31	
		jtag_debug_module	Avalon Slave		0x03000800	0x03000fff	
☑		cpu3	Nios II Processor				
		instruction_master	Avalon Master	clk			
		data_master	Avalon Master		IRQ 0	IRQ 31	
		jtag_debug_module	Avalon Slave		0x03001000	0x030017ff	
☑		bridge	Avalon-MM Clock Crossing Bridge				
		s1	Avalon Slave	clk	0x00000000	0x01ffffff	
		m1	Avalon Master	clk			
☑		ddr_sdram	DDR SDRAM Controller MegaCore Fun...				
		s1	Avalon Slave	clk	0x00000000	0x01ffffff	
☑		pipeline_bridge	Avalon-MM Pipeline Bridge				
		s1	Avalon Slave	clk	0x02000000	0x02001fff	
		m1	Avalon Master				
☑		message_buffer_ram	On-Chip Memory (RAM or ROM)				
		s1	Avalon Slave	clk	0x00000000	0x00000fff	
☑		message_buffer_mu...	Mutex				
		s1	Avalon Slave	clk	0x00001000	0x00001007	
☑		ext_ssram_bus	Avalon-MM Tristate Bridge				
		avalon_slave	Avalon Slave	clk			
		tristate_master	Avalon Tristate Master				
☑		ext_ssram	Cypress CY7C1380C SSRAM				
		s1	Avalon Tristate Slave	clk	0x03200000	0x033fffff	

If an orange triangle appears next to an address in Figure 11–4, it indicates that the address is an offset value and is not the true value of the address in the address map.

Figure 11–5 shows the system interconnect fabric that SOPC Builder creates for the system in Figure 11–4. Figure 11–5 is the same system that is pictured in Figure 11–3 with bridges to control system topology.

Figure 11–5. Example System with a Bridge



Address Mapping for Systems with Avalon-MM Bridges

An Avalon-MM bridge has an address span and range that are defined as follows:

- The address *span* of an Avalon-MM bridge is the smallest power-of-two size that encompasses all of its slave's ranges.
- The address *range* of an Avalon-MM bridge is a numerical range from its base address to its base address plus its (span -1).

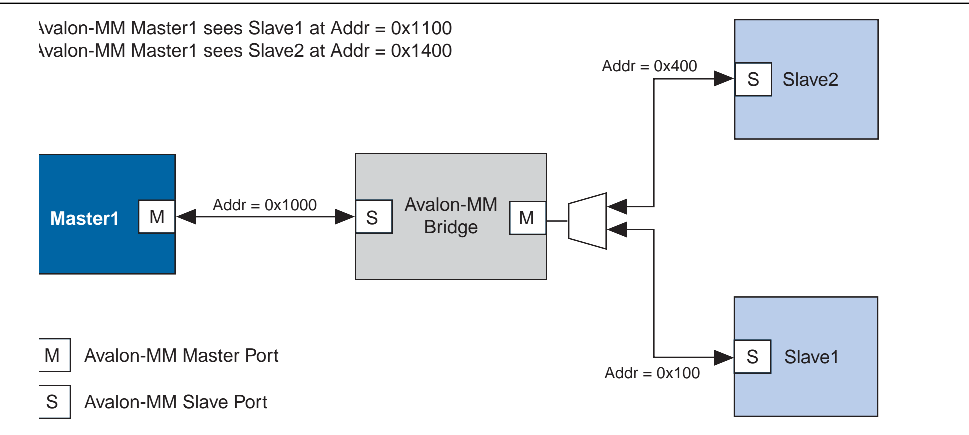
Equation 11–1.

range = [base_address .. (base_address + (span - 1)];

SOPC Builder follows several rules in constructing an address map for a system with Avalon-MM bridges:

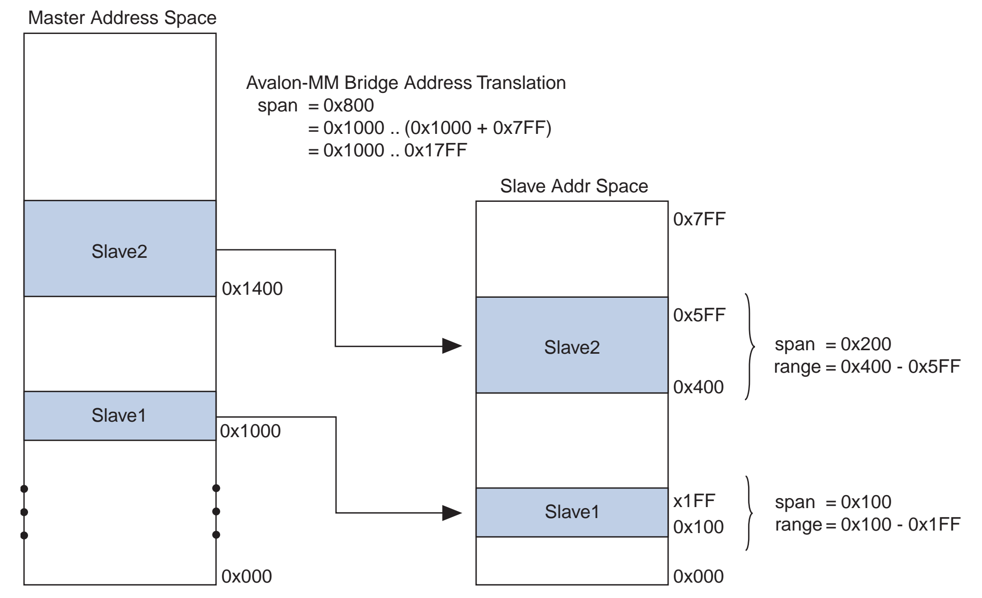
1. The address span of each Avalon-MM slave is rounded up to the nearest power of two.
2. Each Avalon-MM slave connected to a bridge has an address relative to the base address of the bridge. This address must be a multiple of its span. (See Figure 11–6.)

Figure 11–6. Avalon-MM Master and Slave Addresses



3. In the example shown in Figure 11–6, if the address span of Slave 1 is 0x100 and the address span of Slave 2 is 0x200, Figure 11–7 illustrates the address span of the Avalon-MM bridge.

Figure 11–7. The Address Span of an Avalon-MM Bridge



Tools for Visualizing the Address Map

The **Base Address** column of the **System Contents** tab displays the base address *offset* of the Avalon-MM slave relative to the base address of the Avalon-MM bridge to which it is connected. You can see the absolute address map for each master in the system by clicking **Address Map** on the **System Contents** tab.

Differences between Avalon-MM Bridges and Avalon-MM Tristate Bridges

You use Avalon-MM bridges to control topology and separate clock domains for on-chip components. You use tristate bridges to connect to off-chip components and to share pins, decreasing the overall pin count of the device. Tristate bridges are *transparent*, meaning that they do not affect the addresses of the components to which they connect.

For more information about the Avalon-MM tristate bridge, refer to the *SOPC Builder Memory Subsystem Development Walkthrough* chapter in volume 4 of the *Quartus II Handbook*.

Avalon-MM Pipeline Bridge

This section describes the hardware structure and functionality of the Avalon-MM pipeline bridge component.

Component Overview

The Avalon-MM pipeline bridge inserts registers in the path between its master and slaves. In a given SOPC Builder system, if the critical register-to-register delay occurs in the system interconnect fabric, the pipeline bridge can help reduce this delay and improve system f_{MAX}.

The bridge allows you to independently pipeline different groups of signals that can create a critical timing path in the interconnect:

- Master-to-slave signals, such as address, write data, and control signals
- Slave-to-master signals, such as read data
- The `waitrequest` signal to the master

You can also use the Avalon-MM pipeline bridge to control topology without adding a pipeline stage. To instantiate a bridge that does not add any pipeline stages, simply do not select any of the **Pipeline Options** on the parameter page. For the system illustrated in Figure 11–5, a pipeline bridge that does not add a pipeline register stage is optimal because the CPUs benefit from minimal delay from the message buffer mutex and message buffer RAM.

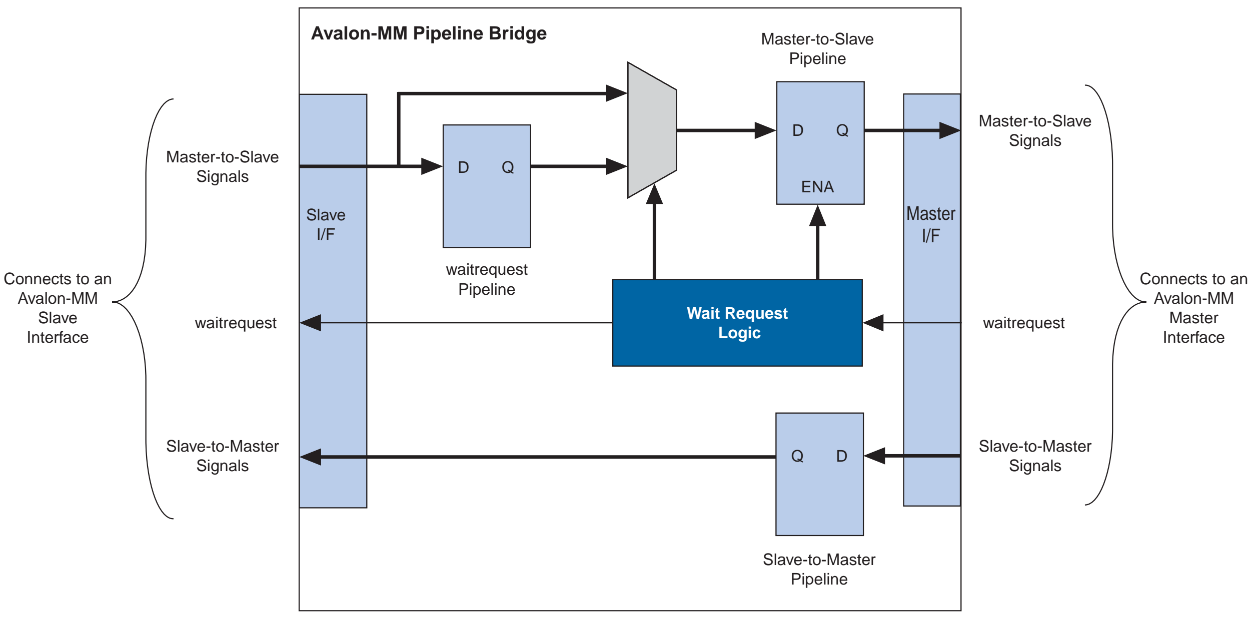
A pipeline bridge with no latency cannot be used with slaves that support pipelined reads. If a slave does not have read latency, you cannot connect it to a bridge with no pipeline stages, because the pipeline bridge slave port has a `readdatavalid` signal. Pipelined read components cannot have zero read latency. Some examples of 0 latency components available in SOPC Builder include the UART, Timer and SPI core. You you are connecting a pipeline bridge to one of these components, increase the read latency from 0 to 1.

The Avalon-MM pipeline bridge component integrates easily into any SOPC Builder system.

Functional Description

Figure 11–8 shows a block diagram of the Avalon-MM pipeline bridge component.

Figure 11–8. Avalon-MM Pipeline Bridge Block Diagram



The following sections describe the component's hardware functionality.

Interfaces

The bridge interface is composed of an Avalon-MM slave and an Avalon-MM master. The data width of the ports is configurable, which can affect how SOPC Builder generates dynamic bus sizing logic in the system interconnect fabric. Both ports support Avalon-MM pipelined transfers with variable latency. Both ports optionally support bursts of lengths that you can configure.

Pipeline Stages and Effects on Latency

The bridge provides three optional register stages to pipeline the following groups of signals.

- Master-to-slave signals, including:
  - `address`
  - `writedata`
  - `write`
  - `read`
  - `byteenable`
  - `chipselect`
  - `burstcount` (optional)
- Slave-to-master signals, including:
  - `readdata`
  - `readdatavalid`
- The `waitrequest` signal to the master

When you include a register stage, it affects the timing and latency of transfers through the bridge, as follows:

- The latency increases by one cycle in each direction.
- Write transfers on the master side of the bridge are decoupled from write transfers on the slave side of the bridge because Avalon-MM write transfers do not require an acknowledge signal from the slave.
- Including the `waitrequest` register stage increases the latency of master-to-slave signals by one additional cycle when the `waitrequest` signal is asserted.

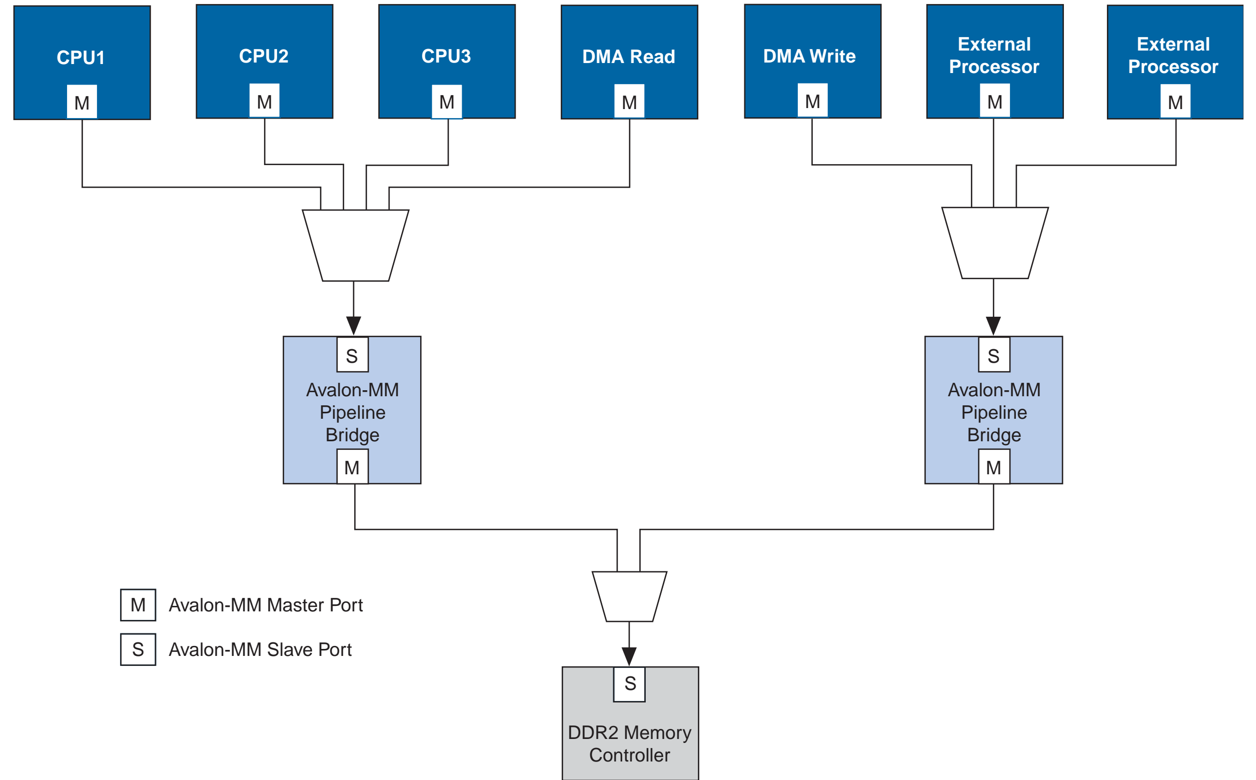
Burst Support

The bridge can support bursts with configurable maximum burst length. When configured to support bursts, the bridge propagates bursts between master-slave pairs, up to the maximum burst length. Not having burst support is equivalent to a maximum burst length of one. In this case, the system interconnect fabric automatically decomposes master-to-bridge bursts into a sequence of individual transfers.

Example System with Avalon-MM Pipeline Bridges

Figure 11–9 illustrates a system in which seven Avalon-MM masters are accessing a single DDR2 memory controller. By inserting two Avalon-MM pipeline bridges, you can limit the complexity of the multiplexer that would be required.

Figure 11–9. Seven Avalon-MM Masters Accessing One Avalon-MM Slave



Clock Crossing Bridge

The Avalon-MM clock-crossing bridge allows you to connect Avalon-MM master and slaves that operate in different clock domains. Without a bridge, SOPC Builder automatically includes generic clock domain crossing (CDC) logic in the system interconnect fabric, but it does not provide optimal performance for high-throughput applications. Because the clock-crossing bridge includes a buffering mechanism, you can pipeline multiple read and write transfers. After an initial penalty for the first read or write, there is no additional latency penalty for pending reads and writes, increasing throughput at the expense primarily of on-chip memory. The clock-crossing bridge has parameterizeable FIFOs for master-to-slave and slave-to-master signals, and allows burst transfers across clock domains.

The Avalon-MM clock-crossing bridge component is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system.

Choosing Clock Crossing Methodology

When determining clock frequencies for your components, you should also consider the impact on the latency that transferring data across clock domains can cause. Whether you use a clock-crossing bridge or rely on the clock domain adapter created automatically by SOPC Builder, additional latency occurs. You should also consider the resource usage and throughput capabilities of each solution.

If you rely on the automatically generated clock crossing adapter to connect master and slave ports driven by separate clock inputs, there is a fixed latency penalty associated to each transfer. Each transfer becomes blocking, meaning that while one transfer is underway another cannot begin until the first completes. For this reason, you should not connect high-speed, pipelined components such as SDRAM memory to a master on a different clock domain without using a clock-crossing bridge between them. The clock crossing bridge, on the other hand, can queue multiple transfers, so that even though the latency increases, the throughput does not decrease.

Because a clock crossing adapter is generated for every master and slave pair, you should use a clock crossing bridge if your design contains multiple master and slave pairs operating in different clock domains. Alternatively, if your design uses a large amount of on-chip memory, you may need to use a clock domain adapter, because the clock-crossing bridge uses on-chip memory resources for buffering.

Functional Description

Figure 11–10 shows a block diagram of the Avalon-MM clock-crossing bridge component. The following sections describe the component's hardware functionality.

Figure 11–10. Avalon-MM Clock-Crossing Bridge Block Diagram

Interfaces

The bridge interface comprises an Avalon-MM slave and an Avalon-MM master. The data width of the ports is configurable, which affects the size of the bridge hardware and how SOPC Builder generates dynamic bus sizing logic in the system interconnect fabric. Both ports support Avalon-MM pipelined transfers with variable latency. Both ports optionally support bursts of user-configurable length. Ideally, the settings for one port match the other, such that there are no mixed data widths or bursting capabilities.

Clock Crossing Bridge and FIFOs

Two FIFOs in the bridge transport address, data, and control signals across the clock domains. One FIFO captures data and controls traveling in the master-to-slave direction, and the other FIFO captures data in the slave-to-master direction. Clock crossing logic surrounding the FIFOs coordinates the details of passing data across the clock-domain boundaries and ensures that the FIFOs do not overflow or underflow.

The signals that pass through the master-to-slave FIFO include:

- `writedata`
- `address`
- `read`
- `write`
- `byteenable`
- `burstcount`, when bursting is enabled

The signals that pass through the slave-to-master FIFO include:

- `readdata`
- `readdatavalid`

You can configure the depth of each FIFO. Because there are more signals traveling in the master-to-slave direction, changing the depth of the master-to-slave FIFO has a greater impact on the memory utilization of the bridge.

For read transfers across the bridge, the FIFOs in both directions incur latency for data to return from the slave. To avoid paying a latency penalty for each transfer, the master can issue multiple reads that are queued in the FIFO. The slave of the bridge asserts `readdatavalid` when it drives valid data and asserts `waitrequest` when it is not ready to accept more reads.

For write transfers, the master-to-slave FIFO causes a delay between the master-to-bridge transfers and the corresponding bridge-to-slave transfers. Because Avalon-MM write transfers do not require an acknowledge from the slave, multiple write transfers from master-to-bridge might complete by the time the bridge initiates the corresponding bridge-to-slave transfers.

Burst Support

The bridge optionally supports bursts with configurable maximum burst length. When configured to support bursts, the bridge propagates bursts between master-slave pairs, up to the maximum burst length. Not having burst support is equivalent to a maximum burst length of one. In this case, the system interconnect fabric automatically breaks master-to-bridge bursts into a sequence of individual transfers.

When you configure the bridge to support bursts, you must configure the slave-to-master FIFO depth deeply enough to capture all burst read data without overflowing. The masters connected to the bridge could potentially fill the master-to-slave FIFO with read burst requests; therefore, the minimum slave-to-master FIFO depth is described by equation given in Example 11–1.

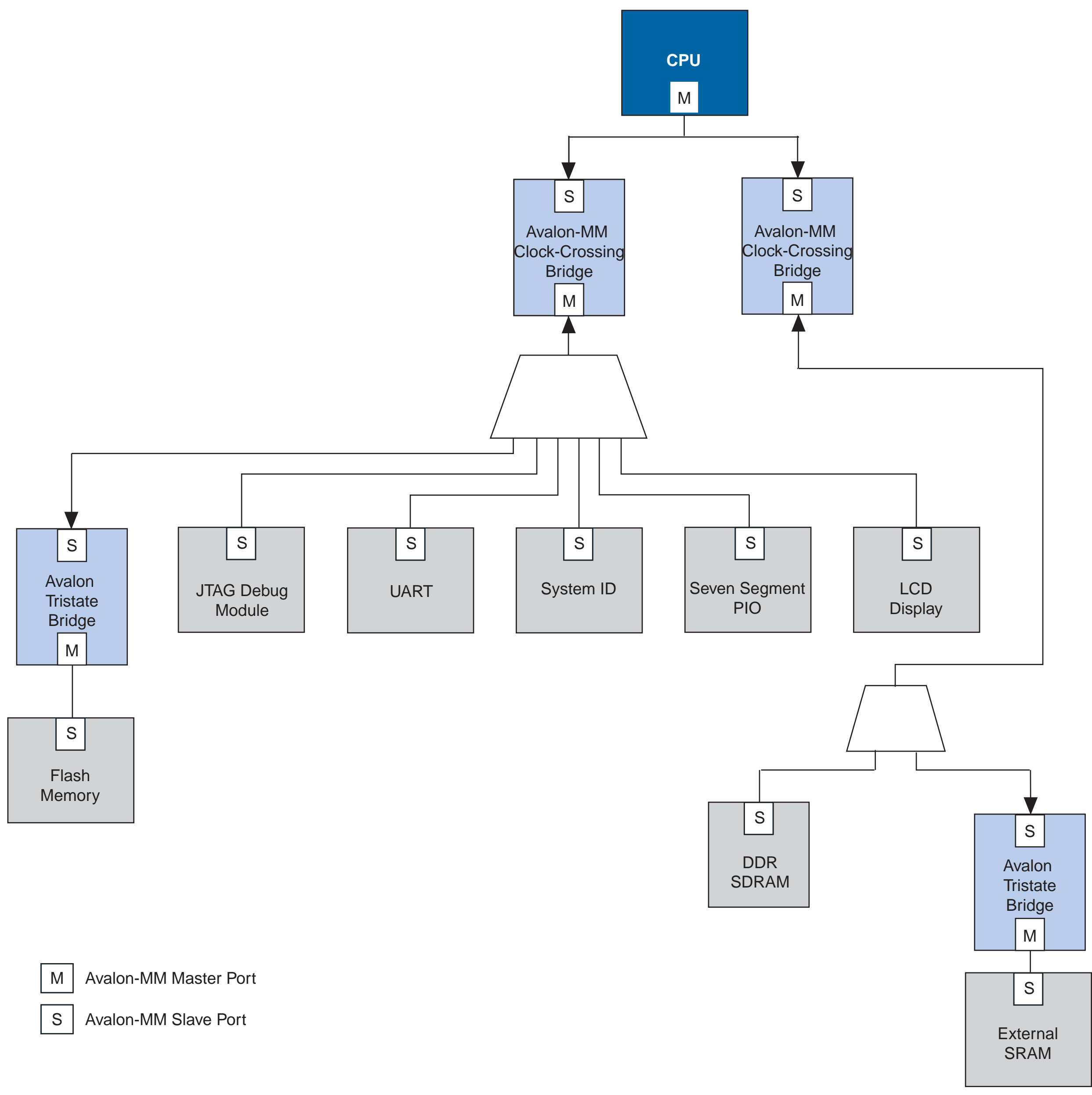
Example 11–1. Minimum Slave-To-Master FIFO Depth

```
= ((master-to-slave FIFO depth) * (max burst length)) + max slave latency/pending reads
```

Example System with Avalon-MM Clock-Crossing Bridges

Figure 11–11 uses Avalon-MM clocking crossing bridges to separate slave components into two groups. The low-performance slave components are placed behind a single bridge and clocked at a low speed. The high performance components are placed behind a second bridge and clocked at a higher speed. By inserting clock-crossing bridges in the system, you optimize the interconnect fabric and allow the Quartus® II fitter to expend effort optimizing paths that require minimal propagation delay.

Figure 11–11. One Avalon-MM Master with Two Groups of Avalon-MM Slaves



Instantiating the Avalon-MM Clock-Crossing Bridge in SOPC Builder

Table 11–1 describes the options available on the **Parameter Settings** page of the MegaWizard™ interface.

Table 11–1. Avalon-MM Clock Crossing Bridge Parameters

Master-to-slave FIFO		
Parameter	**Value**	**Description**
FIFO depth	**8, 16, 32**	Determines the depth of the FIFO.
Construct FIFO with registers instead of memory blocks	**On/Off**	When you turn on this option, the FIFO uses registers as storage instead of embedded memory blocks. This can considerably increase the size of the bridge hardware and lower the f_{MAX}.
Slave-to-master FIFO		
FIFO depth	**8, 16, 32, 64, 128, 256, 512, 1024**	Determines the depth of the FIFO.
Construct FIFO with registers instead of memory blocks	**On/Off**	When you turn on this option, the FIFO uses registers as storage instead of embedded memory blocks. This can considerably increase the size of the bridge hardware and lower the f_{MAX}.
Common options		
Data width	**8, 16, 32, 64, 128, 256,512, 1024**	Determines the data width of the interfaces on the bridge, and affects the size of both FIFOs. For the highest bandwidth, set **Data width** to be as wide as the widest master connected to the bridge.
Slave domain synchronizer length	**2–8**	The number of pipeline stages in the clock crossing logic in the issuing master to target slave direction. Increasing this value leads to a larger meantime between failures (MTBF). You can determine the MTBF for a given design can be determined by running a TimeQuest timing analysis.
Master domain synchronizer length	**2–8**	The number of pipeline stages in the clock crossing logic in the issuing master to target slave direction. Increasing this value leads to a larger meantime between failures (MTBF). You can determine the MTBF for a given design can be determined by running a TimeQuest timing analysis.
Burst settings		
Allow bursts	**On/Off**	Includes logic for the bridge's master and slaves to support bursts. You can use this option to restrict the minimum depth for the slave-to-master FIFO.
Maximum burst size	**2, 4, 8, 16, 32, 64, 128, 256, 512, 1024**	Determines the maximum length of bursts for the bridge to support, when you turn on **Allow bursts**.

Clock Domain Crossing Logic

SOPC Builder generates CDC logic that hides the details of interfacing components operating in different clock domains. The system interconnect fabric upholds the Avalon-MM protocol with each port independently, and therefore masters do not need to incorporate clock adapters in order to interface to slaves on a different domain. The system interconnect fabric logic propagates transfers across clock domain boundaries automatically.

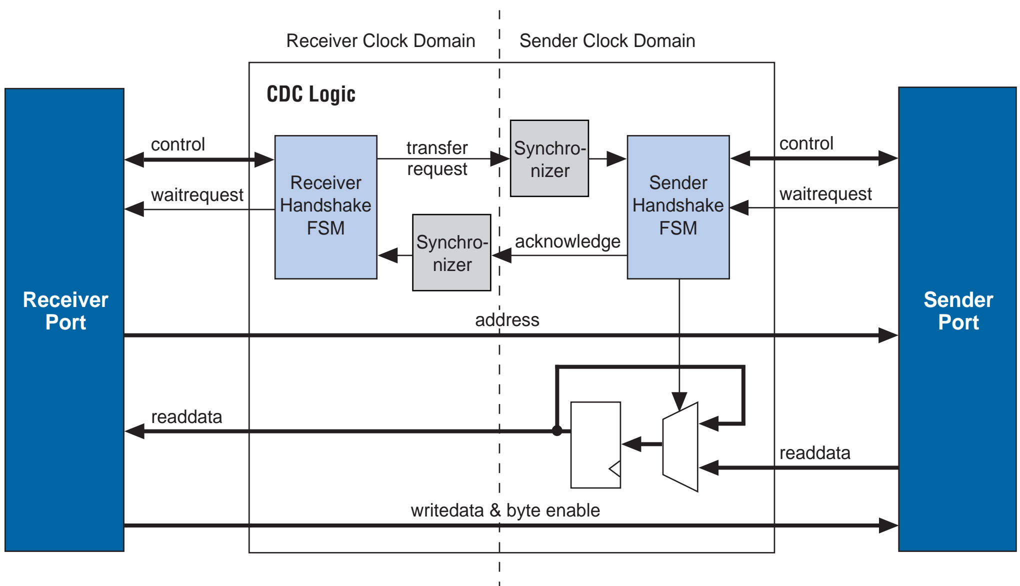
The clock-domain adapters in the system interconnect fabric provide the following benefits that simplify system design efforts:

- Allow component interfaces to operate at different clock frequencies.
- Eliminate the need to design CDC hardware.
- Allow each Avalon-MM port to operate in only one clock domain, which reduces design complexity of components.
- Enable masters to access any slave without communication with the slave clock domain.
- Allow you to focus performance optimization efforts only on components that require fast clock speed.

Description of Clock Domain Adapter

The clock domain adapter consists of two finite state machines (FSM), one in each clock domain, that use a simple hand-shaking protocol to propagate transfer control signals (`read_request`, `write_request`, and the master `waitrequest` signals) across the clock boundary. Figure 11–12 shows a block diagram of the clock domain adapter between one master and one slave.

Figure 11–12. Block Diagram of Clock Crossing Adapter



The synchronizer blocks in Figure 11–12 use multiple stages of flipflops to eliminate the propagation of metastable events on the control signals that enter the handshake FSMs.

The CDC logic works with any clock ratio. Altera tests the CDC logic extensively on a variety of system architectures, both in simulation and in hardware, to ensure that the logic functions correctly.

The typical sequence of events for a transfer across the CDC logic is described as follows:

1. Master asserts address, data, and control signals.
2. The master handshake FSM captures the control signals, and immediately forces the master to wait.

The FSM uses only the control signals, not address and data. For example, the master simply holds the address signal constant until the slave side has safely captured it.

3. Master handshake FSM initiates a transfer request to the slave handshake FSM.
4. The transfer request is synchronized to the slave clock domain.
5. The slave handshake FSM processes the request, performing the requested transfer with the slave.
6. When the slave transfer completes, the slave handshake FSM sends an acknowledge back to the master handshake FSM.
7. The acknowledge is synchronized back to the master clock domain.
8. The master handshake FSM completes the transaction by releasing the master from the wait condition.

Transfers proceed as normal on the slave and the master side, without a special protocol to handle crossing clock domains. From the perspective of a slave, there is nothing different about a transfer initiated by a master in a different clock domain. From the perspective of a master, a transfer across clock domains simply requires extra clock cycles. Similar to other transfer delay cases (for example, arbitration delay or wait states on the slave side), the system interconnect fabric simply forces the master to wait until the transfer terminates. As a result, pipeline master ports do not benefit from pipelining when performing transfers to a different clock domain.

Location of Clock Domain Adapter

You can use the clock crossing bridge described in the following paragraphs for higher throughput clock crossing, at the expense of memory resources.

SOPC Builder automatically determines where to insert the CDC logic, based on the system contents and the connections between components. SOPC Builder places CDC logic to maintain the highest transfer rate for all components. SOPC Builder evaluates the need for CDC logic for each master and slave pair independently, and generates CDC logic wherever necessary.

Duration of Transfers Crossing Clock Domains

CDC logic extends the duration of master transfers across clock domain boundaries. In the worst case which is for reads, each transfer is extended by five master clock cycles and five slave clock cycles. Assuming the default value of 2 for the **Master domain synchronizer length** and the **Slave domain synchronizer length**, the components of this delay are the following:

- Four additional master clock cycles, due to the master-side clock synchronizer
- Four additional slave clock cycles, due to the slave-side clock synchronizer
- One additional clock in each direction, due to potential metastable events as the control signals cross clock domains

Systems that require a higher performance clock should use the Avalon-MM clock crossing bridge instead of the automatically inserted CDC logic. The clock crossing bridge includes a buffering mechanism, so that multiple reads and writes can be pipelined. After paying the initial penalty for the first read or write, there is no additional latency penalty for pending reads and writes, increasing throughput by up to four times, at the expense of added logic resources.

For more information, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook* and *Avalon Memory-Mapped Design Optimizations* in the *Embedded Design Handbook.*

Implementing Multiple Clock Domains in SOPC Builder

You specify the clock domains used by your system on the **System Contents** tab of SOPC Builder. You define the input clocks to the system with the **Clock Settings** table. Clock sources can be driven by external input signals to the SOPC Builder system or by PLLs inside the SOPC Builder system. Clock domains are differentiated based on the name of the clock. You may create multiple asynchronous clocks with the same frequency.

To specify which clock drives which components you must display the **Clock** column in the **System Contents** tab. By default, clock names are not displayed. To display clock names in the **Module Name** column and the clocks in the **Clock** column in the **System Contents** tab, right-click in the **Module Name** column and click **Show All**. To connect a clock to follow these steps.

1. Click in the **Clock** column next to the clock port. A list of available clock signals appears.
2. Select the appropriate signal from the list of available clocks. Figure 11–13 illustrates this step.

Figure 11–13. Assigning Clocks to Components

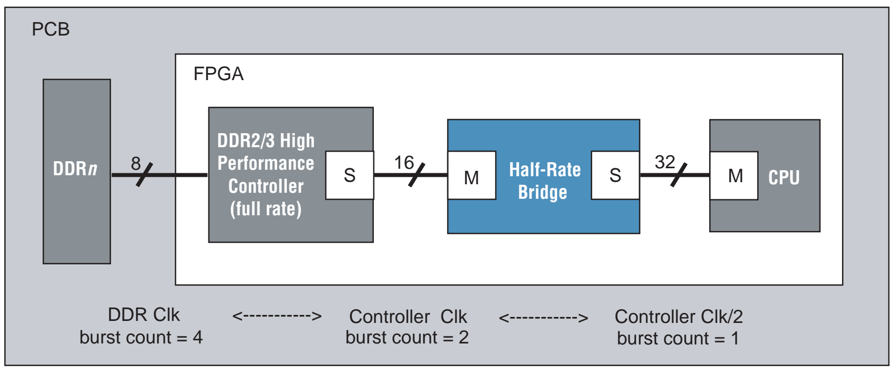
Module Name	Description	Clock	Base	End	IRQ
⊞ high_res_timer	Interval timer	clk	0x02120820	0x0212083F	3
⊞ seven_seg_pio	PIO (Parallel I/O)	clk	0x02120890	0x0212089F	
⊞ reconfig_request_pio	PIO (Parallel I/O)	fastclk	0x021208A0	0x021208AF	
⊞ uart1	UART (RS-232 serial port)	clk	0x02120840	0x0212085F	4
⊞ sysid	System ID Peripheral	clk	0x021208B8	0x021208BF	
⊞ sdram	SDRAM Controller	clk	0x01000000	0x01FFFFFF	
⊞ dma_0	DMA	fastclk	0x00800000	0x0080001F	7
⊞ read_buffer	On-Chip Memory (RAM ...	fastclk	0x00801000	0x00801FFF	
⊞ write_buffer	On-Chip Memory (RAM ...	fastclk	0x00802000	0x00802FFF	

Avalon-MM DDR Memory Half-Rate Bridge

The Avalon Memory-Mapped (MM) Half-Rate Bridge core is a special-purpose clock-crossing bridge intended for CPUs that require low-latency access to high-speed memory. The core works under the assumption that the memory clock is twice the frequency of the CPU clock, with zero phase shift between the two. It allows high speed memory to run at full rate while providing low-latency interface for a CPU to access it by using lightweight logic that translates one single-word request into a two-word burst to a memory running at twice the clock frequency and half the width. For systems with a 8-bit DDR interface, using the Half-Rate DDR Bridge in conjunction with a DDR SDRAM high-performance memory controller creates a datapath that matches the throughput of the DDR memory to the CPU. This half-rate bridge provides the same functionality as the clock crossing bridge, but with significantly lower latency—2 cycles instead of 12.

The core's master interface is designed to be connected to a high-speed DDR SDRAM controller and thus only supports bursting. Because the slave interface is designed to receive single-word requests, it does not support bursting. Figure 11–14 shows a system including an 8-bit DDR memory, a high-performance memory controller, the Half-Rate DDR Bridge, and a CPU.

Figure 11–14. SOPC Builder Memory System Using a DDR Memory Half-Rate Bridge



The Avalon-MM DDR Memory Half-Rate Bridge core has the following features and requirements:

- SOPC Builder ready with TimeQuest Timing Analyzer constraints

© March 2009 Altera Corporation

- Requires master clock and slave clock to be synchronous
- Handles different bus sizes between CPU and memory
- Requires the frequency of the master clock to be double of the slave clock
- Has configurable address and data port widths in the master interface

Resource Usage and Performance

This section lists the resource usage and performance data for supported devices when operating the Half-Rate Bridge with a full-rate DDR SDRAM high-performance memory controller.

Using the Half-Rate Bridge with a full-rate DDR SDRAM high-performance memory controller results an average of 48% performance improvement over a system using a half-rate DDR SDRAM high-performance memory controller in a series of embedded applications. The performance improvement is 62.2% based on the Dhrystone benchmark, and 87.7% when accessing memory bypassing the cache. For memory systems that use the Half-Rate bridge in conjunction with DDR2/3 High Performance Controller, the data throughput is the same on the Half-Rate Bridge master and slave interfaces. The decrease in memory latency on the Half-Rate Bridge slave interface results in higher performance for the processor.

Table 11–2 shows the resource usage for Stratix® II and Stratix III devices.

Table 11–2. Resource Utilization Data for Stratix Devices

Device Family	Combinational ALUTs	ALMs	Logic Register	Memory M512/M4K/ M-RAM
Stratix II	58	143	153	0
Stratix III	59	135	154	0

Table 11–3 lists the resource usage for a Cyclone® III device.

Table 11–3. Resource Utilization Data for Cyclone III Devices

Logic Cells (LC)	Logic Register	LUT-only LC	Register-only LC	LUT/Register LCs	Memory M512/M4K/ M-RAM
233	152	30	84	119	0

Functional Description

The Avalon MM DDR Memory Half Rate Bridge works under two constraints:

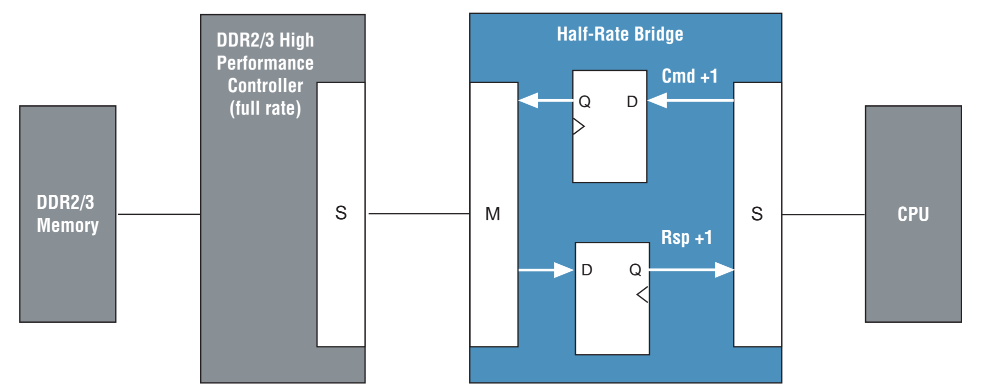
- Its memory-side master has a clock frequency that is synchronous (zero phase shift) to, and twice the frequency of, the CPU-side slave.
- Its memory-side master is half as wide as its CPU-side slave.

The bridge leverages these two constraints to provide lightweight, low-latency clock-crossing logic between the CPU and the memory. These constraints are in contrast with the Avalon-MM Clock-Crossing Bridge, which makes no assumptions about the frequency/phase relationship between the master- and slave-side clocks, and provides higher-latency logic that fully-synchronizes all signals that pass between the two domains.

The Avalon MM DDR Memory Half-Rate Bridge has an Avalon-MM slave interface that accepts single-word (non-bursting) transactions. When the slave interface receives a transaction from a connected CPU, it issues a two-word burst transaction on its master interface (which is half as wide and twice as fast). If the transaction is a read request, the bridge's master interface waits for the slave's two-word response, concatenates the two words, and presents them as a single readdata word on its slave interface to the CPU. Every time the data width is halved, the clock rate is doubled. As a result, the data throughput is matched between the CPU and the off-chip memory device.

Figure 11–15 shows the latency in the Avalon-MM Half-Rate Bridge core. The core adds two cycles of latency in the slave clock domain for read transactions. The first cycle is introduced during the command phase of the transaction and the second cycle, during the response phase of the transaction. The total latency is 2+<*x*>, where <*x*> refers to the latency of the DDR SDRAM high-performance memory controller. Using the clock crossing bridge for this same purpose would impose approximately 12 cycles of additional latency.

Figure 11–15. Avalon-MM DDR Memory Half-Rate Bridge Block Diagram



Instantiating the Core in SOPC Builder

Use the MegaWizard Plug-In Manager for the Avalon-MM Half-Rate Bridge core in SOPC Builder to specify the core's configuration. Table 11–4 describes the parameters that can be configured for the Avalon-MM Half-Rate Bridge core.

Table 11–4. Configurable Parameters for Avalon-MM DDR Memory Half-Rate Bridge Core

Parameters	Value	Description
Data Width	8, 16, 32, 64, 128, 256, 512	The width of the data signal in the master interface.
Address Width	1 - 32	The width of the address signal in the master interface.

Table 11–5 describes the parameters that are derived based on the **Data Width** and **Address Width** settings for the Avalon-MM Half-Rate Bridge core.

Table 11–5. Derived Parameters for Avalon-MM DDR Memory Half-Rate Bridge Core

Parameter	Description
Master interface's Byte Enable Width	The width of the byte-enable signal in the master interface.
Slave interface's Data Width	The width of the data signal in the slave interface.
Slave interface's Address Width	The width of the address signal in the slave interface.
Slave interface's Byte Enable Width	The width of the byte-enable signal in the slave interface.

Example System

The following example provides high-level steps showing how the Avalon-MM DDR Memory Half-Rate Bridge core is connected in a system. This example assumes that you are familiar with the SOPC Builder GUI.

For a quick introduction to this tool, read of the one-hour online course, *Using SOPC Builder*.

1. Add a **Nios II Processor** to the system.
2. Add a **DDR2 SDRAM High-Performance Controller** and configure it to **full-rate** mode.
3. Add **Avalon-MM DDR Memory Half-Rate Bridge** to the system.
4. Configure the parameters of the Avalon-MM DDR Memory Half-Rate Bridge based on the memory controller. For example, for a 32 MByte DDR memory controller in full rate mode with 8 DQ pins (see Figure 11–14), the parameters should be set as the following:
   - **Data Width** = 16

     For a memory controller that has 8 DQ pins, its local interface width is 16 bits. The local interface width and the data width must be the same, therefore data width is set to 16 bits.
   - **Address Width** = 25

     For a memory capacity of 32 MBytes, the byte address is 25 bits. Because the master address of the bridge is byte aligned, the address width is set to 25 bits.
5. Connect `altmemddr_auxhalf` to the slave clock interface (`clk_s1`) of the Half-Rate Bridge.
6. Connect `altmemddr_sysclk` to the master clock interface (`clk_m1`) of the Half-Rate Bridge.
7. Remove all connections between Nios II processor and the memory controller, if there are any.
8. Connect the master interface (`m1`) of the Avalon-MM DDR Memory Half-Rate Bridge to the memory controller slave interface.

9. Connect the slave interface (`s1`) of the Avalon-MM DDR Memory Half-Rate Bridge to the Nios II processor `data_master` interface.
10. Connect `altmemddr_auxhalf` to Nios II processor clock interface.

Device Support

Altera device support for the bridge components is listed in Table 11–6.

Table 11–6. Device Family Support

Device Family	Avalon-MM Pipeline Bridge Support	Avalon-MM Clock-Crossing Bridge Support
Arria® GX	Full	Full
Arria II GX	Full	Full
Stratix®	Full	Full
Stratix II	Full	Full
Stratix II GX	Full	Full
Stratix III	Full	Full
Stratix IV	Full	Full
Cyclone®	Full	Full
Cyclone II	Full	Full
Cyclone III	Full	Full
Hardcopy®	Full	Full
HardCopy II	Full	Full
HardCopy III	Full	Full
MAX®	Full	No support
MAX II	Full	No support

Hardware Simulation Considerations

The bridge components do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard SOPC Builder simulation flow to simulate the component design files inside an SOPC Builder system.

Software Programming Model

The bridge components do not have any user-visible control or status registers. Therefore, software cannot control or configure any aspect of the bridges during run-time. The bridges cannot generate interrupts.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Avalon Memory-Mapped Design Optimizations* in the *Embedded Design Handbook*

- *DDR and DDR2 SDRAM High-Performance Controller User Guide*
- *DDR3 SDRAM High-Performance Controller User Guide*
- *SOPC Builder Memory Subsystem Development Walkthrough* chapter in volume 4 of the *Quartus II Handbook*
- *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*

Document Revision History

Table 11–7 shows the revision history for this chapter.

Table 11–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009, v9.0.0	■ Added information for synchronization when crossing clock domains.	New information to allow user control of metastability.
November 2008 v8.1	■ Clarified connection of clock signals. ■ Added section describing half-rate bridge. ■ Changed page size to 8.5 x 11 inches.	—
May 2008 v8.0	■ Chapter renumbered from 10 to 11. ■ Corrected Figure 11–4 to show correct connectivity between masters and bridges. Show JTAG debug modules for each CPU behind pipeline bridge. ■ Deleted references to Avalon Memory-Mapped and Streaming Interface Specifications and replaced with new Avalon Interface Specifications. ■ Moved clock crossing bridge section from Chapter 2 to this chapter. ■ Added note after Figure 10-4.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

12. Avalon Streaming Interconnect Components

QII54021-9.0.0

Introduction to Interconnect Components

Avalon® Streaming (Avalon-ST) interconnect components facilitate the design of high-speed, low-latency datapaths for the system-on-a-programmable-chip (SOPC) environment. Interconnect components in SOPC Builder act as a part of the system interconnect fabric. They are not end points, but adapters that allow you to connect different, but compatible, streaming interfaces. You use Avalon-ST interconnect components to connect cores that send and receive high-bandwidth data, including multiplexed streams, packets, cells, time-division multiplexed (TDM) frames, and digital signal processor (DSP) data.

The interconnect components that you add to an SOPC Builder system insert logic between a source and sink interface, enabling that interface to operate correctly. This chapter describes four Avalon-ST interconnect components, also called adapters:

- "Timing Adapter" on page 12–3—adapts between sinks and sources that have different characteristics, such as ready latencies.
- "Data Format Adapter" on page 12–6—adapts source and sink interfaces that have different data widths.
- "Channel Adapter" on page 12–8—adapts source and sink interfaces that have different settings for the `channel` signal.
- "Error Adapter" on page 12–9—ensures that per-bit error information recorded at the source is correctly transferred to the sink

All of these interconnect components adapt initially incompatible Avalon-ST source and sink interfaces so that they function correctly, facilitating the development of high-speed, low-latency datapaths.

Interconnect Component Usage

Interconnect components can adapt the data or control signals of the Avalon-ST interface. Typical adaptations to control signals include:

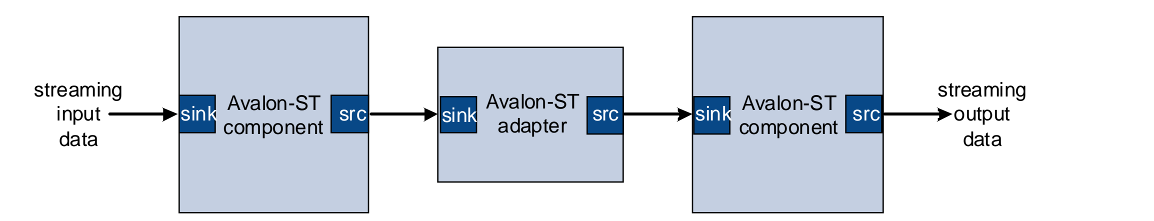
- Adding pipeline stages to adjust the timing of the `ready` signal
- Tying signals that are not used by either the source or sink to 0 or 1

Typical adaptations to data signals include:

- Changing the number of symbols (words) that are driven per cycle
- Changing the number of channels driven

When the interconnect component adapts the data interface, it has one Avalon-ST sink interface and one Avalon-ST source interface, as shown in Figure 12–1. You configure the adapter components manually, using SOPC Builder. In contrast to the Avalon-MM interface, which allows you to create various topologies with a number of different master and slave components, you always use the Avalon-ST interconnect components to adapt point-to-point connections between streaming cores.

Figure 12–1. Example of an Avalon-ST Interconnect Component in an SOPC Builder System

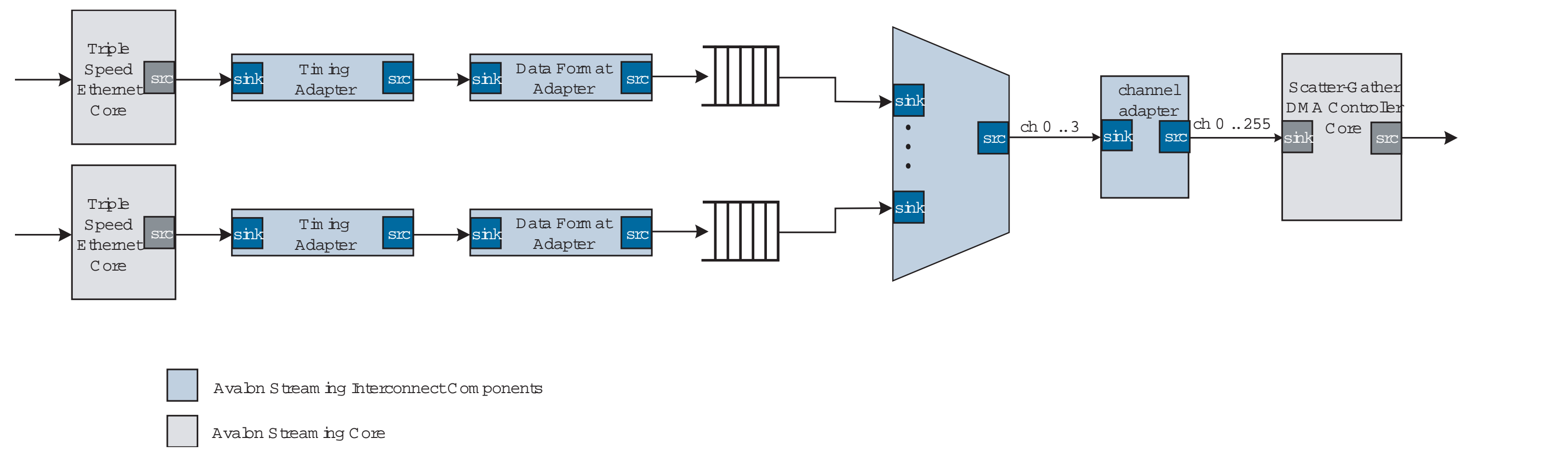


For details about the system interconnect fabric, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*. For details about the Avalon-ST interface protocol, refer to the *Avalon Interface Specifications*.

Figure 12–2 illustrates a datapath that connects a Triple Speed Ethernet MegaCore function to a Scatter-Gather DMA controller core using a timing adapter, data format adapter, and channel adapter so that the cores can interoperate.

Address Mapping

Figure 12–2. Avalon-ST Datapath Constructed Using Avalon Streaming Interconnect Components



The control and status signals for the components containing source or sink interfaces can be mapped to a slave interface which is then accessible in the global Avalon address space.

Timing Adapter

The timing adapter has two functions:

- It adapts source and sink interfaces that support the `ready` signal and those that do not.
- It adapts source and sink interfaces that support the `valid` signal and those that do not.
- It adapts source and sink interfaces that have different ready latencies.

The timing adapter treats all signals other than the `ready` and `valid` signals as payload, and simply drives them from the source to the sink. Table 12–1 outlines the adaptations that the timing adapter provides.

Table 12–1. Timing Adapter

Condition	Adaptation
The source has `ready`, but the sink does not.	In this case, the source can respond to backpressure, but the sink never needs to apply it. The `ready` input to the source interface is connected directly to logical 1.
The source does not have `ready`, but the sink does.	The sink may apply backpressure, but the source is unable to respond to it. There is no logic that the adapter can insert that prevents data loss when the source asserts `valid` but the sink is not ready. The adapter provides simulation time error messages and an error indication if data is ever lost. The user is presented with a warning, and the connection is allowed.
The source and sink both support backpressure, but the sink's ready latency is greater than the source's.	The source responds to `ready` assertion or deassertion faster than the sink requires it. A number of pipeline stages equal to the difference in ready latency are inserted in the `ready` path from the sink back to the source, causing the source and the sink to see the same cycles as `ready` cycles.
The source and sink both support backpressure, but the sink's ready latency is less than the source's.	The source cannot respond to `ready` assertion or deassertion in time to satisfy the sink. A buffer whose depth is equal to the difference in ready latency is inserted to compensate for the source's inability to respond in time.

Resource Usage and Performance

Resource utilization for the timing adapter depends upon the function that it performs. Table 12–2 provides estimated resource utilization for seven different configurations of the timing adapter

Table 12–2. Timing Adapter Estimated Resource Usage and Performance

Input Ready Latency	Output Ready Latency	Stratix® II and Stratix II GX (Approximate LEs)			Cyclone® II		Stratix (Approximate LEs)		
		f_{MAX} (MHz)	ALM Count	Mem Bits	f_{MAX} (MHz)	Logic Cells	f_{MAX} (MHz)	Logic Cells	Mem Bits
1	2	500	2	0	420	2	422	1	0
1	3	500	2	0	420	3	422	2	0
1	4	500	4	0	420	4	422	3	0
1	0	500	21	80	420	183	422	20	80
2	1	456	21	80	401	188	317	21	80
3	1	456	21	80	401	188	317	21	80
4	1	456	21	80	401	188	317	21	80

Instantiating the Timing Adapter in SOPC Builder

You can use the Avalon-ST configuration wizard in SOPC Builder to specify the hardware features. Table 12–3 describes the options available on the **Parameter Settings** page of the configuration wizard

Table 12–3. Avalon-ST Timing Adapter Parameters

Input Interface Parameters	
Parameter	**Description**
Support Backpressure with the ready signal	Turn on this option to add the backpressure functionality to the interface.
Ready Latency	When the `ready` signal is used, the value for `ready_latency` indicates the number of cycles between when the `ready` signal is asserted and when valid data is driven.
Include valid signal	Turn this option on if the interface includes the `valid` signal. Turning this option off means that data being received is always valid.
Output Interface Parameters	
Support Backpressure with the ready signal	Turn on this option to add the backpressure functionality to the interface.
Ready Latency	When the `ready` signal is used, the value for `ready_latency` indicates the number of cycles between when the `ready` signal is asserted and when valid data is driven.
Include valid signal	Turn this option on if the interface includes the `valid` signal. Turning this option off means that data driven is always valid.
Common to Input and Output Interfaces	
Channel Signal Width (bits)	Type the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is eight bits. Set to 0 if channels are not used.
Max Channel	Type the maximum number of channels that the interface supports. Valid values are 0–255.
Data Bits Per Symbol	Type the number of bits per symbol.
Data Symbols Per Beat	Type the number of symbols per active transfer.
Include Packet Support	Turn this option on if the interfaces supports a packet protocol, including the `startofpacket`, `endofpacket` and `empty` signals.
Include Empty Signal	You can use this signal to specify the number of empty symbols in the cycle that includes the `endofpacket` signal. This signal is not necessary if the number of symbols per beat is 1.
Error Signal Width (Bits)	Type the width of the error signal. Valid values are 0–31 bits. Type `0` if the error signal is not used.
Error Signal Description	Type the description for each of the error bits. Separate the description fields by semicolons. For a connection to be made, the description of the error bits in the source and sink must match. Refer to “Error Adapter” on page 12–9 for the adaptations that can be made when the bits do not match.

Data Format Adapter

The data format adapter handles interfaces that have different definitions for the data signal. One of the more common adaptations that this component performs is data width adaptation, such as converting a data interface that drives two, 8-bit symbols per beat to an interface that drives four, 8-bit symbols per beat. The available data format adaptations are listed in Table 12–4.

Table 12–4. Data Format Adapter

Condition	Description of Adapter Logic
The source and sink's bits per symbol are different.	The connection cannot be made.
The source and sink have a different number of symbols per beat.	The adapter converts from the source's width to the sink's width. If the adaptation is from a wider to a narrower interface, a beat of data at the input corresponds to multiple beats of data at the output. If the input `error` signal is asserted for a single beat, it is asserted on output for multiple beats. If the adaptation is from a narrow to a wider interface, multiple input beats are required to fill a single output beat, and the output `error` is the logical OR of the input `error` signal.

Resource Usage and Performance

Resource utilization for the data format adapter depends upon the function that it performs. Table 12–5 provides estimated resource utilization for numerous configurations of the data format adapter.

Table 12–5. Data Format Adapter Estimated Resource Usage and Performance, 8 Bits per Symbol (Part 1 of 2)

Input Symbols per Beat	Output Symbols per Beat	Number of Channels	Packet Support	Stratix II and Stratix II GX (Approximate LEs)			Cyclone II			Stratix (Approximate LEs)		
				f_{MAX} (MHz)	ALM Count	Memory Bits	f_{MAX} (MHz)	Logic Cells	Memory Bits	f_{MAX} (MHz)	Logic Cells	Memory Bits
1	2	1	y	500	96	0	391	93	0	375	105	0
4	1	1	y	459	106	0	311	97	0	306	76	0
4	2	1	y	500	118	0	343	107	0	326	85	0
4	8	1	y	437	326	0	346	370	0	303	330	0
4	16	1	y	357	930	0	264	1005	0	231	806	0
1	2	188	y	321	110	15	187	137	15	209	153	15
4	1	105	y	244	125	2	148	183	2	150	137	2
4	2	105	y	277	101	2	172	134	2	173	108	2
4	8	130	y	322	255	41	175	279	41	187	262	41
4	16	30	y	268	341	106	166	563	106	153	471	106
4	1	105	n	269	107	2	177	185	2	167	99	2
4	2	54	n	290	109	1	193	203	1	176	91	1
4	3	10	n	249	149	18	189	251	16	159	217	18
4	5	222	n	281	300	40	199	381	40	182	316	40
4	6	30	n	312	184	40	201	385	40	198	241	40
4	7	139	n	253	285	56	159	416	56	161	427	56

Table 12–5. Data Format Adapter Estimated Resource Usage and Performance, 8 Bits per Symbol (Part 2 of 2)

Input Symbols per Beat	Output Symbols per Beat	Number of Channels	Packet Support	Stratix II and Stratix II GX (Approximate LEs)			Cyclone II			Stratix (Approximate LEs)		
				f_{MAX} (MHz)	ALM Count	Memory Bits	f_{MAX} (MHz)	Logic Cells	Memory Bits	f_{MAX} (MHz)	Logic Cells	Memory Bits
4	8	198	n	311	281	40	190	247	40	198	257	40
4	15	160	n	259	370	121	165	733	121	149	697	121
4	16	36	n	227	255	105	391	93	0	146	491	105

Instantiating the Data Format Adapter in SOPC Builder

You can use the Avalon-ST configuration wizard in SOPC Builder to specify the hardware features. Table 12–6 describes the options available on the **Parameter Settings** page of the configuration wizard.

Table 12–6. Data Format Adapter Parameters

Input Interface Parameters	
Parameter	**Description**
Data Symbols Per Beat	Type the number of symbols transferred per active cycle.
Include the empty signal	Turn this option on if the cycle that includes the `endofpacket` signal can include empty symbols. This signal is not necessary if the number of symbols per beat is 1.
Output Interface Parameters	
Data Symbols Per Beat	Type the number of symbols transferred per active cycle.
Include the empty signal	Turn this option on if the cycle that includes the `endofpacket` signal can include empty symbols. This signal is not necessary if the number of symbols per beat is 1.
Common to Input & Output	
Channel Signal Width (bits)	Type the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the channel signal is 8 bits. Type `0` if you do not need to send channel numbers.
Max Channel	Type the maximum number of channels that the interface supports. Valid values are 0–255.
Include Packet Support	Turn this option on if the interface supports a packet protocol, including the `startofpacket`, `endofpacket`, and `empty` signals.
Error Signal Width (Bits)	Type the width of the error signal. Valid values are 0–31 bits. Type `0` if the error signal is not used.
Error Signal Description	Type the description for each of the error bits. Separate the description fields by semicolons. For a connection to be made, the description of the error bits in the source and sink must match. Refer to "Error Adapter" on page 12–9 for the adaptations that can be made when the bits do not match.
Data Bits Per Symbol	Type the number of bits per symbol.

Channel Adapter

The channel adapter provides adaptations between interfaces that have different support for the `channel` signal or for the maximum number of channels supported. The adaptations are described in Table 12–7.

Table 12–7. Channel Adapter

Condition	Description of Adapter Logic
The source uses channels, but the sink does not.	You are given a warning at generation time. The adapter provides a simulation error and signals an error for data for any channel from the source other than 0.
The sink has channel, but the source does not.	You are given a warning, and the channel inputs to the sink are all tied to a logical 0.
The source and sink both support channels, and the source's maximum number of channels is less than the sink's.	The source's channel is connected to the sink's channel unchanged. If the sink's channel signal has more bits, the higher bits are tied to a logical 0.
The source and sink both support channels, but the source's maximum number of channels is greater than the sink's.	The source’s channel is connected to the sink’s channel unchanged. If the source’s channel signal has more bits, the higher bits are left unconnected. You are given a warning that channel information may be lost. An adapter provides a simulation error message and an error indication if the value of channel from the source is greater than the sink's maximum number of channels. In addition, the `valid` signal to the sink is deasserted so that the sink never sees data for channels that are out of range.

Resource Usage and Performance

The channel adapter typically uses fewer than 30 LEs. Its frequency is limited by the maximum frequency of the device you choose.

Instantiating the Channel Adapter in SOPC Builder

You can use the Avalon-ST configuration wizard in SOPC Builder to specify the hardware features. Table 12–8 describes the options available on the **Parameter Settings** page of the configuration wizard.

Table 12–8. Avalon-ST Channel Adapter Parameters (Part 1 of 2)

Parameter	Description
Input Interface Parameters	
Channel Signal Width (bits)	Type the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is eight bits. Set to 0 if channels are not used.
Max Channel	Type the maximum number of channels that the interface supports. Valid values are 0–255.
Output Interface Parameters	
Channel Signal Width (bits)	Type the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is eight bits. Set to 0 if channels are not used.
Max Channel	Type the maximum number of channels that the interface supports. Valid values are 0–255.

Table 12–8. Avalon-ST Channel Adapter Parameters (Part 2 of 2)

Parameter	Description
Common to Input and Output Interfaces	
Support Backpressure with the ready signal	Turn on this option to add the backpressure functionality to the interface.
Ready Latency	When the `ready` signal is used, the value for `ready_latency` indicates the number of cycles between when the `ready` signal is asserted and when valid data is driven.
Data Bits Per Symbol	Type the number of bits per symbol.
Data Symbols Per Beat	Type the number of symbols per active transfer.
Include Packet Support	Turn this option on if the interfaces supports a packet protocol, including the `startofpacket`, `endofpacket` and `empty` signals.
Include Empty Signal	You can use this signal to specify the number of empty symbols in the cycle that includes the `endofpacket` signal. This signal is not necessary if the number of symbols per beat is 1.
Error Signal Width (bits)	Type the width of the error signal. Valid values are 0–31 bits. Type 0 if you do not need to send error values.
Error Signal Description	Type the description for each of the error bits. Separate the description fields by semicolons. For a connection to be made, the description of the error bits in the source and sink must match. Refer to "Error Adapter" on page 12–9 for the adaptations that can be made when the bits do not match.

Error Adapter

The error adapter ensures that per-bit error information provided by source interfaces is correctly connected to the sink interface's input error signal. The adaptations are described in Table 12–9.

Instantiating the Error Adapter in SOPC Builder

You can use the Avalon-ST configuration wizard in SOPC Builder to specify the hardware features. Table 12–9 describes the options available on the **Parameter Settings** page of the configuration wizard.

Table 12–9. Avalon-ST Error Adapter Parameters (Part 1 of 2)

Parameter	Description
Input Interface Parameters	
Error Signal Width (bits)	Type the width of the error signal. Valid values are 0–31 bits. Type 0 if the error signal is not used.
Error Signal Description	Type the description for each of the error bits. Separate the description fields by semicolons. For a connection to be made, the description of the error bits in the source and sink must match. Refer to "Error Adapter" on page 12–9 for the adaptations that can be made when the bits do not match.
Output Interface Parameters	
Error Signal Width (bits)	Type the width of the error signal. Valid values are 0–31 bits. Type 0 if you do not need to send error values.

Table 12–9. Avalon-ST Error Adapter Parameters (Part 2 of 2)

Parameter	Description
Error Signal Description	Type the description for each of the error bits. Separate the description fields by semicolons. For a connection to be made, the description of the error bits in the source and sink must match. Refer to "Error Adapter" on page 12–9 for the adaptations that can be made when the bits do not match.
Common to Input and Output Interfaces	
Support Backpressure with the ready signal	Turn on this option to add the backpressure functionality to the interface.
Ready Latency	When the `ready` signal is used, the value for `ready_latency` indicates the number of cycles between when the ready signal is asserted and when valid data is driven.
Channel Signal Width (bits)	Type the width of the `channel` signal. A channel width of 4 allows up to 16 channels. The maximum width of the `channel` signal is eight bits. Set to 0 if channels are not used.
Max Channel	Type the maximum number of channels that the interface supports. Valid values are 0–255.
Data Bits Per Symbol	Type the number of bits per symbol.
Data Symbols Per Beat	Type the number of symbols per active transfer.
Include Packet Support	Turn this option on if the interfaces supports a packet protocol, including the `startofpacket`, `endofpacket` and `empty` signals.
Include Empty Signal	Turn this option on if the cycle that includes the `endofpacket` signal can include empty symbols. This signal is not necessary if the number of symbols per beat is 1.

Installation and Licensing

The Avalon-ST interconnect components are included in the Altera MegaCore® IP Library, which is part of the Quartus II software installation. After you install the MegaCore IP Library, SOPC Builder recognizes these components and can instantiate them into a system.

You can use the Avalon-ST components without a license in any design that targets an Altera device.

Hardware Simulation Considerations

The Avalon-ST interconnect components do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard SOPC Builder simulation flow to simulate the component design files inside an SOPC Builder system.

Software Programming Model

The Avalon-ST interconnect components do not have any control or status registers that you can see. Therefore, software cannot control or configure any aspect of the interconnect components at run-time. These components cannot generate interrupts.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*

Document Revision History

Table 12–10 shows the revision history for this chapter.

Table 12–10. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009, v9.0.0	■ No changes from previous release.	—
November 2008, v8.1.1	■ Removed private comments	—
November 2008, v8.1.0	■ Added documentation for Avalon-ST error adapter. ■ Reformatted parameter settings in tables. ■ Changed page size to 8.5 x 11 inches.	Minor changes for 8.1.
May 2008, v8.0.0	■ Chapter renumbered from 11 to 12. ■ Deleted references to Avalon Memory-Mapped and Streaming Interface Specifications and changed to Avalon Interface Specifications.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

Additional Information

About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 9.0.

How to Contact Altera

For the most up-to-date information about Altera products, see the following table.

Contact *(Note 1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note:

(1) You can also contact your local Altera sales office or sales representative.

Third-Party Software Product Information

Third-party software products described in this handbook are not Altera products, are licensed by Altera from third parties, and are subject to change without notice. Updates to these third-party software products may not be concurrent with Quartus II software releases. Altera has assumed responsibility for the selection of such third-party software products and its use in the Quartus II 9.0 software release. To the extent that the software products described in this handbook are derived from third-party software, no third party warrants the software, assumes any liability regarding use of the software, or undertakes to furnish you any support or information relating to the software. EXCEPT AS EXPRESSLY SET FORTH IN THE APPLICABLE ALTERA PROGRAM LICENSE SUBSCRIPTION AGREEMENT UNDER WHICH THIS SOFTWARE WAS PROVDED TO YOU, ALTERA AND THIRD-PARTY LICENSORS DISCLAIM ALL WARRANTIES WITH RESPECT TO THE USE OF SUCH THIRD-PARTY SOFTWARE CODE OR DOCUMENTATION IN THE SOFTWARE, INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT. For more information, including the latest available version of specific third-party software products, refer to the documentation for the software in question.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, file names, file name extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: <*file name*>, <*project name*>**.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
☞	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
👣	The feet direct you to more information on a particular topic.

Quartus II Handbook Version 9.0

Volume 5: Embedded Peripherals

101 Innovation Drive
San Jose, CA 95134
www.altera.com

QII5V5-9.0

Copyright © 2009 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

I.S. EN ISO 9001

Contents

Chapter Revision Dates xvii

Section I. Off-Chip Interface Peripherals

Chapter 1. SDRAM Controller Core

Core Overview 1-1
Functional Description 1-2
Avalon-MM Interface 1-2
Off-Chip SDRAM Interface 1-3
Signal Timing and Electrical Characteristics 1-3
Synchronizing Clock and Data Signals 1-3
Clock Enable (CKE) Not Supported 1-3
Sharing Pins with Other Avalon-MM Tri-State Devices 1-3
Board Layout and Pinout Considerations 1-4
Performance Considerations 1-4
Open Row Management 1-4
Sharing Data and Address Pins 1-4
Hardware Design and Target Device 1-5
Device Support 1-5
Instantiating the Core in SOPC Builder 1-5
Memory Profile Page 1-6
Timing Page 1-7
Hardware Simulation Considerations 1-7
SDRAM Controller Simulation Model 1-8
SDRAM Memory Model 1-8
Using the Generic Memory Model 1-8
Using the SDRAM Manufacturer's Memory Model 1-8
Example Configurations 1-8
Software Programming Model 1-10
Clock, PLL and Timing Considerations 1-10
Factors Affecting SDRAM Timing 1-11
Symptoms of an Untuned PLL 1-11
Estimating the Valid Signal Window 1-11
Example Calculation 1-13
Referenced Documents 1-15
Document Revision History 1-16

Chapter 2. CompactFlash Core

Core Overview 2-1
Functional Description 2-1
Instantiating the Core in SOPC Builder 2-2
Required Connections 2-2
Device Support 2-3
Software Programming Model 2-3
HAL System Library Support 2-3
Software Files 2-4
Register Maps 2-4
Ide Registers 2-4

Ctl Registers . . . 2-4
Document Revision History . . . 2-5

Chapter 3. Common Flash Interface Controller Core

Core Overview . . . 3-1
Functional Description . . . 3-2
Device and Tools Support . . . 3-2
Instantiating the Core in SOPC Builder . . . 3-2
Attributes Page . . . 3-3
Presets Settings . . . 3-3
Size Settings . . . 3-3
Timing Page . . . 3-3
Software Programming Model . . . 3-4
HAL System Library Support . . . 3-4
Limitations . . . 3-4
Software Files . . . 3-4
Referenced Documents . . . 3-5
Document Revision History . . . 3-5

Chapter 4. EPCS Device Controller Core

Core Overview . . . 4-1
Functional Description . . . 4-1
Avalon-MM Slave Interface and Registers . . . 4-3
Device and Tools Support . . . 4-4
Instantiating the Core in SOPC Builder . . . 4-4
Software Programming Model . . . 4-4
HAL System Library Support . . . 4-4
Software Files . . . 4-4
Referenced Documents . . . 4-5
Document Revision History . . . 4-5

Chapter 5. JTAG UART Core

Core Overview . . . 5-1
Functional Description . . . 5-2
Avalon Slave Interface and Registers . . . 5-2
Read and Write FIFOs . . . 5-2
JTAG Interface . . . 5-3
Host-Target Connection . . . 5-3
Device and Tools Support . . . 5-4
Instantiating the Core in SOPC Builder . . . 5-4
Configuration Page . . . 5-4
Write FIFO Settings . . . 5-4
Read FIFO Settings . . . 5-5
Simulation Settings . . . 5-5
Simulated Input Character Stream . . . 5-5
Prepare Interactive Windows . . . 5-5
Hardware Simulation Considerations . . . 5-6
Software Programming Model . . . 5-6
HAL System Library Support . . . 5-6
Driver Options: Fast vs. Small Implementations . . . 5-8
ioctl() Operations . . . 5-8
Software Files . . . 5-9
Accessing the JTAG UART Core via a Host PC . . . 5-9

Register Map 5-9
Data Register 5-10
Control Register 5-10
Interrupt Behavior 5-11
Referenced Documents 5-12
Document Revision History 5-12

Chapter 6. UART Core
Core Overview 6-1
Functional Description 6-1
Avalon-MM Slave Interface and Registers 6-2
RS-232 Interface 6-2
Transmitter Logic 6-2
Receiver Logic 6-3
Baud Rate Generation 6-3
Device Support 6-3
Instantiating the Core in SOPC Builder 6-3
Configuration Settings 6-4
Baud Rate Options 6-4
Data Bits, Stop Bits, Parity 6-5
Synchronizer Stages 6-5
Flow Control 6-5
Streaming Data (DMA) Control 6-6
Simulation Settings 6-6
Simulated RXD-Input Character Stream 6-7
Prepare Interactive Windows 6-7
Simulated Transmitter Baud Rate 6-7
Simulation Considerations 6-7
Software Programming Model 6-8
HAL System Library Support 6-8
Driver Options: Fast Versus Small Implementations 6-9
ioctl() Operations 6-10
Limitations 6-10
Software Files 6-10
Register Map 6-11
rxdata Register 6-11
txdata Register 6-12
status Register 6-12
control Register 6-14
divisor Register (Optional) 6-14
endofpacket Register (Optional) 6-15
Interrupt Behavior 6-15
Referenced Documents 6-15
Document Revision History 6-16

Chapter 7. SPI Core
Core Overview 7-1
Functional Description 7-1
Example Configurations 7-2
Transmitter Logic 7-3
Receiver Logic 7-3
Master and Slave Modes 7-3
Master Mode Operation 7-3

Slave Mode Operation 7-4
Multi-Slave Environments 7-5
Avalon-MM Interface 7-5
Instantiating the SPI Core in SOPC Builder 7-5
Master/Slave Settings 7-5
Number of Select (SS_n) Signals 7-5
SPI Clock (sclk) Rate 7-5
Specify Delay 7-6
Data Register Settings 7-6
Timing Settings 7-7
Device Support 7-8
Software Programming Model 7-8
Hardware Access Routines 7-8
alt_avalon_spi_command() 7-9
Software Files 7-9
Register Map 7-9
rxdata Register 7-10
txdata Register 7-10
status Register 7-11
control Register 7-12
slaveselect Register 7-12
Referenced Documents 7-12
Document Revision History 7-13

Chapter 8. Optrex 16207 LCD Controller Core

Core Overview 8-1
Functional Description 8-1
Device and Tools Support 8-2
Instantiating the Core in SOPC Builder 8-2
Software Programming Model 8-2
HAL System Library Support 8-2
Displaying Characters on the LCD 8-3
Software Files 8-3
Register Map 8-4
Interrupt Behavior 8-4
Referenced Documents 8-4
Document Revision History 8-4

Chapter 9. PIO Core

Core Overview 9-1
Functional Description 9-1
Data Input and Output 9-2
Edge Capture 9-2
IRQ Generation 9-3
Example Configurations 9-3
Avalon-MM Interface 9-3
Instantiating the PIO Core in SOPC Builder 9-4
Basic Settings 9-4
Width 9-4
Direction 9-4
Output Port Reset Value 9-4
Output Register 9-4
Input Options 9-4

Edge Capture Register 9-4
Interrupt 9-5
Simulation 9-5
Device Support 9-5
Software Programming Model 9-5
Software Files 9-5
Register Map 9-6
data Register 9-6
direction Register 9-6
interruptmask Register 9-7
edgecapture Register 9-7
outset and outclear Registers 9-7
Interrupt Behavior 9-7
Software Files 9-7
Document Revision History 9-8

Chapter 10. Avalon-ST JTAG Interface Core
Core Overview 10-1
Functional Description 10-1
Interfaces 10-2
Special characters 10-2
Operation 10-2
Instantiating the Core in SOPC Builder 10-3
Device Support 10-3
Referenced Documents 10-3
Document Revision History 10-3

Chapter 11. Avalon-ST Serial Peripheral Interface Core
Core Overview 11-1
Functional Description 11-1
Interfaces 11-2
Operation 11-2
Timing 11-3
Limitations 11-3
Instantiating the Core in SOPC Builder 11-3
Device Support 11-3
Referenced Documents 11-4
Document Revision History 11-4

Chapter 12. SPI Slave/JTAG to Avalon Master Bridge Cores
Core Overview 12-1
Functional Description 12-1
Instantiating the Core in SOPC Builder 12-3
Device Support 12-3
Referenced Documents 12-4
Document Revision History 12-4

Chapter 13. PCI Lite Core
Core Overview 13-1
Performance and Resource Utilization 13-1
Functional Description 13-2
PCI-Avalon Bridge Blocks 13-2
Avalon-MM Ports 13-3

Master and Target Performance . . . 13-5
Master Performance . . . 13-5
Target Performance . . . 13-5
PCI-to-Avalon Address Translation . . . 13-6
Avalon-to-PCI Address Translation . . . 13-6
Avalon-To-PCI Read and Write Operation . . . 13-8
Avalon-to-PCI Write Requests . . . 13-9
Avalon-to-PCI Read Requests . . . 13-9
Ordering of Requests . . . 13-10
PCI Interrupt . . . 13-10
Instantiating the Core in SOPC Builder . . . 13-11
PCI Timing Constraint Files . . . 13-12
Additional Tcl Option . . . 13-13
Device Support . . . 13-14
Simulation Considerations . . . 13-14
Features . . . 13-14
Master Transactor (mstr_tranx) . . . 13-14
TASKS Sections . . . 13-14
INITIALIZATION Section . . . 13-15
USER COMMANDS Section . . . 13-15
Simulation Flow . . . 13-15
Referenced Documents . . . 13-17
Document Revision History . . . 13-17

Section II. On-Chip Storage Peripherals

Chapter 14. Avalon-ST Single Clock and Dual Clock FIFO Cores

Core Overview . . . 14-1
Functional Description . . . 14-1
Interfaces . . . 14-2
Operations . . . 14-2
Instantiating the Core in SOPC Builder . . . 14-3
Device Support . . . 14-3
Software Programming Model . . . 14-4
HAL System Library Support . . . 14-4
Register Map . . . 14-4
Referenced Documents . . . 14-4
Document Revision History . . . 14-4

Chapter 15. On-Chip FIFO Memory Core

Core Overview . . . 15-1
Functional Description . . . 15-1
Avalon-MM Write Slave to Avalon-MM Read Slave . . . 15-2
Avalon-ST Sink to Avalon-ST Source . . . 15-2
Avalon-MM Write Slave to Avalon-ST Source . . . 15-3
Avalon-ST Sink to Avalon-MM Read Slave . . . 15-4
Status Interface . . . 15-5
Clocking Modes . . . 15-6
Device Support . . . 15-6
Instantiating the Core in SOPC Builder . . . 15-6
FIFO Settings . . . 15-6
Depth . . . 15-6
Clock Settings . . . 15-6

 © March 2009 Altera Corporation

Status Port 15-6
FIFO Implementation 15-6
Interface Parameters 15-7
Input 15-7
Output 15-7
Allow Backpressure 15-7
Avalon-MM Port Settings 15-7
Avalon-ST Port Settings 15-7
Software Programming Model 15-8
HAL System Library Support 15-8
Software Files 15-8
Programming with the On-Chip FIFO Memory 15-8
Software Control 15-9
Software Example 15-12
On-Chip FIFO Memory API 15-13
altera_avalon_fifo_init() 15-13
altera_avalon_fifo_read_status() 15-13
altera_avalon_fifo_read_ienable() 15-14
altera_avalon_fifo_read_almostfull() 15-14
altera_avalon_fifo_read_almostempty() 15-14
altera_avalon_fifo_read_event() 15-14
altera_avalon_fifo_read_level() 15-15
altera_avalon_fifo_clear_event() 15-15
altera_avalon_fifo_write_ienable() 15-15
altera_avalon_fifo_write_almostfull() 15-16
altera_avalon_fifo_write_almostempty() 15-16
altera_avalon_write_fifo() 15-16
altera_avalon_write_other_info() 15-17
altera_avalon_fifo_read_fifo() 15-17
Referenced Documents 15-18
Document Revision History 15-18

Chapter 16. Avalon-ST Multi-Channel Shared Memory FIFO Core
Core Overview 16-1
Performance and Resource Utilization 16-2
Functional Description 16-3
Interfaces 16-3
Avalon-ST Interfaces 16-3
Avalon-MM Interfaces 16-4
Operation 16-4
Instantiating the Core in SOPC Builder 16-5
Device Support 16-5
Software Programming Model 16-5
HAL System Library Support 16-5
Register Map 16-6
Referenced Documents 16-6
Document Revision History 16-6

Section III. Transport and Communication

Chapter 17. Avalon Streaming Channel Multiplexer and Demultiplexer Cores
Core Overview 17-1
Resource Usage and Performance 17-1

Multiplexer 17-2
Functional Description 17-2
Input Interfaces 17-3
Output Interface 17-3
Instantiating the Multiplexer in SOPC Builder 17-3
Functional Parameters 17-3
Output Interface 17-4
Demultiplexer 17-4
Functional Description 17-4
Input Interface 17-5
Output Interfaces 17-5
Instantiating the Demultiplexer in SOPC Builder 17-5
Functional Parameters 17-5
Input Interface 17-6
Device Support 17-6
Hardware Simulation Considerations 17-6
Software Programming Model 17-6
Document Revision History 17-7

Chapter 18. Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores
Core Overview 18-1
Functional Description 18-1
Interfaces 18-2
Operation—Avalon-ST Bytes to Packets Converter Core 18-2
Operation—Avalon-ST Packets to Bytes Converter Core 18-3
Instantiating the Core in SOPC Builder 18-3
Device Support 18-4
Referenced Documents 18-4
Document Revision History 18-4

Chapter 19. Avalon Packets to Transactions Converter Core
Core Overview 19-1
Functional Description 19-1
Interfaces 19-2
Operation 19-2
Packet Formats 19-2
Supported Transactions 19-3
Malformed Packets 19-3
Instantiating the Core in SOPC Builder 19-4
Device Support 19-4
Referenced Documents 19-4
Document Revision History 19-4

Chapter 20. Avalon-ST Round Robin Scheduler Core
Core Overview 20-1
Performance and Resource Utilization 20-1
Functional Description 20-2
Interfaces 20-2
Almost-Full Status Interface 20-2
Request Interface 20-3
Operations 20-3
Instantiating the Core in SOPC Builder 20-4
Device Support 20-4

Document Revision History 20-4

Section IV. Peripherals

Chapter 21. Scatter-Gather DMA Controller Core

Core Overview 21-1
Example Systems 21-1
Comparison of SG-DMA Controller Core and DMA Controller Core 21-2
In This Chapter 21-2
Resource Usage and Performance 21-3
Functional Description 21-3
Functional Blocks and Configurations 21-4
Descriptor Processor 21-4
DMA Read Block 21-4
DMA Write Block 21-4
Memory-to-Memory Configuration 21-5
Memory-to-Stream Configuration 21-5
Stream-to-Memory Configuration 21-6
DMA Descriptors 21-6
Descriptor Processing 21-7
Building and Updating Descriptor List 21-8
Error Conditions 21-8
Device Support 21-9
Instantiating the Core in SOPC Builder 21-9
Simulation Considerations 21-10
Software Programming Model 21-10
HAL System Library Support 21-10
Software Files 21-10
Register Maps 21-10
DMA Descriptors 21-13
Timeouts 21-14
Programming with SG-DMA Controller 21-15
Data Structure 21-15
SG-DMA API 21-16
alt_avalon_sgdma_do_async_transfer() 21-17
alt_avalon_sgdma_do_sync_transfer() 21-17
alt_avalon_sgdma_construct_mem_to_mem_desc() 21-18
alt_avalon_sgdma_construct_stream_to_mem_desc() 21-19
alt_avalon_sgdma_construct_mem_to_stream_desc() 21-20
alt_avalon_sgdma_check_descriptor_status() 21-21
alt_avalon_sgdma_register_callback() 21-21
alt_avalon_sgdma_start() 21-21
alt_avalon_sgdma_stop() 21-22
alt_avalon_sgdma_open() 21-22
Referenced Documents 21-23
Document Revision History 21-23

Chapter 22. DMA Controller Core

Core Overview 22-1
Functional Description 22-1
Setting Up DMA Transactions 22-2
The Master Read and Write Ports 22-3
Addressing and Address Incrementing 22-3

Instantiating the Core in SOPC Builder 22-4
DMA Parameters (Basic) 22-4
Transfer Size 22-4
Burst Transactions 22-4
FIFO Implementation 22-4
Advanced Options 22-5
Allowed Transactions 22-5
Device Support 22-5
Software Programming Model 22-5
HAL System Library Support 22-5
ioctl() Operations 22-6
Limitations 22-6
Software Files 22-6
Register Map 22-7
status Register 22-7
readaddress Register 22-8
writeaddress Register 22-8
length Register 22-8
control Register 22-8
Interrupt Behavior 22-10
Referenced Documents 22-10
Document Revision History 22-10

Chapter 23. Video Sync Generator and Pixel Converter Cores

Core Overview 23-1
Video Sync Generator 23-2
Functional Description 23-2
Instantiating the Core in SOPC Builder 23-3
Signals 23-4
Timing Diagrams 23-4
Pixel Converter 23-5
Functional Description 23-5
Instantiating the Core in SOPC Builder 23-5
Signals 23-6
Device Support 23-6
Hardware Simulation Considerations 23-6
Referenced Documents 23-6
Document Revision History 23-7

Chapter 24. Interval Timer Core

Core Overview 24-1
Functional Description 24-1
Avalon-MM Slave Interface 24-2
Device Support 24-2
Instantiating the Core in SOPC Builder 24-3
Timeout Period 24-3
Counter Size 24-3
Hardware Options 24-3
Register Options 24-4
Output Signal Options 24-4
Configuring the Timer as a Watchdog Timer 24-4
Software Programming Model 24-5
HAL System Library Support 24-5

System Clock Driver . . . 24-5
Timestamp Driver . . . 24-5
Limitations . . . 24-6
Software Files . . . 24-6
Register Map . . . 24-6
status Register . . . 24-7
control Register . . . 24-7
period_n Registers . . . 24-8
snap_n Registers . . . 24-8
Interrupt Behavior . . . 24-8
Referenced Documents . . . 24-8
Document Revision History . . . 24-9

Chapter 25. System ID Core

Core Overview . . . 25-1
Functional Description . . . 25-1
Device Support . . . 25-2
Instantiating the Core in SOPC Builder . . . 25-2
Software Programming Model . . . 25-2
alt_avalon_sysid_test() . . . 25-2
Document Revision History . . . 25-3

Chapter 26. Mutex Core

Core Overview . . . 26-1
Functional Description . . . 26-1
Device Support . . . 26-2
Instantiating the Core in SOPC Builder . . . 26-2
Software Programming Model . . . 26-2
Software Files . . . 26-2
Hardware Access Routines . . . 26-3
Mutex API . . . 26-4
altera_avalon_mutex_is_mine() . . . 26-4
altera_avalon_mutex_first_lock() . . . 26-4
altera_avalon_mutex_lock() . . . 26-4
altera_avalon_mutex_open() . . . 26-5
altera_avalon_mutex_trylock() . . . 26-5
altera_avalon_mutex_unlock() . . . 26-5
Document Revision History . . . 26-6

Chapter 27. Mailbox Core

Core Overview . . . 27-1
Functional Description . . . 27-1
Device Support . . . 27-2
Instantiating the Core in SOPC Builder . . . 27-2
Software Programming Model . . . 27-3
Software Files . . . 27-3
Programming with the Mailbox Core . . . 27-3
Mailbox API . . . 27-5
altera_avalon_mailbox_close() . . . 27-5
altera_avalon_mailbox_get() . . . 27-5
altera_avalon_mailbox_open() . . . 27-5
altera_avalon_mailbox_pend() . . . 27-6
altera_avalon_mailbox_post() . . . 27-6

Document Revision History 27-6

Section V. Test and Debug Peripherals

Chapter 28. Cyclone III Remote Update Controller Core

Core Overview 28-1
Functional Description 28-1
Avalon-MM Slave Interface and Registers 28-2
Device Support 28-2
Instantiating the Core in SOPC Builder 28-2
Software Programming Model 28-3
Setting the Configuration Offset 28-3
Shifting the Configuration Offset Value 28-3
Setting up the Watchdog Timer 28-3
Triggering a Reconfiguration 28-4
Code Example 28-5
Related Documentation 28-6
Document Revision History 28-6

Chapter 29. Performance Counter Core

Core Overview 29-1
Functional Description 29-2
Section Counters 29-2
Global Counter 29-2
Register Map 29-2
System Reset Considerations 29-3
Device and Tools Support 29-3
Instantiating the Core in SOPC Builder 29-3
Define Counters 29-3
Multiple Clock Domain Considerations 29-3
Hardware Simulation Considerations 29-4
Software Programming Model 29-4
Software Files 29-4
Using the Performance Counter 29-4
API Summary 29-4
Startup 29-5
Global Counter Usage 29-5
Section Counter Usage 29-5
Viewing Counter Values 29-5
Interrupt Behavior 29-6
Performance Counter API 29-6
PERF_RESET() 29-7
PERF_START_MEASURING() 29-7
PERF_STOP_MEASURING() 29-7
PERF_BEGIN() 29-7
PERF_END() 29-8
perf_print_formatted_report() 29-9
perf_get_total_time() 29-9
perf_get_section_time() 29-10
perf_get_num_starts() 29-10
alt_get_cpu_freq() 29-10
Referenced Documents 29-11
Document Revision History 29-11

Chapter 30. Avalon Streaming Test Pattern Generator and Checker Cores

Core Overview 30-1
Resource Utilization and Performance 30-1
Test Pattern Generator 30-3
Functional Description 30-3
Command Interface 30-3
Control and Status Interface 30-4
Output Interface 30-4
Instantiating the Test Pattern Generator in SOPC Builder 30-4
Functional Parameter 30-4
Output Interface 30-4
Test Pattern Checker 30-5
Functional Description 30-5
Input Interface 30-5
Control and Status Interface 30-6
Instantiating the Test Pattern Checker in SOPC Builder 30-6
Functional Parameter 30-6
Input Parameters 30-6
Device Support 30-6
Hardware Simulation Considerations 30-6
Software Programming Model 30-7
HAL System Library Support 30-7
Software Files 30-7
Register Maps 30-8
Test Pattern Generator Control and Status Registers 30-8
Test Pattern Generator Command Registers 30-9
Test Pattern Checker Control and Status Registers 30-10
Test Pattern Generator API 30-12
data_source_reset() 30-12
data_source_init() 30-12
data_source_get_id() 30-12
data_source_get_supports_packets() 30-13
data_source_get_num_channels() 30-13
data_source_get_symbols_per_cycle() 30-13
data_source_set_enable() 30-13
data_source_get_enable() 30-14
data_source_set_throttle() 30-14
data_source_get_throttle() 30-14
data_source_is_busy() 30-14
data_source_fill_level() 30-15
data_source_send_data() 30-15
Test Pattern Checker API 30-16
data_sink_reset() 30-16
data_sink_init() 30-16
data_sink_get_id() 30-16
data_sink_get_supports_packets() 30-16
data_sink_get_num_channels() 30-17
data_sink_get_symbols_per_cycle() 30-17
data_sink_set enable() 30-17
data_sink_get_enable() 30-17
data_sink_set_throttle() 30-18
data_sink_get_throttle() 30-18
data_sink_get_packet_count() 30-18
data_sink_get_symbol_count() 30-18

data_sink_get_error_count() 30-19
data_sink_get_exception() 30-19
data_sink_exception_is_exception() 30-19
data_sink_exception_has_data_error() 30-19
data_sink_exception_has_missing_sop() 30-20
data_sink_exception_has_missing_eop() 30-20
data_sink_exception_signalled_error() 30-20
data_sink_exception_channel() 30-20
Document Revision History 30-21

Section VI. Clock Control Peripherals

Chapter 31. PLL Cores

Core Overview 31-1
Functional Description 31-1
ALTPLL Megafunction 31-2
Clock Outputs 31-2
PLL Status and Control Signals 31-3
System Reset Considerations 31-3
Device Support 31-3
Instantiating the Cores in SOPC Builder 31-3
Instantiating the Avalon ALTPLL Core 31-3
Instantiating the PLL Core 31-3
PLL Settings Page 31-4
Interface Page 31-4
Finish 31-4
Hardware Simulation Considerations 31-5
Register Definitions and Bit List 31-5
Status Register 31-5
Control Register 31-6
Referenced Documents 31-6
Document Revision History 31-6

Additional Information

About this Handbook Info-1
How to Contact Altera Info-1
Third-Party Software Product Information Info-1
Typographic Conventions Info-2

Chapter Revision Dates

The chapters in this book, *Quartus II Handbook Version 9.0 Volume 5: Embedded Peripherals*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1 SDRAM Controller Core
Revised: *March 2009*
Part Number: *NII51005-9.0.0*

Chapter 2 CompactFlash Core
Revised: *March 2009*
Part Number: *QII55005-9.0.0*

Chapter 3 Common Flash Interface Controller Core
Revised: *March 2009*
Part Number: *NII51013-9.0.0*

Chapter 4 EPCS Device Controller Core
Revised: *March 2009*
Part Number: *NII51012-9.0.0*

Chapter 5 JTAG UART Core
Revised: *March 2009*
Part Number: *NII51009-9.0.0*

Chapter 6 UART Core
Revised: *March 2009*
Part Number: *NII51010-9.0.0*

Chapter 7 SPI Core
Revised: *March 2009*
Part Number: *NII51011-9.0.0*

Chapter 8 Optrex 16207 LCD Controller Core
Revised: *March 2009*
Part Number: *NII51019-9.0.0*

Chapter 9 PIO Core
Revised: *March 2009*
Part Number: *NII51007-9.0.0*

Chapter 10 Avalon-ST JTAG Interface Core
Revised: *March 2009*
Part Number: *QII55008-9.0.0*

Chapter 11 Avalon-ST Serial Peripheral Interface Core
Revised: *March 2009*
Part Number: *QII55009-9.0.0*

Chapter 12 SPI Slave/JTAG to Avalon Master Bridge Cores
Revised: *March 2009*
Part Number: *QII55011-9.0.0*

Chapter 13 PCI Lite Core
Revised: *March 2009*
Part Number: *QII55010-9.0.0*

Chapter 14 Avalon-ST Single Clock and Dual Clock FIFO Cores
Revised: *March 2009*
Part Number: *QII55014-9.0.0*

Chapter 15 On-Chip FIFO Memory Core
Revised: *March 2009*
Part Number: *QII55002-9.0.0*

Chapter 16 Avalon-ST Multi-Channel Shared Memory FIFO Core
Revised: *March 2009*
Part Number: *QII55015-9.0.0*

Chapter 17 Avalon Streaming Channel Multiplexer and Demultiplexer Cores
Revised: *March 2009*
Part Number: *QII55004-9.0.0*

Chapter 18 Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores
Revised: *March 2009*
Part Number: *QII55012-9.0.0*

Chapter 19 Avalon Packets to Transactions Converter Core
Revised: *March 2009*
Part Number: *QII55013-9.0.0*

Chapter 20 Avalon-ST Round Robin Scheduler Core
Revised: *March 2009*
Part Number: *QII55016-9.0.0*

Chapter 21 Scatter-Gather DMA Controller Core
Revised: *March 2009*
Part Number: *QII55003-9.0.0*

Chapter 22 DMA Controller Core
Revised: *March 2009*
Part Number: *NII51006-9.0.0*

Chapter 23 Video Sync Generator and Pixel Converter Cores
Revised: *March 2009*
Part Number: *QII55006-9.0.0*

Chapter 24 Interval Timer Core
Revised: *March 2009*
Part Number: *NII51008-9.0.0*

© March 2009 Altera Corporation

Chapter 25 System ID Core
Revised: *March 2009*
Part Number: *NII51014-9.0.0*

Chapter 26 Mutex Core
Revised: *March 2009*
Part Number: *NII51020-9.0.0*

Chapter 27 Mailbox Core
Revised: *March 2009*
Part Number: *NII53001-9.0.0*

Chapter 28 Cyclone III Remote Update Controller Core
Revised: *March 2009*
Part Number: *QII55005-9.0.0*

Chapter 29 Performance Counter Core
Revised: *March 2009*
Part Number: *QII55001-9.0.0*

Chapter 30 Avalon Streaming Test Pattern Generator and Checker Cores
Revised: *March 2009*
Part Number: *QII55007-9.0.0*

Chapter 31 PLL Cores
Revised: *March 2009*
Part Number: *NII53002-9.0.0*

Section I. Off-Chip Interface Peripherals

This section describes the interfaces to off-chip devices provided for SOPC Builder systems.

This section includes the following chapters:

- Chapter 1, SDRAM Controller Core
- Chapter 2, CompactFlash Core
- Chapter 3, Common Flash Interface Controller Core
- Chapter 4, EPCS Device Controller Core
- Chapter 5, JTAG UART Core
- Chapter 6, UART Core
- Chapter 7, SPI Core
- Chapter 8, Optrex 16207 LCD Controller Core
- Chapter 9, PIO Core
- Chapter 10, Avalon-ST JTAG Interface Core
- Chapter 11, Avalon-ST Serial Peripheral Interface Core
- Chapter 12, SPI Slave/JTAG to Avalon Master Bridge Cores
- Chapter 13, PCI Lite Core

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

1. SDRAM Controller Core

NII51005-9.0.0

Core Overview

The SDRAM controller core with Avalon® interface provides an Avalon Memory-Mapped (Avalon-MM) interface to off-chip SDRAM. The SDRAM controller allows designers to create custom systems in an Altera® device that connect easily to SDRAM chips. The SDRAM controller supports standard SDRAM as described in the PC100 specification.

SDRAM is commonly used in cost-sensitive applications requiring large amounts of volatile memory. While SDRAM is relatively inexpensive, control logic is required to perform refresh operations, open-row management, and other delays and command sequences. The SDRAM controller connects to one or more SDRAM chips, and handles all SDRAM protocol requirements. Internal to the device, the core presents an Avalon-MM slave port that appears as linear memory (flat address space) to Avalon-MM master peripherals.

The core can access SDRAM subsystems with various data widths (8, 16, 32, or 64 bits), various memory sizes, and multiple chip selects. The Avalon-MM interface is latency-aware, allowing read transfers to be pipelined. The core can optionally share its address and data buses with other off-chip Avalon-MM tri-state devices. This feature is valuable in systems that have limited I/O pins, yet must connect to multiple memory chips in addition to SDRAM.

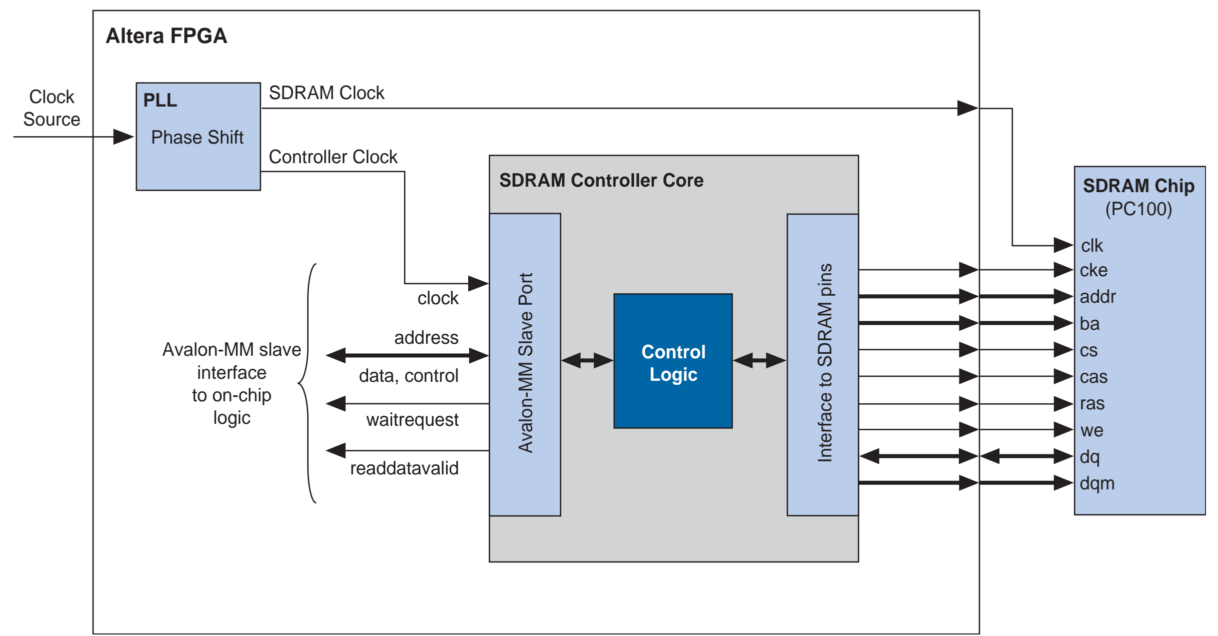
The SDRAM controller core with Avalon interface is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description" on page 1–2
- "Device Support" on page 1–5
- "Instantiating the Core in SOPC Builder" on page 1–5
- "Hardware Simulation Considerations" on page 1–7
- "Software Programming Model" on page 1–10
- "Clock, PLL and Timing Considerations" on page 1–10

Functional Description

Figure 1–1 shows a block diagram of the SDRAM controller core connected to an external SDRAM chip.

Figure 1–1. SDRAM Controller with Avalon Interface Block Diagram



The following sections describe the components of the SDRAM controller core in detail. All options are specified at system generation time, and cannot be changed at runtime.

Avalon-MM Interface

The Avalon-MM slave port is the user-visible part of the SDRAM controller core. The slave port presents a flat, contiguous memory space as large as the SDRAM chip(s). When accessing the slave port, the details of the PC100 SDRAM protocol are entirely transparent. The Avalon-MM interface behaves as a simple memory interface. There are no memory-mapped configuration registers.

The Avalon-MM slave port supports peripheral-controlled wait states for read and write transfers. The slave port stalls the transfer until it can present valid data. The slave port also supports read transfers with variable latency, enabling high-bandwidth, pipelined read transfers. When a master peripheral reads sequential addresses from the slave port, the first data returns after an initial period of latency. Subsequent reads can produce new data every clock cycle. However, data is not guaranteed to return every clock cycle, because the SDRAM controller must pause periodically to refresh the SDRAM.

For details about Avalon-MM transfer types, refer to the *Avalon Interface Specifications*.

Off-Chip SDRAM Interface

The interface to the external SDRAM chip presents the signals defined by the PC100 standard. These signals must be connected externally to the SDRAM chip(s) through I/O pins on the Altera device.

Signal Timing and Electrical Characteristics

The timing and sequencing of signals depends on the configuration of the core. The hardware designer configures the core to match the SDRAM chip chosen for the system. See "Instantiating the Core in SOPC Builder" on page 1–5 for details. The electrical characteristics of the device pins depend on both the target device family and the assignments made in the Quartus® II software. Some device families support a wider range of electrical standards, and therefore are capable of interfacing with a greater variety of SDRAM chips. For details, refer to the device handbook for the target device family.

Synchronizing Clock and Data Signals

The clock for the SDRAM chip (SDRAM clock) must be driven at the same frequency as the clock for the Avalon-MM interface on the SDRAM controller (controller clock). As in all synchronous designs, you must ensure that address, data, and control signals at the SDRAM pins are stable when a clock edge arrives. As shown in Figure 1–1, you can use an on-chip phase-locked loop (PLL) to alleviate clock skew between the SDRAM controller core and the SDRAM chip. At lower clock speeds, the PLL might not be necessary. At higher clock rates, a PLL is necessary to ensure that the SDRAM clock toggles only when signals are stable on the pins. The PLL block is not part of the SDRAM controller core. If a PLL is necessary, you must instantiate it manually. You can instantiate the PLL core interface, which is an SOPC Builder component, or instantiate an ALTPLL megafunction outside the SOPC Builder system module.

If you use a PLL, you must tune the PLL to introduce a clock phase shift so that SDRAM clock edges arrive after synchronous signals have stabilized. See "Clock, PLL and Timing Considerations" on page 1–10 for details.

For more information about instantiating a PLL in your SOPC Builder system, refer to *PLL Core* chapter in volume 5 of the *Quartus II Handbook*. The Nios® II development tools provide example hardware designs that use the SDRAM controller core in conjunction with a PLL, which you can use as a reference for your custom designs. The Nios II development tools are available free for download from www.altera.com.

Clock Enable (CKE) Not Supported

The SDRAM controller does not support clock-disable modes. The SDRAM controller permanently asserts the CKE signal on the SDRAM.

Sharing Pins with Other Avalon-MM Tri-State Devices

If an Avalon-MM tri-state bridge is present in the SOPC Builder system, the SDRAM controller core can share pins with the existing tri-state bridge. In this case, the core's `addr`, `dq` (data) and `dqm` (byte-enable) pins are shared with other devices connected to the Avalon-MM tri-state bridge. This feature conserves I/O pins, which is valuable in systems that have multiple external memory chips (for example, flash, SRAM, and SDRAM), but too few pins to dedicate to the SDRAM chip. See "Performance Considerations" for details about how pin sharing affects performance.

The SDRAM addresses must connect all address bits regardless of the size of the word so that the low-order address bits on the tri-state bridge align with the low-order address bits on the memory device. The Avalon-MM tristate address signal always presents a byte address. It is not possible to drop A0 of the tri-state bridge for memories when the smallest access size is 16 bits or A0-A1 of the tri-state bridge when the smallest access size is 32 bits.

Board Layout and Pinout Considerations

When making decisions about the board layout and device pinout, try to minimize the skew between the SDRAM signals. For example, when assigning the device pinout, group the SDRAM signals, including the SDRAM clock output, physically close together. Also, you can use the **Fast Input Register** and **Fast Output Register** logic options in the Quartus II software. These logic options place registers for the SDRAM signals in the I/O cells. Signals driven from registers in I/O cells have similar timing characteristics, such as t_{CO}, t_{SU}, and t_{H}.

Performance Considerations

Under optimal conditions, the SDRAM controller core’s bandwidth approaches one word per clock cycle. However, because of the overhead associated with refreshing the SDRAM, it is impossible to reach one word per clock cycle. Other factors affect the core’s performance, as described in the following sections.

Open Row Management

SDRAM chips are arranged as multiple banks of memory, in which each bank is capable of independent open-row address management. The SDRAM controller core takes advantage of open-row management for a single bank. Continuous reads or writes within the same row and bank operate at rates approaching one word per clock. Applications that frequently access different destination banks require extra management cycles to open and close rows.

Sharing Data and Address Pins

When the controller shares pins with other tri-state devices, average access time usually increases and bandwidth decreases. When access to the tri-state bridge is granted to other devices, the SDRAM incurs overhead to open and close rows. Furthermore, the SDRAM controller has to wait several clock cycles before it is granted access again.

To maximize bandwidth, the SDRAM controller automatically maintains control of the tri-state bridge as long as back-to-back read or write transactions continue within the same row and bank.

This behavior may degrade the average access time for other devices sharing the Avalon-MM tri-state bridge.

The SDRAM controller closes an open row whenever there is a break in back-to-back transactions, or whenever a refresh transaction is required. As a result:

- The controller cannot permanently block access to other devices sharing the tri-state bridge.

- The controller is guaranteed not to violate the SDRAM's row open time limit.

Hardware Design and Target Device

The target device affects the maximum achievable clock frequency of a hardware design. Certain device families achieve higher f_{MAX} performance than other families. Furthermore, within a device family, faster speed grades achieve higher performance. The SDRAM controller core can achieve 100 MHz in Altera's high-performance device families, such as Stratix® series. However, the core might not achieve 100 MHz performance in all Altera device families.

The f_{MAX} performance also depends on the SOPC Builder system design. The SDRAM controller clock can also drive other logic in the system module, which might affect the maximum achievable frequency. For the SDRAM controller core to achieve f_{MAX} performance of 100 MHz, all components driven by the same clock must be designed for a 100 MHz clock rate, and timing analysis in the Quartus II software must verify that the overall hardware design is capable of 100 MHz operation.

Device Support

The SDRAM Controller with Avalon interface core supports all Altera device families. Different device families support different I/O standards, which may affect the ability of the core to interface to certain SDRAM chips. For details about supported I/O types, refer to the device handbook for the target device family.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the SDRAM controller in SOPC Builder to specify hardware and simulation features. The SDRAM controller MegaWizard has two pages: **Memory Profile** and **Timing**. This section describes the options available on each page.

The **Presets** list offers several pre-defined SDRAM configurations as a convenience. If the SDRAM subsystem on the target board matches one of the preset configurations, you can configure the SDRAM controller core easily by selecting the appropriate preset value. The following preset configurations are defined:

- Micron MT8LSDT1664HG module
- Four SDR100 8 MByte × 16 chips
- Single Micron MT48LC2M32B2-7 chip
- Single Micron MT48LC4M32B2-7 chip
- Single NEC D4564163-A80 chip (64 MByte × 16)
- Single Alliance AS4LC1M16S1-10 chip
- Single Alliance AS4LC2M8S0-10 chip

Selecting a preset configuration automatically changes values on the **Memory Profile** and **Timing** tabs to match the specific configuration. Altering a configuration setting on any page changes the **Preset** value to **custom**.

Memory Profile Page

The **Memory Profile** page allows you to specify the structure of the SDRAM subsystem such as address and data bus widths, the number of chip select signals, and the number of banks. Table 1–1 lists the settings available on the **Memory Profile** page.

Table 1–1. Memory Profile Page Settings

Settings		Allowed Values	Default Values	Description
Data Width		8, 16, 32, 64	32	SDRAM data bus width. This value determines the width of the `dq` bus (data) and the `dqm` bus (byte-enable).
Architecture Settings	Chip Selects	1, 2, 4, 8	1	Number of independent chip selects in the SDRAM subsystem. By using multiple chip selects, the SDRAM controller can combine multiple SDRAM chips into one memory subsystem.
	Banks	2, 4	4	Number of SDRAM banks. This value determines the width of the `ba` bus (bank address) that connects to the SDRAM. The correct value is provided in the data sheet for the target SDRAM.
Address Width Settings	Row	11, 12, 13, 14	12	Number of row address bits. This value determines the width of the `addr` bus. The Row and Column values depend on the geometry of the chosen SDRAM. For example, an SDRAM organized as 4096 (2^{12}) rows by 512 columns has a Row value of 12.
	Column	>= 8, and less than Row value	8	Number of column address bits. For example, the SDRAM organized as 4096 rows by 512 (2^9) columns has a Column value of 9.
Share pins via tri-state bridge dq/dqm/addr I/O pins		On, Off	Off	When set to No, all pins are dedicated to the SDRAM chip. When set to Yes, the `addr`, `dq`, and `dqm` pins can be shared with a tristate bridge in the system. In this case, select the appropriate tristate bridge from the pull-down menu.
Include a functional memory model in the system testbench		On, Off	On	When on, SOPC Builder creates a functional simulation model for the SDRAM chip. This default memory model accelerates the process of creating and verifying systems that use the SDRAM controller. See “Hardware Simulation Considerations” on page 1–7.

Based on the settings entered on the **Memory Profile** page, the wizard displays the expected memory capacity of the SDRAM subsystem in units of megabytes, megabits, and number of addressable words. Compare these expected values to the actual size of the chosen SDRAM to verify that the settings are correct.

Timing Page

The **Timing** page allows designers to enter the timing specifications of the SDRAM chip(s) used. The correct values are available in the manufacturer's data sheet for the target SDRAM. Table 1–2 lists the settings available on the **Timing** page.

Table 1–2. Timing Page Settings

Settings	Allowed Values	Default Value	Description
CAS latency	1, 2, 3	3	Latency (in clock cycles) from a read command to data out.
Initialization refresh cycles	1–8	2	This value specifies how many refresh cycles the SDRAM controller performs as part of the initialization sequence after reset.
Issue one refresh command every	—	15.625 µs	This value specifies how often the SDRAM controller refreshes the SDRAM. A typical SDRAM requires 4,096 refresh commands every 64 ms, which can be achieved by issuing one refresh command every 64 ms / 4,096 = 15.625 µs.
Delay after power up, before initialization	—	100 µs	The delay from stable clock and power to SDRAM initialization.
Duration of refresh command (t_rfc)	—	70 ns	Auto Refresh period.
Duration of precharge command (t_rp)	—	20 ns	Precharge command period.
ACTIVE to READ or WRITE delay (t_rcd)	—	20 ns	ACTIVE to READ or WRITE delay.
Access time (t_ac)	—	17 ns	Access time from clock edge. This value may depend on CAS latency.
Write recovery time (t_wr, No auto precharge)	—	14 ns	Write recovery if explicit precharge commands are issued. This SDRAM controller always issues explicit precharge commands.

Regardless of the exact timing values you specify, the actual timing achieved for each parameter is an integer multiple of the Avalon clock period. For the **Issue one refresh command every** parameter, the actual timing is the greatest number of clock cycles that does not exceed the target value. For all other parameters, the actual timing is the smallest number of clock ticks that provides a value greater than or equal to the target value.

Hardware Simulation Considerations

This section discusses considerations for simulating systems with SDRAM. Three major components are required for simulation:

- A simulation model for the SDRAM controller.
- A simulation model for the SDRAM chip(s), also called the memory model.
- A simulation testbench that wires the memory model to the SDRAM controller pins.

Some or all of these components are generated by SOPC Builder at system generation time.

SDRAM Controller Simulation Model

The SDRAM controller design files generated by SOPC Builder are suitable for both synthesis and simulation. Some simulation features are implemented in the HDL using "translate on/off" synthesis directives that make certain sections of HDL code invisible to the synthesis tool.

The simulation features are implemented primarily for easy simulation of Nios and Nios II processor systems using the ModelSim® simulator. The SDRAM controller simulation model is not ModelSim specific. However, minor changes may be required to make the model work with other simulators.

If you change the simulation directives to create a custom simulation flow, be aware that SOPC Builder overwrites existing files during system generation. Take precautions to ensure your changes are not overwritten.

Refer to *AN 351: Simulating Nios II Processor Designs* for a demonstration of simulation of the SDRAM controller in the context of Nios II embedded processor systems.

SDRAM Memory Model

This section describes the two options for simulating a memory model of the SDRAM chip(s).

Using the Generic Memory Model

If the **Include a functional memory model the system testbench** option is enabled at system generation, SOPC Builder generates an HDL simulation model for the SDRAM memory. In the auto-generated system testbench, SOPC Builder automatically wires this memory model to the SDRAM controller pins.

Using the automatic memory model and testbench accelerates the process of creating and verifying systems that use the SDRAM controller. However, the memory model is a generic functional model that does not reflect the true timing or functionality of real SDRAM chips. The generic model is always structured as a single, monolithic block of memory. For example, even for a system that combines two SDRAM chips, the generic memory model is implemented as a single entity.

Using the SDRAM Manufacturer's Memory Model

If the **Include a functional memory model the system testbench** option is not enabled, you are responsible for obtaining a memory model from the SDRAM manufacturer, and manually wiring the model to the SDRAM controller pins in the system testbench.

Example Configurations

The following examples show how to connect the SDRAM controller outputs to an SDRAM chip or chips. The bus labeled `ctl` is an aggregate of the remaining signals, such as `cas_n`, `ras_n`, `cke` and `we_n`.

Figure 1–2 shows a single 128-Mbit SDRAM chip with 32-bit data. The address, data, and control signals are wired directly from the controller to the chip. The result is a 128-Mbit (16-Mbyte) memory space.

Figure 1–2. Single 128-Mbit SDRAM Chip with 32-Bit Data

Figure 1–3 shows two 64-Mbit SDRAM chips, each with 16-bit data. The address and control signals connect in parallel to both chips. The chips share the chipselect (`cs_n`) signal. Each chip provides half of the 32-bit data bus. The result is a logical 128-Mbit (16-Mbyte) 32-bit data memory.

Figure 1–3. Two 64-MBit SDRAM Chips Each with 16-Bit Data

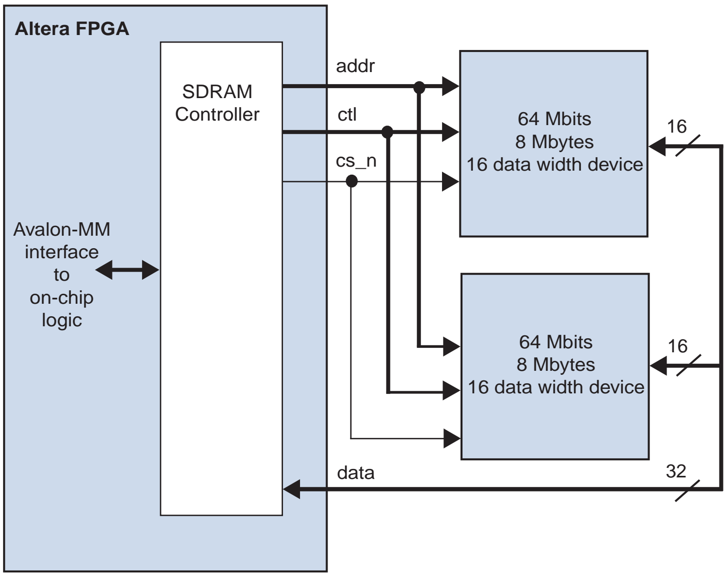
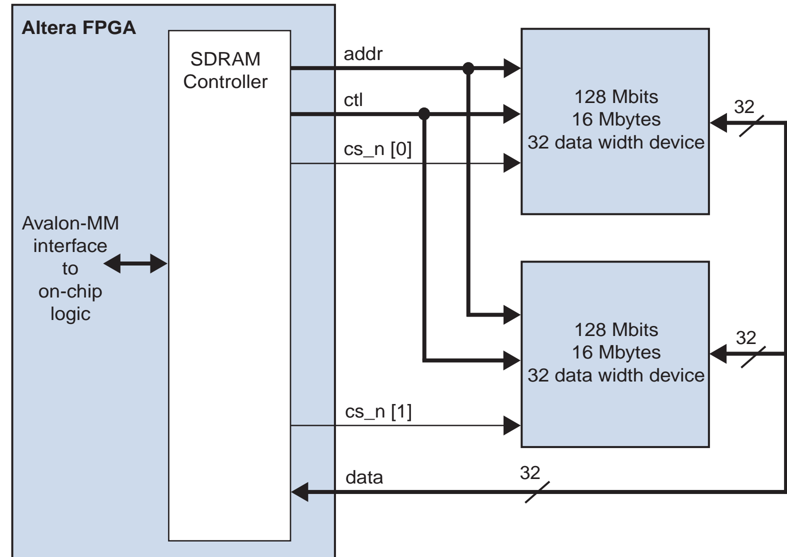


Figure 1–4 shows two 128-Mbit SDRAM chips, each with 32-bit data. The address, data, and control signals connect in parallel to the two chips. The chipselect bus (`cs_n[1:0]`) determines which chip is selected. The result is a logical 256-Mbit 32-bit wide memory.

Figure 1–4. Two 128-Mbit SDRAM Chips Each with 32-Bit Data



Software Programming Model

The SDRAM controller behaves like simple memory when accessed via the Avalon-MM interface. There are no software-configurable settings and no memory-mapped registers. No software driver routines are required for a processor to access the SDRAM controller.

Clock, PLL and Timing Considerations

This section describes issues related to synchronizing signals from the SDRAM controller core with the clock that drives the SDRAM chip. During SDRAM transactions, the address, data, and control signals are valid at the SDRAM pins for a window of time, during which the SDRAM clock must toggle to capture the correct values. At slower clock frequencies, the clock naturally falls within the valid window. At higher frequencies, you must compensate the SDRAM clock to align with the valid window.

Determine when the valid window occurs either by calculation or by analyzing the SDRAM pins with an oscilloscope. Then use a PLL to adjust the phase of the SDRAM clock so that edges occur in the middle of the valid window. Tuning the PLL might require trial-and-error effort to align the phase shift to the properties of your target board.

For details about the PLL circuitry in your target device, refer to the appropriate device family handbook. For details about configuring the PLLs in Altera devices, refer to the *ALTPLL Megafunction User Guide*.

Factors Affecting SDRAM Timing

The location and duration of the window depends on several factors:

- Timing parameters of the device and SDRAM I/O pins — I/O timing parameters vary based on device family and speed grade.
- Pin location on the device — I/O pins connected to row routing have different timing than pins connected to column routing.
- Logic options used during the Quartus II compilation — Logic options such as the **Fast Input Register** and **Fast Output Register** logic affect the design fit. The location of logic and registers inside the device affects the propagation delays of signals to the I/O pins.
- SDRAM CAS latency

As a result, the valid window timing is different for different combinations of FPGA and SDRAM devices. The window depends on the Quartus II software fitting results and pin assignments.

Symptoms of an Untuned PLL

Detecting when the PLL is not tuned correctly might be difficult. Data transfers to or from the SDRAM might not fail universally. For example, individual transfers to the SDRAM controller might succeed, whereas burst transfers fail. For processor-based systems, if software can perform read or write data to SDRAM, but cannot run when the code is located in SDRAM, the PLL is probably tuned incorrectly.

Estimating the Valid Signal Window

This section describes how to estimate the location and duration of the valid signal window using timing parameters provided in the SDRAM datasheet and the Quartus II software compilation report. After finding the window, tune the PLL so that SDRAM clock edges occur exactly in the middle of the window.

Calculating the window is a two-step process. First, determine by how much time the SDRAM clock can lag the controller clock, and then by how much time it can lead. After finding the maximum lag and lead values, calculate the midpoint between them.

These calculations provide an estimation only. The following delays can also affect proper PLL tuning, but are not accounted for by these calculations.

- Signal skew due to delays on the printed circuit board — These calculations assume zero skew.
- Delay from the PLL clock output nodes to destinations — These calculations assume that the delay from the PLL SDRAM-clock output-node to the pin is the same as the delay from the PLL controller-clock output-node to the clock inputs in the SDRAM controller. If these clock delays are significantly different, you must account for this phase shift in your window calculations.

Figure 1–5 shows how to calculate the maximum length of time that the SDRAM clock can lag the controller clock, and Figure 1–6 shows how to calculate the maximum lead. Lag is a negative time shift, relative to the controller clock, and lead is a positive time shift. The SDRAM clock can lag the controller clock by the lesser of the maximum lag for a read cycle or that for a write cycle. In other words, *Maximum Lag* = minimum(*Read Lag*, *Write Lag*). Similarly, the SDRAM clock can lead by the lesser of the maximum lead for a read cycle or for a write cycle. In other words, *Maximum Lead* = minimum(*Read Lead*, *Write Lead*).

Figure 1–5. Calculating the Maximum SDRAM Clock Lag

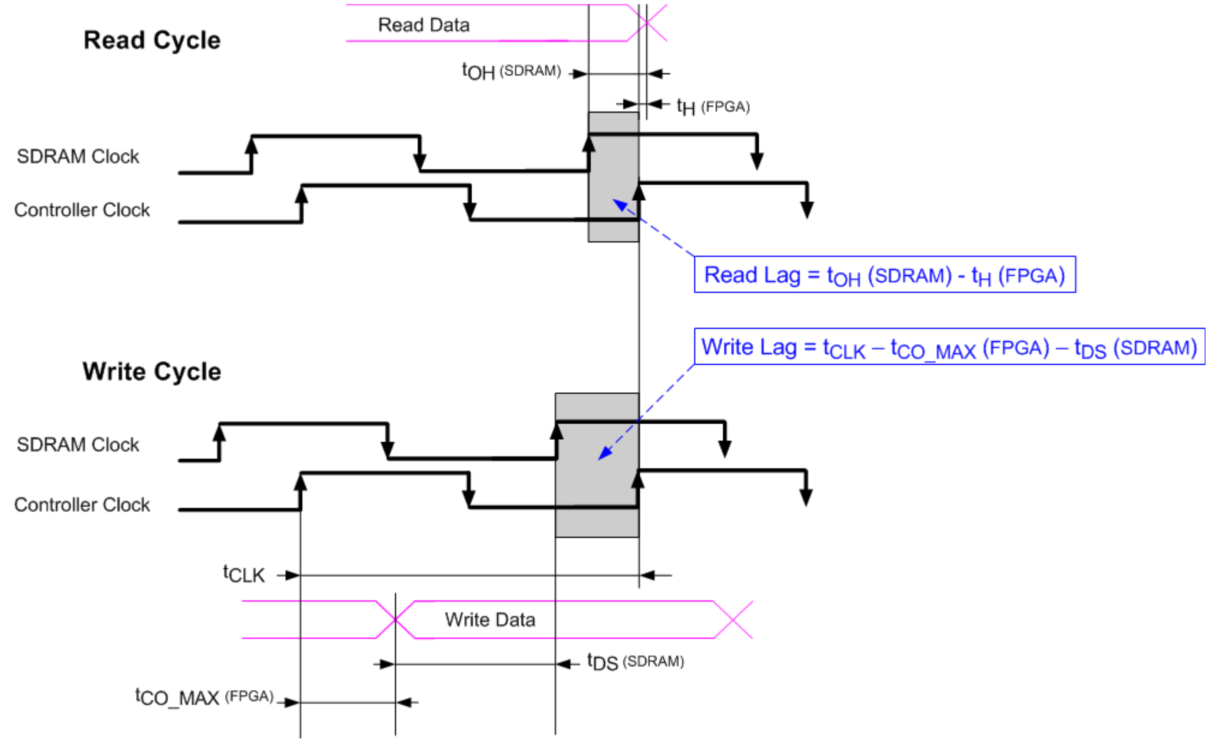
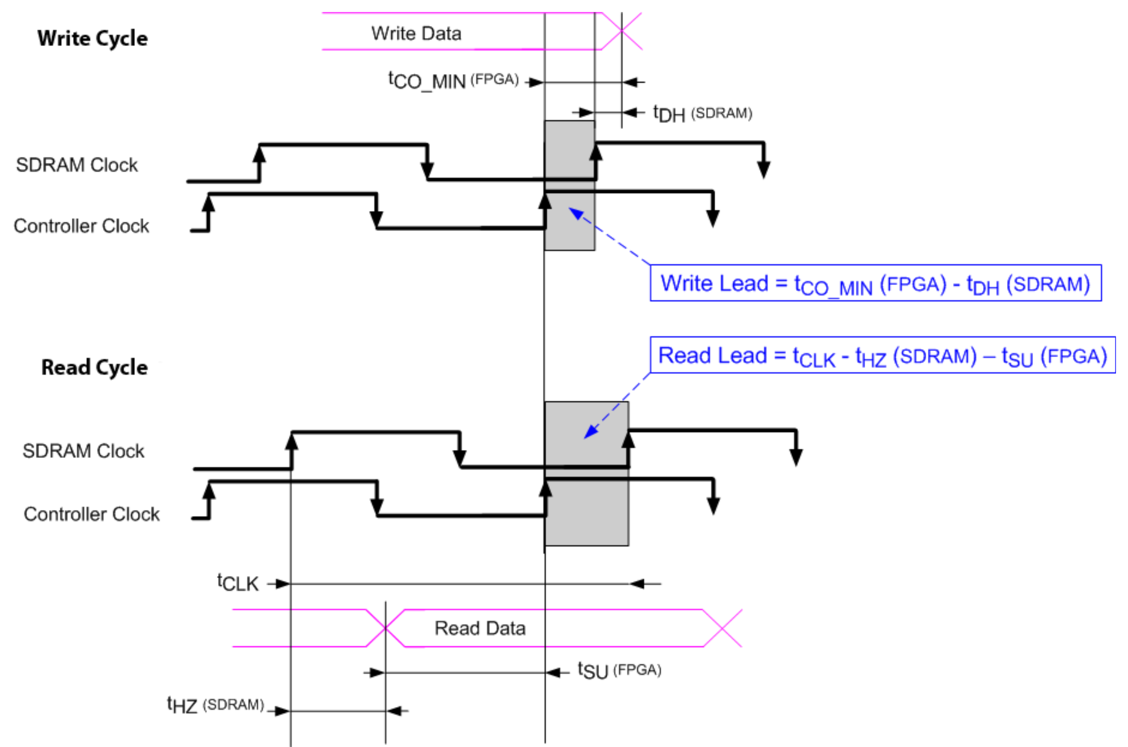


Figure 1–6. Calculating the Maximum SDRAM Clock Lead



Example Calculation

This section demonstrates a calculation of the signal window for a Micron MT48LC4M32B2-7 SDRAM chip and design targeting the Stratix II EP2S60F672C5 device. This example uses a CAS latency (CL) of 3 cycles, and a clock frequency of 50 MHz. All SDRAM signals on the device are registered in I/O cells, enabled with the **Fast Input Register** and **Fast Output Register** logic options in the Quartus II software.

Table 1–3 shows the relevant timing parameters excerpted from the MT48LC4M32B2 device datasheet.

Table 1–3. Timing Parameters for Micron MT48LC4M32B2 SDRAM Device (Part 1 of 2)

Parameter		Symbol	Value (ns) in -7 Speed Grade	
			Min.	**Max.**
Access time from CLK (pos. edge)	CL = 3	$t_{AC(3)}$	—	5.5
	CL = 2	$t_{AC(2)}$	—	8
	CL = 1	$t_{AC(1)}$	—	17
Address hold time		t_{AH}	1	—
Address setup time		t_{AS}	2	—
CLK high-level width		t_{CH}	2.75	—
CLK low-level width		t_{CL}	2.75	—

Table 1–3. Timing Parameters for Micron MT48LC4M32B2 SDRAM Device (Part 2 of 2)

Parameter		Symbol	Value (ns) in -7 Speed Grade	
			Min.	**Max.**
Clock cycle time	CL = 3	$t_{CK(3)}$	7	—
	CL = 2	$t_{CK(2)}$	10	—
	CL = 1	$t_{CK(1)}$	20	—
CKE hold time		t_{CKH}	1	—
CKE setup time		t_{CKS}	2	—
CS#, RAS#, CAS#, WE#, DQM hold time		t_{CMH}	1	—
CS#, RAS#, CAS#, WE#, DQM setup time		t_{CMS}	2	—
Data-in hold time		t_{DH}	1	
Data-in setup time		t_{DS}	2	
Data-out high-impedance time	CL = 3	$t_{HZ(3)}$		5.5
	CL = 2	$t_{HZ(2)}$	—	8
	CL = 1	$t_{HZ(1)}$	—	17
Data-out low-impedance time		t_{LZ}	1	—
Data-out hold time		t_{OH}	2.5	

Table 1–4 shows the relevant timing information, obtained from the Timing Analyzer section of the Quartus II Compilation Report. The values in the table are the maximum or minimum values among all device pins related to the SDRAM. The variance in timing between the SDRAM pins on the device is small (less than 100 ps) because the registers for these signals are placed in the I/O cell.

Table 1–4. FPGA I/O Timing Parameters

Parameter	Symbol	Value (ns)
Clock period	t_{CLK}	20
Minimum clock-to-output time	t_{CO_MIN}	2.399
Maximum clock-to-output time	t_{CO_MAX}	2.477
Maximum hold time after clock	t_{H_MAX}	–5.607
Maximum setup time before clock	t_{SU_MAX}	5.936

You must compile the design in the Quartus II software to obtain the I/O timing information for the design. Although Altera device family datasheets contain generic I/O timing information for each device, the Quartus II Compilation Report provides the most precise timing information for your specific design.

The timing values found in the compilation report can change, depending on fitting, pin location, and other Quartus II logic settings. When you recompile the design in the Quartus II software, verify that the I/O timing has not changed significantly.

© March 2009 Altera Corporation

The following examples illustrate the calculations from Figure 1–5 and Figure 1–6 using the values from Table 1–3 and Table 1–4.

The SDRAM clock can lag the controller clock by the lesser of *Read Lag* or *Write Lag*:

Read Lag = t_{OH}(SDRAM) – t_{H_MAX}(FPGA)

= 2.5 ns – (–5.607 ns) = 8.107 ns

or

Write Lag= t_{CLK} – t_{CO_MAX}(FPGA) – t_{DS}(SDRAM)

= 20 ns – 2.477 ns – 2 ns = 15.523 ns

The SDRAM clock can lead the controller clock by the lesser of *Read Lead* or *Write Lead*:

Read Lead= t_{CO_MIN}(FPGA) – t_{DH}(SDRAM)

= 2.399 ns – 1.0 ns = 1.399 ns

or

Write Lead= t_{CLK} – $t_{HZ(3)}$(SDRAM) – t_{SU_MAX}(FPGA)

= 20 ns – 5.5 ns – 5.936 ns = 8.564 ns

Therefore, for this example you can shift the phase of the SDRAM clock from –8.107 ns to 1.399 ns relative to the controller clock. Choosing a phase shift in the middle of this window results in the value (–8.107 + 1.399)/2 = –3.35 ns.

Referenced Documents

This chapter references the following documents:

- *ALTPLL Megafunction User Guide*
- *AN 351: Simulating Nios II Processor Designs*
- *Avalon Interface Specifications*
- *PLL Core* chapter in volume 5 of the *Quartus II Handbook*

Document Revision History

Table 1–5 shows the revision history for this chapter.

Table 1–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0.	No change from previous release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

© March 2009 Altera Corporation

2. CompactFlash Core

QII55005-9.0.0

Core Overview

The CompactFlash core allows you to connect SOPC Builder systems to CompactFlash storage cards in true IDE mode by providing an Avalon® Memory-Mapped (Avalon-MM) interface to the registers on the storage cards. The core supports PIO mode 0.

The CompactFlash core also provides an Avalon-MM slave interface which can be used by Avalon-MM master peripherals such as a Nios® II processor to communicate with the CompactFlash core and manage its operations.

The CompactFlash core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated systems.

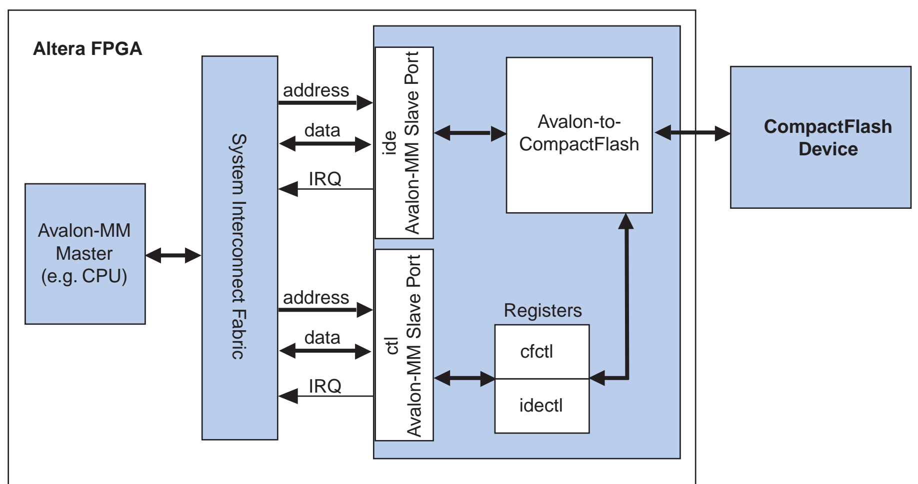
This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 2–2
- "Device Support" on page 2–3
- "Software Programming Model" on page 2–3

Functional Description

Figure 2–1 shows a block diagram of the CompactFlash core in a typical system configuration.

Figure 2–1. SOPC Builder System With a CompactFlash Core



As shown in Figure 2–1, the CompactFlash core provides two Avalon-MM slave interfaces: the `ide` slave port for accessing the registers on the CompactFlash device and the `ctl` slave port for accessing the core's internal registers. These registers can be used by Avalon-MM master peripherals such as a Nios II processor to control the operations of the CompactFlash core and to transfer data to and from the CompactFlash device.

You can set the CompactFlash core to generate two active-high interrupt requests (IRQs): one signals the insertion and removal of a CompactFlash device and the other passes interrupt signals from the CompactFlash device.

The CompactFlash core maps the Avalon-MM bus signals to the CompactFlash device with proper timing, thus allowing Avalon-MM master peripherals to directly access the registers on the CompactFlash device.

For more information, refer to the CF+ and CompactFlash specifications available at www.compactflash.org.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the CompactFlash core in SOPC Builder to add the core to a system. There are no user-configurable settings for this core.

Required Connections

Table 2–1 lists the required connections between the CompactFlash core and the CompactFlash device.

Table 2–1. Required Connections (Part 1 of 2)

CompactFlash Interface Signal Name	Pin Type	CompactFlash Pin Number
`addr[0]`	Output	20
`addr[1]`	Output	19
`addr[2]`	Output	18
`addr[3]`	Output	17
`addr[4]`	Output	16
`addr[5]`	Output	15
`addr[6]`	Output	14
`addr[7]`	Output	12
`addr[8]`	Output	11
`addr[9]`	Output	10
`addr[10]`	Output	8
`atasel_n`	Output	9
`cs_n[0]`	Output	7
`cs_n[1]`	Output	32
`data[0]`	Input/Output	21
`data[1]`	Input/Output	22

© March 2009 Altera Corporation

Table 2–1. Required Connections (Part 2 of 2)

CompactFlash Interface Signal Name	Pin Type	CompactFlash Pin Number
data[2]	Input/Output	23
data[3]	Input/Output	2
data[4]	Input/Output	3
data[5]	Input/Output	4
data[6]	Input/Output	5
data[7]	Input/Output	6
data[8]	Input/Output	47
data[9]	Input/Output	48
data[10]	Input/Output	49
data[11]	Input/Output	27
data[12]	Input/Output	28
data[13]	Input/Output	29
data[14]	Input/Output	30
data[15]	Input/Output	31
detect	Input	25 or 26
intrq	Input	37
iord_n	Output	34
iordy	Input	42
iowr_n	Output	35
power	Output	CompactFlash power controller, if present
reset_n	Output	41
rfu	Output	44
we_n	Output	46

Device Support

The CompactFlash interface core supports all Altera® device families.

Software Programming Model

This section describes the software programming model for the CompactFlash core.

HAL System Library Support

The Altera-provided HAL API functions include a device driver that you can use to initialize the CompactFlash core. To perform other operations, use the low-level macros provided. For more information on the macros, refer to the files listed in the section "Software Files" on page 2–4.

Software Files

The CompactFlash core provides the following software files. These files define the low-level access to the hardware. Application developers should not modify these files.

- **altera_avalon_cf_regs.h**—The header file that defines the core's register maps.
- **altera_avalon_cf.h, altera_avalon_cf.c**—The header and source code for the functions and variables required to integrate the driver into the HAL system library.

Register Maps

This section describes the register maps for the Avalon-MM slave interfaces.

Ide Registers

The `ide` port in the CompactFlash core allows you to access the IDE registers on a CompactFlash device. Table 2–2 shows the register map for the `ide` port.

Table 2–2. Ide Register Map

Offset	Register Names	
	Read Operation	Write Operation
0	`RD Data`	`WR Data`
1	`Error`	`Features`
2	`Sector Count`	`Sector Count`
3	`Sector No`	`Sector No`
4	`Cylinder Low`	`Cylinder Low`
5	`Cylinder High`	`Cylinder High`
6	`Select Card/Head`	`Select Card/Head`
7	`Status`	`Command`
14	`Alt Status`	`Device Control`

Ctl Registers

The `ctl` port in the CompactFlash core provides access to the registers which control the core's operation and interface. Table 2–3 shows the register map for the `ctl` port.

Table 2–3. Ctl Register Map

Offset	Register	Fields				
		31:4	3	2	1	0
0	`cfctl`	Reserved	`IDET`	`RST`	`PWR`	`DET`
1	`idectl`	Reserved				`IIDE`
2	Reserved	Reserved				
3	Reserved	Reserved				

Cfctl Register

The `cfctl` register controls the operations of the CompactFlash core. Reading the `cfctl` register clears the interrupt. Table 2–4 describes the `cfctl` register bits.

Table 2–4. cfctl Register Bits

Bit Number	Bit Name	Read/Write	Description
0	`DET`	RO	Detect. This bit is set to 1 when the core detects a CompactFlash device.
1	`PWR`	RW	Power. When this bit is set to 1, power is being supplied to the CompactFlash device.
2	`RST`	RW	Reset. When this bit is set to 1, the CompactFlash device is held in a reset state. Setting this bit to 0 returns the device to its active state.
3	`IDET`	RW	Detect Interrupt Enable. When this bit is set to 1, the CompactFlash core generates an interrupt each time the value of the `det` bit changes.

Idectl Register

The `idectl` register controls the interface to the CompactFlash device. Table 2–5 describes the `idectl` register bit.

Table 2–5. idectl Register

Bit Number	Bit Name	Read/Write	Description
0	`IIDE`	RW	IDE Interrupt Enable. When this bit is set to 1, the CompactFlash core generates an interrupt following an interrupt generated by the CompactFlash device. Setting this bit to 0 disables the IDE interrupt.

Document Revision History

Table 2–6 shows the revision history for this chapter.

Table 2–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Added the mode supported by the CompactFlash core.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

3. Common Flash Interface Controller Core

NII51013-9.0.0

Core Overview

The common flash interface controller core with Avalon® interface (CFI controller) allows you to easily connect SOPC Builder systems to external flash memory that complies with the Common Flash Interface (CFI) specification. The CFI controller is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system.

For the Nios® II processor, Altera provides hardware abstraction layer (HAL) driver routines for the CFI controller. The drivers provide universal access routines for CFI-compliant flash memories. Therefore, you do not need to write any additional code to program CFI-compliant flash devices. The HAL driver routines take advantage of the HAL generic device model for flash memory, which allows you to access the flash memory using the familiar HAL application programming interface (API), the ANSI C standard library functions for file I/O, or both.

The Nios II Embedded Design Suite (EDS) provides a flash programmer utility based on the Nios II processor and the CFI controller. The flash programmer utility can be used to program any CFI-compliant flash memory connected to an Altera® device.

For more information about how to read and write flash using the HAL API, refer to the *Nios II Software Developer's Handbook*. For more information on the flash programmer utility, refer to the *Nios II Flash Programmer User Guide*.

Further information about the Common Flash Interface specification is available at www.intel.com. As an example of a flash device supported by the CFI controller, see the data sheet for the AMD Am29LV065D-120R, available at www.amd.com.

The common flash interface controller core supersedes previous Altera flash cores distributed with SOPC Builder or Nios development kits. All flash chips associated with these previous cores comply with the CFI specification, and therefore are supported by the CFI controller.

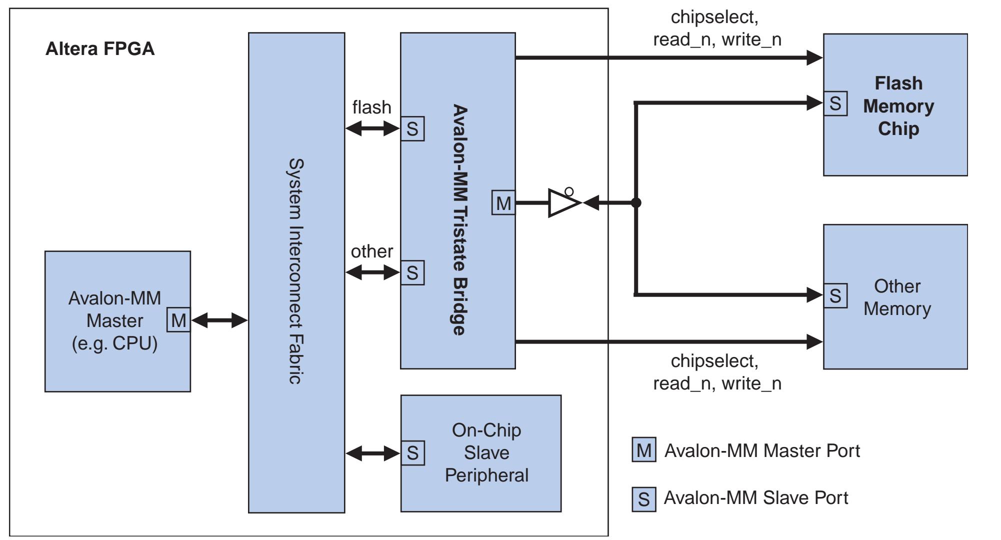
This chapter contains the following sections:

- "Functional Description" on page 3–2
- "Device and Tools Support" on page 3–2
- "Instantiating the Core in SOPC Builder" on page 3–2
- "Software Programming Model" on page 3–4

Functional Description

Figure 3–1 shows a block diagram of the CFI controller in a typical system configuration. As shown in Figure 3–1, the Avalon Memory-Mapped (Avalon-MM) interface for flash devices is connected through an Avalon-MM tristate bridge. The tristate bridge creates an off-chip memory bus that allows the flash chip to share address and data pins with other memory chips. It provides separate chipselect, read, and write pins to each chip connected to the memory bus. The CFI controller hardware is minimal; it is simply an Avalon-MM tristate slave port configured with waitstates, setup, and hold time appropriate for the target flash chip. This slave port is capable of Avalon-MM tristate slave read and write transfers.

Figure 3–1. An SOPC Builder System Integrating a CFI Controller



Avalon-MM master ports can perform read transfers directly from the CFI controller's Avalon-MM port. See "Software Programming Model" on page 3–4 for more detail on writing/erasing flash memory.

Device and Tools Support

The CFI controller supports all Altera device families. The CFI controller provides drivers for the Nios II HAL system library.

Instantiating the Core in SOPC Builder

Hardware designers use the MegaWizard™ interface for the CFI controller in SOPC Builder to specify the core features. The following sections describe the available options.

Attributes Page

The options on this page control the basic hardware configuration of the CFI controller.

Presets Settings

The **Presets** setting is a drop-down menu of flash chips that have already been characterized for use with the CFI controller. After you select one of the chips in the **Presets** menu, the wizard updates all settings on both tabs (except for the Board Info setting) to work with the specified flash chip.

The options provided are not intended to cover the wide range of flash devices available in the market. If the flash chip on your target board does not appear in the **Presets** list, you must configure the other settings manually.

Size Settings

The size setting specifies the size of the flash device. There are two settings:

- **Address Width**—The width of the flash chip's address bus.
- **Data Width—**The width of the flash chip's data bus

The size settings cause SOPC Builder to allocate the correct amount of address space for this device. SOPC Builder will automatically generate dynamic bus sizing logic that appropriately connects the flash chip to Avalon-MM master ports of different data widths.

For details about dynamic bus sizing, refer to the *Avalon Interface Specifications*.

Timing Page

The options on this page specify the timing requirements for read and write transfers with the flash device.

Refer to the specifications provided with the common flash device you are using to obtain the timing values you need to calculate the values of the parameters on the **Timing** page.

The settings available on the **Timing** page are:

- **Setup**—After asserting `chipselect`, the time required before asserting the `read` or `write` signals. You can determine the value of this parameter by using the following formula:

  Setup = t_{CE} (chip enable to output delay) - t_{OE} (output enable to output delay)

- **Wait**—The time required for the `read` or `write` signals to be asserted for each transfer. Use the following guideline to determine an appropriate value for this parameter:

  The sum of **Setup**, **Wait**, and board delay must be less than t_{ACC}, where:

  - Board delay is determined by the T_{CO} on the device address pins, T_{SU} on the device data pins and propagation delay on the board traces in both directions.
  - t_{ACC} is the address to output delay.

- **Hold**—After deasserting the `write` signal, the time required before deasserting the `chipselect` signal.
- **Units**—The timing units used for the **Setup**, **Wait**, and **Hold** values. Possible values include ns, µs, ms, and clock cycles.

For more information about signal timing for the Avalon-MM interface, refer to the *Avalon Interface Specifications*.

Software Programming Model

This section describes the software programming model for the CFI controller. In general, any Avalon-MM master in the system can read the flash chip directly as a memory device. For Nios II processor users, Altera provides HAL system library drivers that enable you to erase and write the flash memory using the HAL API functions.

HAL System Library Support

The Altera-provided driver implements a HAL flash device driver that integrates into the HAL system library for Nios II systems. Programs call the familiar HAL API functions to program CFI-compliant flash memory. You do not need to know anything about the details of the underlying drivers.

The HAL API for programming flash, including C code examples, is described in detail in the *Nios II Software Developer's Handbook*. The Nios II EDS also provides a reference design called Flash Tests that demonstrates erasing, writing, and reading flash memory.

Limitations

Currently, the Altera-provided drivers for the CFI controller support only Intel, AMD and Spansion flash chips.

Software Files

The CFI controller provides the following software files. These files define the low-level access to the hardware, and provide the routines for the HAL flash device driver. Application developers should not modify these files.

- **altera_avalon_cfi_flash.h, altera_avalon_cfi_flash.c**—The header and source code for the functions and variables required to integrate the driver into the HAL system library.
- **altera_avalon_cfi_flash_funcs.h**, **altera_avalon_cfi_flash_table.c**—The header and source code for functions concerned with accessing the CFI table.
- **altera_avalon_cfi_flash_amd_funcs.h**, **altera_avalon_cfi_flash_amd.c**—The header and source code for programming AMD CFI-compliant flash chips.
- **altera_avalon_cfi_flash_intel_funcs.h**, **altera_avalon_cfi_flash_intel.c**—The header and source code for programming Intel CFI-compliant flash chips.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *Nios II Flash Programmer User Guide*
- *Nios II Software Developer's Handbook*

Document Revision History

Table 3–1 shows the revision history for this chapter.

Table 3–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. Added description to parameters on **Timing** page.	—
May 2008 v8.0.0	Updated the CFI controllers supported by Altera-provided drivers.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

4. EPCS Device Controller Core

NII51012-9.0.0

Core Overview

The EPCS device controller core with Avalon® interface allows Nios® II systems to access an Altera® EPCS serial configuration device. Altera provides drivers that integrate into the Nios II hardware abstraction layer (HAL) system library, allowing you to read and write the EPCS device using the familiar HAL application program interface (API) for flash devices.

Using the EPCS device controller core, Nios II systems can:

- Store program code in the EPCS device. The EPCS device controller core provides a boot-loader feature that allows Nios II systems to store the main program code in an EPCS device.
- Store non-volatile program data, such as a serial number, a NIC number, and other persistent data.
- Manage the device configuration data. For example, a network-enabled embedded system can receive new FPGA configuration data over a network, and use the core to program the new data into an EPCS serial configuration device.

The EPCS device controller core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. The flash programmer utility in the Nios II IDE allows you to manage and program data contents into the EPCS device.

For information about the EPCS serial configuration device family, refer to the *Serial Configuration Devices (EPCS1, EPCS4, EPCS16, EPCS64 and EPCS128) Data Sheet*. For details about using the Nios II HAL API to read and write flash memory, refer to the *Nios II Software Developer's Handbook*. For details about managing and programming the EPCS memory contents, refer to the *Nios II Flash Programmer User Guide*.

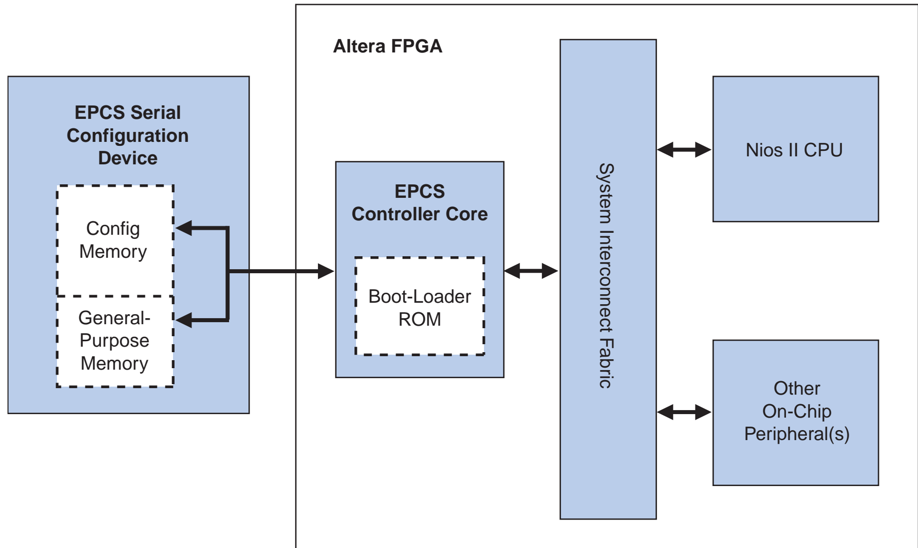
For Nios II processor users, the EPCS device controller core supersedes the Active Serial Memory Interface (ASMI) device. New designs should use the EPCS device controller core instead of the ASMI core.

Functional Description

Figure 4–1 shows a block diagram of the EPCS device controller core in a typical system configuration. As shown in Figure 4–1, the EPCS device's memory can be thought of as two separate regions:

- **FPGA configuration memory**—FPGA configuration data is stored in this region.
- **General-purpose memory**—If the FPGA configuration data does not fill up the entire EPCS device, any left-over space can be used for general-purpose data and system startup code.

Figure 4–1. Nios II System Integrating an EPCS Device Controller Core



By virtue of the HAL generic device model for flash devices, accessing the EPCS device using the HAL API is the same as accessing any flash memory. The EPCS device has a special-purpose hardware interface, so Nios II programs must read and write the EPCS memory using the provided HAL flash drivers.

The EPCS device controller core contains an on-chip memory for storing a boot-loader program. When used in conjunction with Cyclone®, Cyclone II, and Cyclone III devices, the core requires 512 bytes of boot-loader ROM. For Stratix® II and Stratix III devices, the core requires 1 KByte of boot-loader ROM. The Nios II processor can be configured to boot from the EPCS device controller core. To do so, set the Nios II reset address to the base address of the EPCS device controller core. In this case, after reset the CPU first executes code from the boot-loader ROM, which copies data from the EPCS general-purpose memory region into a RAM. Then, program control transfers to the RAM. The Nios II IDE provides facilities to compile a program for storage in the EPCS device, and create a programming file to program into the EPCS device.

For more information, refer to the *Nios II Flash Programmer User Guide*.

The Altera EPCS configuration device connects to the FPGA through dedicated pins on the FPGA, not through general-purpose I/O pins. In all Altera device families except Cyclone III, the EPCS device controller core does not create any I/O ports on the top-level SOPC Builder system module. If the EPCS device and the FPGA are wired together on a board for configuration using the EPCS device (in other words, active serial configuration mode), no further connection is necessary between the EPCS device controller core and the EPCS device. When you compile the SOPC Builder system in the Quartus II software, the EPCS device controller core signals are routed automatically to the device pins for the EPCS device.

If you program the EPCS device using the Quartus® II Programmer, all previous content is erased. To program the EPCS device with a combination of FPGA configuration data and Nios II program data, use the Nios II IDE flash programmer utility.

You have the flexibility to connect the output pins of Cyclone III devices, which are exported to the top-level design, to any EPCS devices. Perform the following tasks in the Quartus® II software to make the necessary pin assignments:

- On the **Dual-purpose pins** page (**Assignments** > **Devices** > **Device and Pin Options**), ensure that the following pins are assigned to the respective values:
  - `Data[0]` = **Use as regular I/O**
  - `Data[1]` = **Use as regularr I/O**
  - `DCLK` = **Use as regular I/O**
  - `FLASH_nCE/nCS0` = **Use as regular I/O**
- Using the Pin Planner (**Assignments > Pins**), ensure that the following pins are assigned to the respective configuration functions on the device:
  - `data0_to_the_epcs_controller` = `DATA0`
  - `sdo_from the_epcs_controller` = `DATA1,ASDO`
  - `dclk_from_epcs_controller` = `DCLK`
  - `sce_from_the_epcs_controller` = `FLASH_nCE`

For more information about the configuration pins in Cyclone III devices, refer to the Pin-Out Files for Altera Device page.

Avalon-MM Slave Interface and Registers

The EPCS device controller core has a single Avalon-MM slave interface that provides access to both boot-loader code and registers that control the core. As shown in Table 4–1, the first segment is dedicated to the boot-loader code, and the next seven words are control and data registers. A Nios II CPU can read the instruction words, starting from the core's base address as flat memory space, which enables the CPU to reset the core's address space.

The EPCS device controller core includes an interrupt signal that can be used to interrupt the CPU when a transfer has completed.

Table 4–1. EPCS Device Controller Core Register Map

Offset—Cyclone and Cyclone II (32-bit Word Address)	Offset—Other Device Families (32-bit Word Address)	Register Name	R/W	Bit Description 31:0
`0x000 .. 0x0FF`	`0x000 .. 0x1FF`	`Boot ROM Memory`	R	Boot Loader Code
`0x100`	`0x080`	`Read Data`	R	*(1)*
`0x101`	`0x081`	`Write Data`	W	
`0x102`	`0x082`	`Status`	R/W	
`0x103`	`0x083`	`Control`	R/W	
`0x104`	`0x084`	`Reserved`	—	
`0x105`	`0x085`	`Slave Enable`	R/W	
`0x106`	`0x086`	`End of Packet`	R/W	

Note to Table 4–1:

(1) Altera does not publish the usage of the control and data registers. To access the EPCS device, you must use the HAL drivers provided by Altera.

Device and Tools Support

The EPCS device controller core supports all Altera device families except the Hardcopy® series. The core must be connected to a Nios II processor. The core provides drivers for HAL-based Nios II systems, and the precompiled boot loader code compatible with the Nios II processor.

Instantiating the Core in SOPC Builder

You can add the EPCS device controller core from the **System Contents** tab in SOPC Builder. There are no user-configurable settings for this component.

Only one EPCS device controller core can be instantiated in each FPGA design.

Software Programming Model

This section describes the software programming model for the EPCS device controller core. Altera provides HAL system library drivers that enable you to erase and write the EPCS memory using the HAL API functions. Altera does not publish the usage of the cores registers. Therefore, you must use the HAL drivers provided by Altera to access the EPCS device.

HAL System Library Support

The Altera-provided driver implements a HAL flash device driver that integrates into the HAL system library for Nios II systems. Programs call the familiar HAL API functions to program the EPCS memory. You do not need to know the details of the underlying drivers to use them.

The HAL API for programming flash, including C-code examples, is described in detail in the *Nios II Software Developer's Handbook*. For details about managing and programming the EPCS device contents, refer to the *Nios II Flash Programmer User Guide*.

Software Files

The EPCS device controller core provides the following software files. These files provide low-level access to the hardware and drivers that integrate into the Nios II HAL system library. Application developers should not modify these files.

- **altera_avalon_epcs_flash_controller.h**, **altera_avalon_epcs_flash_controller.c**—Header and source files that define the drivers required for integration into the HAL system library.
- **epcs_commands.h**, **epcs_commands.c**—Header and source files that directly control the EPCS device hardware to read and write the device. These files also rely on the Altera SPI core drivers.

© March 2009 Altera Corporation

Referenced Documents

This chapter references the following documents:

- *Nios II Flash Programmer User Guide*
- *Nios II Software Developer's Handbook*
- *Serial Configuration Devices (EPCS1, EPCS4, EPCS16, EPCS64 and EPCS128) Data Sheet*

Document Revision History

Table 4–2 shows the revision history for this chapter.

Table 4–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Updated the boot ROM memory offset for other device familes in Table 4–1.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	■ Updated the boot rom size. ■ Added additional steps to perform to connect output pins in Cyclone III devices.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

5. JTAG UART Core

NII51009-9.0.0

Core Overview

The JTAG UART core with Avalon® interface implements a method to communicate serial character streams between a host PC and an SOPC Builder system on an Altera® FPGA. In many designs, the JTAG UART core eliminates the need for a separate RS-232 serial connection to a host PC for character I/O. The core provides an Avalon interface that hides the complexities of the JTAG interface from embedded software programmers. Master peripherals (such as a Nios® II processor) communicate with the core by reading and writing control and data registers.

The JTAG UART core uses the JTAG circuitry built in to Altera FPGAs, and provides host access via the JTAG pins on the FPGA. The host PC can connect to the FPGA via any Altera JTAG download cable, such as the USB-Blaster™ cable. Software support for the JTAG UART core is provided by Altera. For the Nios II processor, device drivers are provided in the HAL system library, allowing software to access the core using the ANSI C Standard Library **stdio.h** routines. For the host PC, Altera provides JTAG terminal software that manages the connection to the target, decodes the JTAG data stream, and displays characters on screen.

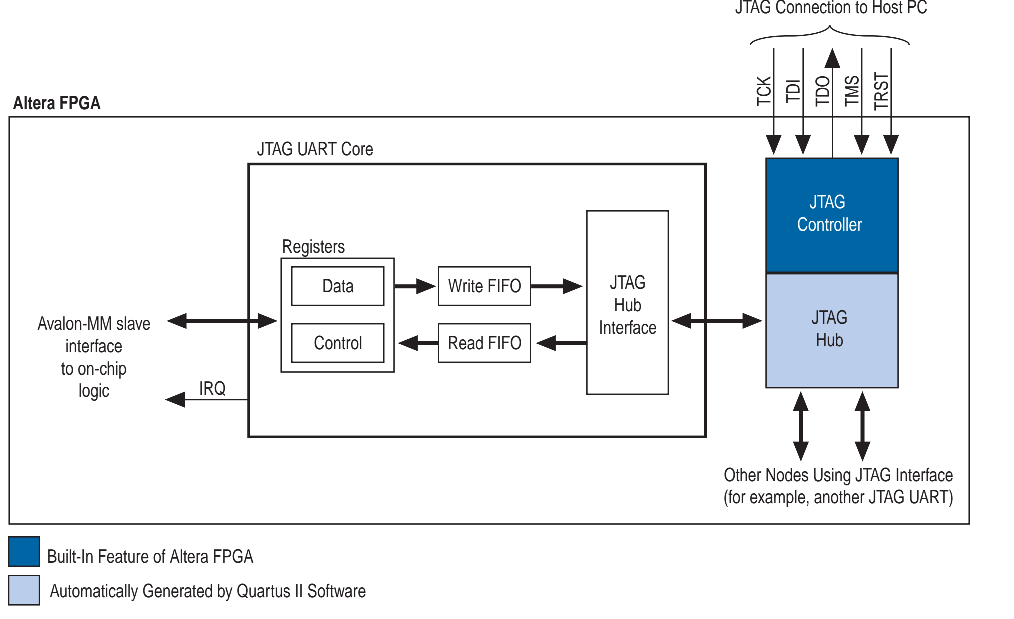
The JTAG UART core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description" on page 5–2
- "Device and Tools Support" on page 5–4
- "Instantiating the Core in SOPC Builder" on page 5–4
- "Hardware Simulation Considerations" on page 5–6
- "Software Programming Model" on page 5–6

Functional Description

Figure 5–1 shows a block diagram of the JTAG UART core and its connection to the JTAG circuitry inside an Altera FPGA. The following sections describe the components of the core.

Figure 5–1. JTAG UART Core Block Diagram



Avalon Slave Interface and Registers

The JTAG UART core provides an Avalon slave interface to the JTAG circuitry on an Altera FPGA. The user-visible interface to the JTAG UART core consists of two 32-bit registers, `data` and `control`, that are accessed through an Avalon slave port. An Avalon master, such as a Nios II processor, accesses the registers to control the core and transfer data over the JTAG connection. The core operates on 8-bit units of data at a time; eight bits of the `data` register serve as a one-character payload.

The JTAG UART core provides an active-high interrupt output that can request an interrupt when read data is available, or when the write FIFO is ready for data. For further details see "Interrupt Behavior" on page 5–11.

Read and Write FIFOs

The JTAG UART core provides bidirectional FIFOs to improve bandwidth over the JTAG connection. The FIFO depth is parameterizable to accommodate the available on-chip memory. The FIFOs can be constructed out of memory blocks or registers, allowing you to trade off logic resources for memory resources, if necessary.

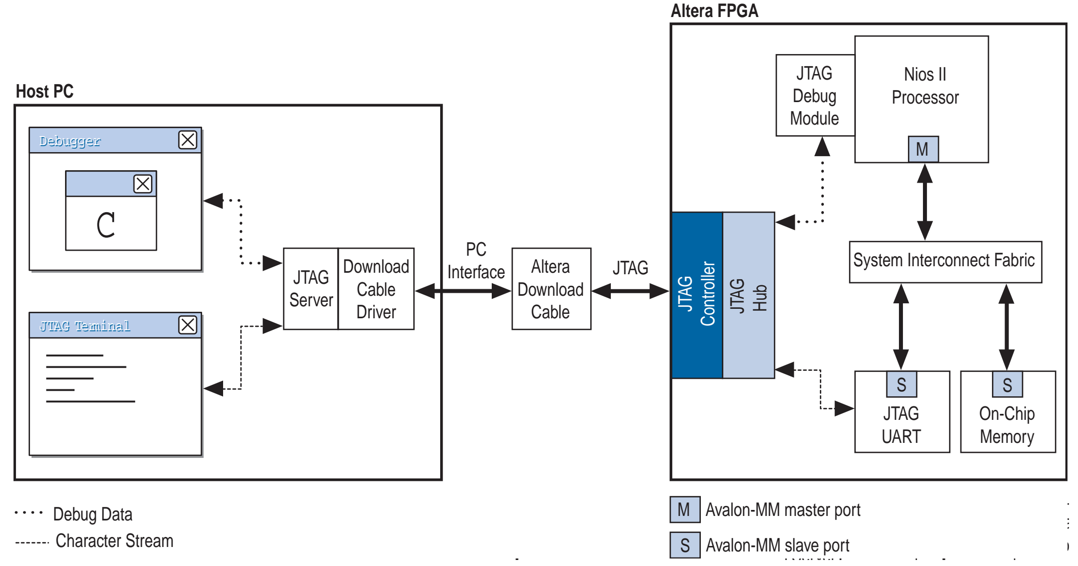
JTAG Interface

Altera FPGAs contain built-in JTAG control circuitry between the device's JTAG pins and the logic inside the device. The JTAG controller can connect to user-defined circuits called *nodes* implemented in the FPGA. Because several nodes may need to communicate via the JTAG interface, a JTAG hub, which is a multiplexer, is necessary. During logic synthesis and fitting, the Quartus® II software automatically generates the JTAG hub logic. No manual design effort is required to connect the JTAG circuitry inside the device; the process is presented here only for clarity.

Host-Target Connection

Figure 5–2 shows the connection between a host PC and an SOPC Builder-generated system containing a JTAG UART core.

Figure 5–2. Example System Using the JTAG UART Core



The JTAG controller on the FPGA and the download cable driver on the host PC implement a simple data-link layer between host and target. All JTAG nodes inside the FPGA are multiplexed through the single JTAG connection. JTAG server software on the host PC controls and decodes the JTAG data stream, and maintains distinct connections with nodes inside the FPGA.

The example system in Figure 5–2 contains one JTAG UART core and a Nios II processor. Both agents communicate with the host PC over a single Altera download cable. Thanks to the JTAG server software, each host application has an independent connection to the target. Altera provides the JTAG server drivers and host software required to communicate with the JTAG UART core.

Systems with multiple JTAG UART cores are possible, and all cores communicate via the same JTAG interface. To maintain coherent data streams, only one processor should communicate with each JTAG UART core.

Device and Tools Support

The JTAG UART core supports all Altera® device families. The JTAG UART core is supported by the Nios II hardware abstraction layer (HAL) system library.

To view the character stream on the host PC, the JTAG UART core must be used in conjunction with the JTAG terminal software provided by Altera. Nios II processor users access the JTAG UART via the Nios II IDE or the **nios2-terminal** command-line utility.

For further details, refer to the *Nios II Software Developer's Handbook* or the Nios II IDE online help.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the JTAG UART core in SOPC Builder to specify the core features. The following sections describe the available options.

Configuration Page

The options on this page control the hardware configuration of the JTAG UART core. The default settings are pre-configured to behave optimally with the Altera-provided device drivers and JTAG terminal software. Most designers should not change the default values, except for the **Construct using registers instead of memory blocks** option.

Write FIFO Settings

The write FIFO buffers data flowing from the Avalon interface to the host. The following settings are available:

- **Depth**—The write FIFO depth can be set from 8 to 32,768 bytes. Only powers of two are allowed. Larger values consume more on-chip memory resources. A depth of 64 is generally optimal for performance, and larger values are rarely necessary.
- **IRQ Threshold—**The write IRQ threshold governs how the core asserts its IRQ in response to the FIFO emptying. As the JTAG circuitry empties data from the write FIFO, the core asserts its IRQ when the number of characters remaining in the FIFO reaches this threshold value. For maximum bandwidth, a processor should service the interrupt by writing more data and preventing the write FIFO from emptying completely. A value of 8 is typically optimal. See "Interrupt Behavior" on page 5–11 for further details.
- **Construct using registers instead of memory blocks**—Turning on this option causes the FIFO to be constructed out of on-chip logic resources. This option is useful when memory resources are limited. Each byte consumes roughly 11 logic elements (LEs), so a FIFO depth of 8 (bytes) consumes roughly 88 LEs.

Read FIFO Settings

The read FIFO buffers data flowing from the host to the Avalon interface. Settings are available to control the depth of the FIFO and the generation of interrupts.

- **Depth**—The read FIFO depth can be set from 8 to 32,768 bytes. Only powers of two are allowed. Larger values consume more on-chip memory resources. A depth of 64 is generally optimal for performance, and larger values are rarely necessary.
- **IRQ Threshold—**The IRQ threshold governs how the core asserts its IRQ in response to the FIFO filling up. As the JTAG circuitry fills up the read FIFO, the core asserts its IRQ when the amount of space remaining in the FIFO reaches this threshold value. For maximum bandwidth, a processor should service the interrupt by reading data and preventing the read FIFO from filling up completely. A value of 8 is typically optimal. See "Interrupt Behavior" on page 5–11 for further details.
- **Construct using registers instead of memory blocks**—Turning on this option causes the FIFO to be constructed out of logic resources. This option is useful when memory resources are limited. Each byte consumes roughly 11 LEs, so a FIFO depth of 8 (bytes) consumes roughly 88 LEs.

Simulation Settings

At system generation time, when SOPC Builder generates the logic for the JTAG UART core, a simulation model is also constructed. The simulation model offers features to simplify simulation of systems using the JTAG UART core. Changes to the simulation settings do not affect the behavior of the core in hardware; the settings affect only functional simulation.

Simulated Input Character Stream

You can enter a character stream that will be simulated entering the read FIFO upon simulated system reset. The MegaWizard Interface accepts an arbitrary character string, which is later incorporated into the test bench. After reset, this character string is pre-initialized in the read FIFO, giving the appearance that an external JTAG terminal program is sending a character stream to the JTAG UART core.

Prepare Interactive Windows

At system generation time, the JTAG UART core generator can create ModelSim® macros to open interactive windows during simulation. These windows allow the user to send and receive ASCII characters via a console, giving the appearance of a terminal session with the system executing in hardware. The following options are available:

- **Do not generate ModelSim aliases for interactive windows**—This option does not create any ModelSim macros for character I/O.
- **Create ModelSim alias to open a window showing output as ASCII text**—This option creates a ModelSim macro to open a console window that displays output from the write FIFO. Values written to the write FIFO via the Avalon interface are displayed in the console as ASCII characters.

- **Create ModelSim alias to open an interactive stimulus/response window**—This option creates a ModelSim macro to open a console window that allows input and output interaction with the core. Values written to the write FIFO via the Avalon interface are displayed in the console as ASCII characters. Characters typed into the console are fed into the read FIFO, and can be read via the Avalon interface. When this option is enabled, the simulated character input stream option is ignored.

Hardware Simulation Considerations

The simulation features were created for easy simulation of Nios II processor systems when using the ModelSim simulator. The simulation model is implemented in the JTAG UART core's top-level HDL file. The synthesizable HDL and the simulation HDL are implemented in the same file. Some simulation features are implemented using `translate on/off` synthesis directives that make certain sections of HDL code visible only to the synthesis tool.

For complete details about simulating the JTAG UART core in Nios II systems, refer to *AN 351: Simulating Nios II Processor Designs*.

Other simulators can be used, but require user effort to create a custom simulation process. You can use the auto-generated ModelSim scripts as references to create similar functionality for other simulators.

Do not edit the simulation directives if you are using Altera's recommended simulation procedures. If you change the simulation directives to create a custom simulation flow, be aware that SOPC Builder overwrites existing files during system generation. Take precautions to ensure your changes are not overwritten.

Software Programming Model

The following sections describe the software programming model for the JTAG UART core, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides HAL system library drivers that enable you to access the JTAG UART using the ANSI C standard library functions, such as `printf()` and `getchar()`.

HAL System Library Support

The Altera-provided driver implements a HAL character-mode device driver that integrates into the HAL system library for Nios II systems. HAL users should access the JTAG UART via the familiar HAL API and the ANSI C standard library, rather than accessing the JTAG UART registers. `ioctl()` requests are defined that allow HAL users to control the hardware-dependent aspects of the JTAG UART.

If your program uses the Altera-provided HAL device driver to access the JTAG UART hardware, accessing the device registers directly will interfere with the correct behavior of the driver.

For Nios II processor users, the HAL system library API provides complete access to the JTAG UART core's features. Nios II programs treat the JTAG UART core as a character mode device, and send and receive data using the ANSI C standard library functions, such as `getchar()` and `printf()`.

Example 5–1 demonstrates the simplest possible usage, printing a message to `stdout` using `printf()`. In this example, the SOPC Builder system contains a JTAG UART core, and the HAL system library is configured to use this JTAG UART device for `stdout`.

Example 5–1. *Printing Characters to a JTAG UART Core as stdout*

```
#include <stdio.h>
int main ()
{
   printf("Hello world.\n");
   return 0;
}
```

Example 5–2 demonstrates reading characters from and sending messages to a JTAG UART core using the C standard library. In this example, the SOPC Builder system contains a JTAG UART core named `jtag_uart` that is not necessarily configured as the `stdout` device. In this case, the program treats the device like any other node in the HAL file system.

Example 5–2. Transmitting Characters to a JTAG UART Core

```
/* A simple program that recognizes the characters 't' and 'v' */
#include <stdio.h>
#include <string.h>
int main ()
{
   char* msg = "Detected the character 't'.\n";
   FILE* fp;
   char prompt = 0;

fp = fopen ("/dev/jtag_uart", "r+"); //Open file for reading and writing
if (fp)
{
   while (prompt != 'v')
   {  // Loop until we receive a 'v'.
      prompt = getc(fp);  // Get a character from the JTAG UART.
      if (prompt == 't')
      {  // Print a message if character is 't'.
         fwrite (msg, strlen (msg), 1, fp);
      }

      if (ferror(fp))// Check if an error occurred with the file
      pointer clearerr(fp);// If so, clear it.
   }

   fprintf(fp, "Closing the JTAG UART file handle.\n");
   fclose (fp);
   }

   return 0;
}
```

In this example, the `ferror(fp)` is used to check if an error occurred on the JTAG UART connection, such as a disconnected JTAG connection. In this case, the driver detects that the JTAG connection is disconnected, reports an error (`EIO`), and discards data for subsequent transactions. If this error ever occurs, the C library latches the value until you explicitly clear it with the `clearerr()` function.

For complete details of the HAL system library, refer to the *Nios II Software Developer's Handbook*.

The Nios II Embedded Design Suite (EDS) provides a number of software example designs that use the JTAG UART core.

Driver Options: Fast vs. Small Implementations

To accommodate the requirements of different types of systems, the JTAG UART driver has two variants, a fast version and a small version. The fast behavior is used by default. Both the fast and small drivers fully support the C standard library functions and the HAL API.

The fast driver is an interrupt-driven implementation, which allows the processor to perform other tasks when the device is not ready to send or receive data. Because the JTAG UART data rate is slow compared to the processor, the fast driver can provide a large performance benefit for systems that could be performing other tasks in the interim. In addition, the fast version of the Altera Avalon JTAG UART monitors the connection to the host. The driver discards characters if no host is connected, or if the host is not running an application that handles the I/O stream.

The small driver is a polled implementation that waits for the JTAG UART hardware before sending and receiving each character. The performance of the small driver is poor if you are sending large amounts of data. The small version assumes that the host is always connected, and will never discard characters. Therefore, the small driver will hang the system if the JTAG UART hardware is ever disconnected from the host while the program is sending or receiving data. There are two ways to enable the small footprint driver:

- Enable the small footprint setting for the HAL system library project. This option affects device drivers for all devices in the system.
- Specify the preprocessor option `-DALTERA_AVALON_JTAG_UART_SMALL`. Use this option if you want the small, polled implementation of the JTAG UART driver, but you do not want to affect the drivers for other devices.

ioctl() Operations

The fast version of the JTAG UART driver supports the `ioctl()` function to allow HAL-based programs to request device-specific operations. Specifically, you can use the `ioctl()` operations to control the timeout period, and to detect whether or not a host is connected. The fast driver defines the `ioctl()` operations shown in Table 5–1.

Table 5–1. JTAG UART ioctl() Operations for the Fast Driver Only

Request	Meaning
`TIOCSTIMEOUT`	Set the timeout (in seconds) after which the driver will decide that the host is not connected. A timeout of 0 makes the target assume that the host is always connected. The `ioctl` arg parameter passed in must be a pointer to an integer.
`TIOCGCONNECTED`	Sets the integer arg parameter to a value that indicates whether the host is connected and acting as a terminal (1), or not connected (0). The `ioctl` arg parameter passed in must be a pointer to an integer.

For details about the `ioctl()` function, refer to the *Nios II Software Developer's Handbook*.

Software Files

The JTAG UART core is accompanied by the following software files. These files define the low-level interface to the hardware, and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_jtag_uart_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used only by device driver functions.
- **altera_avalon_jtag_uart.h**, **altera_avalon_jtag_uart.c**—These files implement the HAL system library device driver.

Accessing the JTAG UART Core via a Host PC

Host software is necessary for a PC to access the JTAG UART core. The Nios II IDE supports the JTAG UART core, and displays character I/O in a console window. Altera also provides a command-line utility called **nios2-terminal** that opens a terminal session with the JTAG UART core.

For further details, refer to the *Nios II Software Developer's Handbook* and Nios II IDE online help.

Register Map

Programmers using the HAL API never access the JTAG UART core directly via its registers. In general, the register map is only useful to programmers writing a device driver for the core.

The Altera-provided HAL device driver accesses the device registers directly. If you are writing a device driver, and the HAL driver is active for the same device, your driver will conflict and fail to operate.

Table 5–2 shows the register map for the JTAG UART core. Device drivers control and communicate with the core through the two, 32-bit memory-mapped registers.

Table 5–2. JTAG UART Core Register Map

Offset	Register Name	R/W	Bit Description															
			31	...	16	15	14	...	11	10	9	8	7	...	2	1	0
0	`data`	RW	`RAVAIL`			`RVALID`	Reserved						`DATA`				
1	`control`	RW	`WSPACE`			Reserved				`AC`	`WI`	`RI`	Reserved			`WE`	`RE`

Note to Table 5–2:

(1) Reserved fields—Read values are undefined. Write zero.

Data Register

Embedded software accesses the read and write FIFOs via the `data` register. Table 5–3 describes the function of each bit.

Table 5–3. data Register Bits

Bit(s)	Name	Access	Description
[7:0]	`DATA`	R/W	The value to transfer to/from the JTAG core. When writing, the `DATA` field holds a character to be written to the write FIFO. When reading, the `DATA` field holds a character read from the read FIFO.
[15]	`RVALID`	R	Indicates whether the `DATA` field is valid. If `RVALID`=1, the `DATA` field is valid, otherwise `DATA` is undefined.
[32:16]	`RAVAIL`	R	The number of characters remaining in the read FIFO (after the current read).

A read from the `data` register returns the first character from the FIFO (if one is available) in the `DATA` field. Reading also returns information about the number of characters remaining in the FIFO in the `RAVAIL` field. A write to the `data` register stores the value of the `DATA` field in the write FIFO. If the write FIFO is full, the character is lost.

Control Register

Embedded software controls the JTAG UART core's interrupt generation and reads status information via the `control` register. Table 5–4 describes the function of each bit.

Table 5–4. Control Register Bits

Bit(s)	Name	Access	Description
0	`RE`	R/W	Interrupt-enable bit for read interrupts.
1	`WE`	R/W	Interrupt-enable bit for write interrupts.
8	`RI`	R	Indicates that the read interrupt is pending.
9	`WI`	R	Indicates that the write interrupt is pending.
10	`AC`	R/C	Indicates that there has been JTAG activity since the bit was cleared. Writing 1 to `AC` clears it to 0.
[32:16]	`WSPACE`	R	The number of spaces available in the write FIFO.

A read from the `control` register returns the status of the read and write FIFOs. Writes to the register can be used to enable/disable interrupts, or clear the `AC` bit.

© March 2009 Altera Corporation

The RE and WE bits enable interrupts for the read and write FIFOs, respectively. The WI and RI bits indicate the status of the interrupt sources, qualified by the values of the interrupt enable bits (WE and RE). Embedded software can examine RI and WI to determine the condition that generated the IRQ. See "Interrupt Behavior" on page 5–11 for further details.

The AC bit indicates that an application on the host PC has polled the JTAG UART core via the JTAG interface. Once set, the AC bit remains set until it is explicitly cleared via the Avalon interface. Writing 1 to AC clears it. Embedded software can examine the AC bit to determine if a connection exists to a host PC. If no connection exists, the software may choose to ignore the JTAG data stream. When the host PC has no data to transfer, it can choose to poll the JTAG UART core as infrequently as once per second. Delays caused by other host software using the JTAG download cable could cause delays of up to 10 seconds between polls.

Interrupt Behavior

The JTAG UART core generates an interrupt when either of the individual interrupt conditions is pending and enabled.

Interrupt behavior is of interest to device driver programmers concerned with the bandwidth performance to the host PC. Example designs and the JTAG terminal program provided with Nios II Embedded Design Suite (EDS) are pre-configured with optimal interrupt behavior.

The JTAG UART core has two kinds of interrupts: write interrupts and read interrupts. The WE and RE bits in the `control` register enable/disable the interrupts.

The core can assert a write interrupt whenever the write FIFO is nearly empty. The nearly empty threshold, `write_threshold`, is specified at system generation time and cannot be changed by embedded software. The write interrupt condition is set whenever there are `write_threshold` or fewer characters in the write FIFO. It is cleared by writing characters to fill the write FIFO beyond the `write_threshold`. Embedded software should only enable write interrupts after filling the write FIFO. If it has no characters remaining to send, embedded software should disable the write interrupt.

The core can assert a read interrupt whenever the read FIFO is nearly full. The nearly full threshold value, `read_threshold`, is specified at system generation time and cannot be changed by embedded software. The read interrupt condition is set whenever the read FIFO has `read_threshold` or fewer spaces remaining. The read interrupt condition is also set if there is at least one character in the read FIFO and no more characters are expected. The read interrupt is cleared by reading characters from the read FIFO.

For optimum performance, the interrupt thresholds should match the interrupt response time of the embedded software. For example, with a 10-MHz JTAG clock, a new character is provided (or consumed) by the host PC every 1 µs. With a threshold of 8, the interrupt response time must be less than 8 µs. If the interrupt response time is too long, performance suffers. If it is too short, interrupts occurs too often.

For Nios II processor systems, read and write thresholds of 8 are an appropriate default.

Referenced Documents

This chapter references the *Nios II Software Developer's Handbook*.

Document Revision History

Table 5–5 shows the revision history for this chapter.

Table 5–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

© March 2009 Altera Corporation

6. UART Core

NII51010-9.0.0

Core Overview

The UART core with Avalon® interface implements a method to communicate serial character streams between an embedded system on an Altera® FPGA and an external device. The core implements the RS-232 protocol timing, and provides adjustable baud rate, parity, stop, and data bits, and optional RTS/CTS flow control signals. The feature set is configurable, allowing designers to implement just the necessary functionality for a given system.

The core provides an Avalon Memory-Mapped (Avalon-MM) slave interface that allows Avalon-MM master peripherals (such as a Nios® II processor) to communicate with the core simply by reading and writing control and data registers.

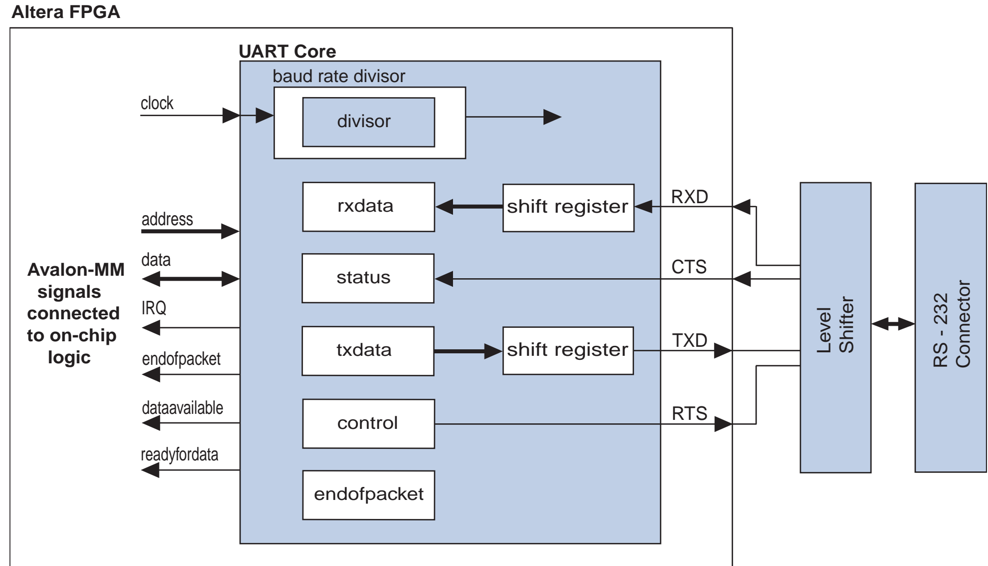
The UART core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 6–3
- "Instantiating the Core in SOPC Builder" on page 6–3
- "Simulation Considerations" on page 6–7
- "Software Programming Model" on page 6–8

Functional Description

Figure 6–1 shows a block diagram of the UART core.

Figure 6–1. Block Diagram of the UART Core in a Typical System



The core has two user-visible parts:

- The register file, which is accessed via the Avalon-MM slave port
- The RS-232 signals, `RXD`, `TXD`, `CTS`, and `RTS`

Avalon-MM Slave Interface and Registers

The UART core provides an Avalon-MM slave interface to the internal register file. The user interface to the UART core consists of six, 16-bit registers: `control`, `status`, `rxdata`, `txdata`, `divisor`, and `endofpacket`. A master peripheral, such as a Nios II processor, accesses the registers to control the core and transfer data over the serial connection.

The UART core provides an active-high interrupt request (IRQ) output that can request an interrupt when new data has been received, or when the core is ready to transmit another character. For further details, refer "Interrupt Behavior" on page 6–15.

The Avalon-MM slave port is capable of transfers with flow control. The UART core can be used in conjunction with a direct memory access (DMA) peripheral with Avalon-MM flow control to automate continuous data transfers between, for example, the UART core and memory.

For more information, refer to the *Timer Core* chapter in volume 5 of the *Quartus II Handbook*. For details about the Avalon-MM interface, refer to the *Avalon Interface Specifications*.

RS-232 Interface

The UART core implements RS-232 asynchronous transmit and receive logic. The UART core sends and receives serial data via the `TXD` and `RXD` ports. The I/O buffers on most Altera FPGA families do not comply with RS-232 voltage levels, and may be damaged if driven directly by signals from an RS-232 connector. To comply with RS-232 voltage signaling specifications, an external level-shifting buffer is required (for example, Maxim MAX3237) between the FPGA I/O pins and the external RS-232 connector.

The UART core uses a logic 0 for mark, and a logic 1 for space. An inverter inside the FPGA can be used to reverse the polarity of any of the RS-232 signals, if necessary.

Transmitter Logic

The UART transmitter consists of a 7-, 8-, or 9-bit `txdata` holding register and a corresponding 7-, 8-, or 9-bit transmit shift register. Avalon-MM master peripherals write the `txdata` holding register via the Avalon-MM slave port. The transmit shift register is loaded from the `txdata` register automatically when a serial transmit shift operation is not currently in progress. The transmit shift register directly feeds the `TXD` output. Data is shifted out to `TXD` LSB first.

These two registers provide double buffering. A master peripheral can write a new value into the `txdata` register while the previously written character is being shifted out. The master peripheral can monitor the transmitter's status by reading the `status` register's transmitter ready (`TRDY`), transmitter shift register empty (`tmt`), and transmitter overrun error (`TOE`) bits.

The transmitter logic automatically inserts the correct number of start, stop, and parity bits in the serial TXD data stream as required by the RS-232 specification.

Receiver Logic

The UART receiver consists of a 7-, 8-, or 9-bit receiver-shift register and a corresponding 7-, 8-, or 9-bit `rxdata` holding register. Avalon-MM master peripherals read the `rxdata` holding register via the Avalon-MM slave port. The `rxdata` holding register is loaded from the receiver shift register automatically every time a new character is fully received.

These two registers provide double buffering. The `rxdata` register can hold a previously received character while the subsequent character is being shifted into the receiver shift register.

A master peripheral can monitor the receiver's status by reading the `status` register's read-ready (`RRDY`), receiver-overrun error (`ROE`), break detect (`BRK`), parity error (`PE`), and framing error (`FE`) bits. The receiver logic automatically detects the correct number of start, stop, and parity bits in the serial RXD stream as required by the RS-232 specification. The receiver logic checks for four exceptional conditions, frame error, parity error, receive overrun error, and break, in the received data and sets corresponding `status` register bits.

Baud Rate Generation

The UART core's internal baud clock is derived from the Avalon-MM clock input. The internal baud clock is generated by a clock divider. The divisor value can come from one of the following sources:

- A constant value specified at system generation time
- The 16-bit value stored in the `divisor` register

The `divisor` register is an optional hardware feature. If it is disabled at system generation time, the divisor value is fixed and the baud rate cannot be altered.

Device Support

The UART core supports all Altera® device families.

Instantiating the Core in SOPC Builder

Instantiating the UART in hardware creates at least two I/O ports for each UART core: An `RXD` input, and a `TXD` output. Optionally, the hardware may include flow control signals, the `CTS` input and `RTS` output.

Use the MegaWizard™ interface for the UART core in SOPC Builder to configure the hardware feature set. The following sections describe the available options.

Configuration Settings

This section describes the configuration settings.

Baud Rate Options

The UART core can implement any of the standard baud rates for RS-232 connections. The baud rate can be configured in one of two ways:

- **Fixed rate**—The baud rate is fixed at system generation time and cannot be changed via the Avalon-MM slave port.
- **Variable rate**—The baud rate can vary, based on a clock divisor value held in the `divisor` register. A master peripheral changes the baud rate by writing new values to the `divisor` register.

The baud rate is calculated based on the clock frequency provided by the Avalon-MM interface. Changing the system clock frequency in hardware without regenerating the UART core hardware results in incorrect signaling.

Baud Rate (bps) Setting

The **Baud Rate** setting determines the default baud rate after reset. The **Baud Rate** option offers standard preset values.

The baud rate value is used to calculate an appropriate clock divisor value to implement the desired baud rate. Baud rate and divisor values are related as shown in Equation 6–1 and Equation 6–2:

Equation 6–1.

$$\text{divisor} = \text{int}\left(\frac{\text{clock frequency}}{\text{baud rate}} + 0.5\right)$$

Equation 6–2.

$$\text{baud rate} = \frac{\text{clock frequency}}{\text{divisor} + 1}$$

Baud Rate Can Be Changed By Software Setting

When this setting is on, the hardware includes a 16-bit `divisor` register at address offset 4. The `divisor` register is writable, so the baud rate can be changed by writing a new value to this register.

When this setting is off, the UART hardware does not include a `divisor` register. The UART hardware implements a constant baud divisor, and the value cannot be changed after system generation. In this case, writing to address offset 4 has no effect, and reading from address offset 4 produces an undefined result.

 © March 2009 Altera Corporation

Data Bits, Stop Bits, Parity

The UART core's parity, data bits and stop bits are configurable. These settings are fixed at system generation time; they cannot be altered via the register file. Table 6–1 explains the settings.

Table 6–1. Data Bits Settings

Setting	Legal Values	Description
Data Bits	7, 8, 9	This setting determines the widths of the `txdata`, `rxdata`, and `endofpacket` registers.
Stop Bits	1, 2	This setting determines whether the core transmits 1 or 2 stop bits with every character. The core always terminates a receive transaction at the first stop bit, and ignores all subsequent stop bits, regardless of this setting.
Parity	None, Even, Odd	This setting determines whether the UART core transmits characters with parity checking, and whether it expects received characters to have parity checking. When **Parity** is set to **None**, the transmit logic sends data without including a parity bit, and the receive logic presumes the incoming data does not include a parity bit. The `PE` bit in the `status` register is not implemented; it always reads 0. When **Parity** is set to **Odd** or **Even**, the transmit logic computes and inserts the required parity bit into the outgoing TXD bitstream, and the receive logic checks the parity bit in the incoming RXD bitstream. If the receiver finds data with incorrect parity, the `PE` bit in the `status` register is set to 1. When **Parity** is **Even**, the parity bit is 0 if the character has an even number of 1 bits; otherwise the parity bit is 1. Similarly, when parity is **Odd**, the parity bit is 0 if the character has an odd number of 1 bits.

Synchronizer Stages

The option **Synchronizer Stages** allows you to specify the length of synchronization register chains. These register chains are used when a metastable event is likely to occur and the length specified determines the meantime before failure. The register chain length, however, affects the latency of the core.

For more information on metastability in Altera devices, refer to *AN 42: Metastability in Altera Devices*. For more information on metastability analysis and synchronization register chains, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Flow Control

When the option **Include CTS/RTS pins and control register bits** is turned on, the UART core includes the following features:

- `cts_n` (logic negative CTS) input port
- `rts_n` (logic negative RTS) output port
- `CTS` bit in the `status` register
- `DCTS` bit in the `status` register
- `RTS` bit in the `control` register
- `IDCTS` bit in the `control` register

Based on these hardware facilities, an Avalon-MM master peripheral can detect CTS and transmit RTS flow control signals. The CTS input and RTS output ports are tied directly to bits in the status and control registers, and have no direct effect on any other part of the core. When using flow control, be sure the terminal program on the host side is also configured for flow control.

When the **Include CTS/RTS pins and control register bits** setting is off, the core does not include the aforementioned hardware and continuous writes to the UART may loose data. The control/status bits CTS, DCTS, IDCTS, and RTS are not implemented; they always read as 0.

Streaming Data (DMA) Control

The UART core's Avalon-MM interface optionally implements Avalon-MM transfers with flow control. Flow control allows an Avalon-MM master peripheral to write data only when the UART core is ready to accept another character, and to read data only when the core has data available. The UART core can also optionally include the end-of-packet register.

Include End-of-Packet Register

When this setting is on, the UART core includes:

- A 7-, 8-, or 9-bit endofpacket register at address-offset 5. The data width is determined by the **Data Bits** setting.
- EOP bit in the status register.
- IEOP bit in the control register.
- endofpacket signal in the Avalon-MM interface to support data transfers with flow control to and from other master peripherals in the system.

End-of-packet (EOP) detection allows the UART core to terminate a data transaction with an Avalon-MM master with flow control. EOP detection can be used with a DMA controller, for example, to implement a UART that automatically writes received characters to memory until a specified character is encountered in the incoming RXD stream. The terminating (EOP) character's value is determined by the endofpacket register.

When the EOP register is disabled, the UART core does not include the EOP resources. Writing to the endofpacket register has no effect, and reading produces an undefined value.

Simulation Settings

When the UART core's logic is generated, a simulation model is also created. The simulation model offers features to simplify and accelerate simulation of systems that use the UART core. Changes to the simulation settings do not affect the behavior of the UART core in hardware; the settings affect only functional simulation.

For examples of how to use the following settings to simulate Nios II systems, refer to *AN 351: Simulating Nios II Embedded Processor Designs*.

Simulated RXD-Input Character Stream

You can enter a character stream that is simulated entering the `RXD` port upon simulated system reset. The UART core's MegaWizard™ interface accepts an arbitrary character string, which is later incorporated into the UART simulation model. After reset in reset, the string is input into the `RXD` port character-by-character as the core is able to accept new data.

Prepare Interactive Windows

At system generation time, the UART core generator can create ModelSim macros that facilitate interaction with the UART model during simulation. You can turn on the following options:

- **Create ModelSim alias to open streaming output window** to create a ModelSim macro that opens a window to display all output from the `TXD` port.
- **Create ModelSim alias to open interactive stimulus window** to create a ModelSim macro that opens a window to accept stimulus for the `RXD` port. The window sends any characters typed in the window to the `RXD` port.

Simulated Transmitter Baud Rate

RS-232 transmission rates are often slower than any other process in the system, and it is seldom useful to simulate the functional model at the true baud rate. For example, at 115,200 bps, it typically takes thousands of clock cycles to transfer a single character. The UART simulation model has the ability to run with a constant clock divisor of 2, allowing the simulated UART to transfer bits at half the system clock speed, or roughly one character per 20 clock cycles. You can choose one of the following options for the simulated transmitter baud rate:

- **Accelerated (use divisor = 2)**—`TXD` emits one bit per 2 clock cycles in simulation.
- **Actual (use true baud divisor)**—`TXD` transmits at the actual baud rate, as determined by the `divisor` register.

Simulation Considerations

The simulation features were created for easy simulation of Nios II processor systems when using the ModelSim simulator. The documentation for the processor documents the suggested usage of these features. Other usages may be possible, but will require additional user effort to create a custom simulation process.

The simulation model is implemented in the UART core's top-level HDL file; the synthesizable HDL and the simulation HDL are implemented in the same file. The simulation features are implemented using `translate on` and `translate off` synthesis directives that make certain sections of HDL code visible only to the synthesis tool.

Do not edit the simulation directives if you are using Altera's recommended simulation procedures. If you do change the simulation directives for your custom simulation flow, be aware that SOPC Builder overwrites existing files during system generation. Take precaution so that your changes are not overwritten.

For details about simulating the UART core in Nios II processor systems, refer to *AN 351: Simulating Nios II Processor Designs*.

Software Programming Model

The following sections describe the software programming model for the UART core, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides hardware abstraction layer (HAL) system library drivers that enable you to access the UART core using the ANSI C standard library functions, such as `printf()` and `getchar()`.

HAL System Library Support

The Altera-provided driver implements a HAL character-mode device driver that integrates into the HAL system library for Nios II systems. HAL users should access the UART via the familiar HAL API and the ANSI C standard library, rather than accessing the UART registers. `ioctl()` requests are defined that allow HAL users to control the hardware-dependent aspects of the UART.

If your program uses the HAL device driver to access the UART hardware, accessing the device registers directly interferes with the correct behavior of the driver.

For Nios II processor users, the HAL system library API provides complete access to the UART core's features. Nios II programs treat the UART core as a character mode device, and send and receive data using the ANSI C standard library functions.

The driver supports the CTS/RTS control signals when they are enabled in SOPC Builder. Refer to "Driver Options: Fast Versus Small Implementations" on page 6–9.

The following code demonstrates the simplest possible usage, printing a message to `stdout` using `printf()`. In this example, the SOPC Builder system contains a UART core, and the HAL system library has been configured to use this device for `stdout`.

Example 6–1. Example: Printing Characters to a UART Core as stdout

```
#include <stdio.h>
int main ()
{
   printf("Hello world.\n");
   return 0;
}
```

The following code demonstrates reading characters from and sending messages to a UART device using the C standard library. In this example, the SOPC Builder system contains a UART core named `uart1` that is not necessarily configured as the `stdout` device. In this case, the program treats the device like any other node in the HAL file system.

Example 6–2. Example: Sending and Receiving Characters

```
/* A simple program that recognizes the characters 't' and 'v' */
#include <stdio.h>
#include <string.h>
int main ()
{
    char* msg = "Detected the character 't'.\n";
    FILE* fp;
    char prompt = 0;

    fp = fopen ("/dev/uart1", "r+"); //Open file for reading and writing
    if (fp)
    {
        while (prompt != 'v')
        {  // Loop until we receive a 'v'.
            prompt = getc(fp);  // Get a character from the UART.
            if (prompt == 't')
            {  // Print a message if character is 't'.
                fwrite (msg, strlen (msg), 1, fp);
            }
        }

        fprintf(fp, "Closing the UART file.\n");
        fclose (fp);
    }

    return 0;
}
```

For more information about the HAL system library, refer to the *Nios II Software Developer's Handbook*.

Driver Options: Fast Versus Small Implementations

To accommodate the requirements of different types of systems, the UART driver provides two variants: a fast version and a small version. The fast version is the default. Both fast and small drivers fully support the C standard library functions and the HAL API.

The fast driver is an interrupt-driven implementation, which allows the processor to perform other tasks when the device is not ready to send or receive data. Because the UART data rate is slow compared to the processor, the fast driver can provide a large performance benefit for systems that could be performing other tasks in the interim.

The small driver is a polled implementation that waits for the UART hardware before sending and receiving each character. There are two ways to enable the small footprint driver:

- Enable the small footprint setting for the HAL system library project. This option affects device drivers for all devices in the system as well.
- Specify the preprocessor option `-DALTERA_AVALON_UART_SMALL`. You can use this option if you want the small, polled implementation of the UART driver, but do not want to affect the drivers for other devices.

Refer to the help system in the Nios II IDE for details about how to set HAL properties and preprocessor options.

If the CTS/RTS flow control signals are enabled in hardware, the fast driver automatically uses them. The small driver always ignores them.

ioctl() Operations

The UART driver supports the `ioctl()` function to allow HAL-based programs to request device-specific operations. Table 6–2 defines operation requests that the UART driver supports.

Table 6–2. UART ioctl() Operations

Request	Description
`TIOCEXCL`	Locks the device for exclusive access. Further calls to `open()` for this device will fail until either this file descriptor is closed, or the lock is released using the `TIOCNXCL` `ioctl` request. For this request to succeed there can be no other existing file descriptors for this device. The parameter `arg` is ignored.
`TIOCNXCL`	Releases a previous exclusive access lock. The parameter `arg` is ignored.

Additional operation requests are also optionally available for the fast driver only, as shown in Table 6–3. To enable these operations in your program, you must set the preprocessor option `-DALTERA_AVALON_UART_USE_IOCTL`.

Table 6–3. Optional UART ioctl() Operations for the Fast Driver Only

Request	Description
`TIOCMGET`	Returns the current configuration of the device by filling in the contents of the input termios structure. *(1)* A pointer to this structure is supplied as the value of the parameter `opt`.
`TIOCMSET`	Sets the configuration of the device according to the values contained in the input termios structure. *(1)* A pointer to this structure is supplied as the value of the parameter `arg`.

Note to Table 8–3:

(1) The termios structure is defined by the Newlib C standard library. You can find the definition in the file <*Nios II EDS install path*>**/components/altera_hal/HAL/inc/sys/termios.h**.

For details about the `ioctl()` function, refer to the *Nios II Software Developer's Handbook*.

Limitations

The HAL driver for the UART core does not support the endofpacket register. Refer to "Register Map" for details.

Software Files

The UART core is accompanied by the following software files. These files define the low-level interface to the hardware, and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_uart_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used only by device driver functions.
- **altera_avalon_uart.h**, **altera_avalon_uart.c**—These files implement the UART core device driver for the HAL system library.

© March 2009 Altera Corporation

Register Map

Programmers using the HAL API never access the UART core directly via its registers. In general, the register map is only useful to programmers writing a device driver for the core.

The Altera-provided HAL device driver accesses the device registers directly. If you are writing a device driver and the HAL driver is active for the same device, your driver will conflict and fail to operate.

Table 6–4 shows the register map for the UART core. Device drivers control and communicate with the core through the memory-mapped registers.

Table 6–4. UART Core Register Map

Offset	Register Name	R/W	Description/Register Bits													
			15:13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	`rxdata`	RO	Reserved					*(1)*	*(1)*	Receive Data						
1	`txdata`	WO	Reserved					*(1)*	*(1)*	Transmit Data						
2	`status` *(2)*	RW	Reserved	eop	cts	dcts	(1)	e	rrdy	trdy	tmt	toe	roe	brk	fe	pe
3	`control`	RW	Reserved	ieop	rts	idcts	trbk	ie	irrdy	itrdy	itmt	itoe	iroe	ibrk	ife	ipe
4	`divisor` *(3)*	RW	Baud Rate Divisor													
5	`endof-packet` *(3)*	RW	Reserved					*(1)*	*(1)*	End-of-Packet Value						

Notes to Table 6–4:

(1) These bits may or may not exist, depending on the **Data Width** hardware option. If they do not exist, they read zero, and writing has no effect.

(2) Writing zero to the `status` register clears the `dcts`, `e`, `toe`, `roe`, `brk`, `fe`, and `pe` bits.

(3) This register may or may not exist, depending on hardware configuration options. If it does not exist, reading returns an undefined value and writing has no effect.

Some registers and bits are optional. These registers and bits exists in hardware only if it was enabled at system generation time. Optional registers and bits are noted in the following sections.

rxdata Register

The `rxdata` register holds data received via the `RXD` input. When a new character is fully received via the `RXD` input, it is transferred into the `rxdata` register, and the `status` register's `rrdy` bit is set to 1. The `status` register's `rrdy` bit is set to 0 when the `rxdata` register is read. If a character is transferred into the `rxdata` register while the `rrdy` bit is already set (in other words, the previous character was not retrieved), a receiver-overrun error occurs and the `status` register's `roe` bit is set to 1. New characters are always transferred into the `rxdata` register, regardless of whether the previous character was read. Writing data to the `rxdata` register has no effect.

txdata Register

Avalon-MM master peripherals write characters to be transmitted into the txdata register. Characters should not be written to txdata until the transmitter is ready for a new character, as indicated by the TRDY bit in the status register. The TRDY bit is set to 0 when a character is written into the txdata register. The TRDY bit is set to 1 when the character is transferred from the txdata register into the transmitter shift register. If a character is written to the txdata register when TRDY is 0, the result is undefined. Reading the txdata register returns an undefined value.

For example, assume the transmitter logic is idle and an Avalon-MM master peripheral writes a first character into the txdata register. The TRDY bit is set to 0, then set to 1 when the character is transferred into the transmitter shift register. The master can then write a second character into the txdata register, and the TRDY bit is set to 0 again. However, this time the shift register is still busy shifting out the first character to the TXD output. The TRDY bit is not set to 1 until the first character is fully shifted out and the second character is automatically transferred into the transmitter shift register.

status Register

The status register consists of individual bits that indicate particular conditions inside the UART core. Each status bit is associated with a corresponding interrupt-enable bit in the control register. The status register can be read at any time. Reading does not change the value of any of the bits. Writing zero to the status register clears the DCTS, E, TOE, ROE, BRK, FE, and PE bits.

The status register bits are shown in Table 6–5.

Table 6–5. status Register Bits (Part 1 of 2)

Bit	Name	Access	Description
0 *(1)*	PE	RC	Parity error. A parity error occurs when the received parity bit has an unexpected (incorrect) logic level. The PE bit is set to 1 when the core receives a character with an incorrect parity bit. The PE bit stays set to 1 until it is explicitly cleared by a write to the status register. When the PE bit is set, reading from the rxdata register produces an undefined value. If the **Parity** hardware option is not enabled, no parity checking is performed and the PE bit always reads 0. Refer to “Data Bits, Stop Bits, Parity” on page 6–5.
1	FE	RC	Framing error. A framing error occurs when the receiver fails to detect a correct stop bit. The FE bit is set to 1 when the core receives a character with an incorrect stop bit. The FE bit stays set to 1 until it is explicitly cleared by a write to the status register. When the FE bit is set, reading from the rxdata register produces an undefined value.
2	BRK	RC	Break detect. The receiver logic detects a break when the RXD pin is held low (logic 0) continuously for longer than a full-character time (data bits, plus start, stop, and parity bits). When a break is detected, the BRK bit is set to 1. The BRK bit stays set to 1 until it is explicitly cleared by a write to the status register.
3	ROE	RC	Receive overrun error. A receive-overrun error occurs when a newly received character is transferred into the rxdata holding register before the previous character is read (in other words, while the RRDY bit is 1). In this case, the ROE bit is set to 1, and the the previous contents of rxdata are overwritten with the new character. The ROE bit stays set to 1 until it is explicitly cleared by a write to the status register.

 © March 2009 Altera Corporation

Table 6–5. status Register Bits (Part 2 of 2)

Bit	Name	Access	Description
4	TOE	RC	Transmit overrun error. A transmit-overrun error occurs when a new character is written to the txdata holding register before the previous character is transferred into the shift register (in other words, while the TRDY bit is 0). In this case the TOE bit is set to 1. The TOE bit stays set to 1 until it is explicitly cleared by a write to the status register.
5	TMT	R	Transmit empty. The TMT bit indicates the transmitter shift register's current state. When the shift register is in the process of shifting a character out the TXD pin, TMT is set to 0. When the shift register is idle (in other words, a character is not being transmitted) the TMT bit is 1. An Avalon-MM master peripheral can determine if a transmission is completed (and received at the other end of a serial link) by checking the TMT bit.
6	TRDY	R	Transmit ready. The TRDY bit indicates the txdata holding register's current state. When the txdata register is empty, it is ready for a new character, and TRDY is 1. When the txdata register is full, TRDY is 0. An Avalon-MM master peripheral must wait for TRDY to be 1 before writing new data to txdata.
7	RRDY	R	Receive character ready. The RRDY bit indicates the rxdata holding register's current state. When the rxdata register is empty, it is not ready to be read and RRDY is 0. When a newly received value is transferred into the rxdata register, RRDY is set to 1. Reading the rxdata register clears the RRDY bit to 0. An Avalon-MM master peripheral must wait for RRDY to equal 1 before reading the rxdata register.
8	E	RC	Exception. The E bit indicates that an exception condition occurred. The E bit is a logical-OR of the TOE, ROE, BRK, FE, and PE bits. The E bit and its corresponding interrupt-enable bit (IE) bit in the control register provide a convenient method to enable/disable IRQs for all error conditions. The E bit is set to 0 by a write operation to the status register.
10 *(1)*	DCTS	RC	Change in clear to send (CTS) signal. The DCTS bit is set to 1 whenever a logic-level transition is detected on the CTS_N input port (sampled synchronously to the Avalon-MM clock). This bit is set by both falling and rising transitions on CTS_N. The DCTS bit stays set to 1 until it is explicitly cleared by a write to the status register. If the **Flow Control** hardware option is not enabled, the DCTS bit always reads 0. Refer to "Flow Control" on page 6–5.
11 *(1)*	CTS	R	Clear-to-send (CTS) signal. The CTS bit reflects the CTS_N input's instantaneous state (sampled synchronously to the Avalon-MM clock). The CTS_N input has no effect on the transmit or receive processes. The only visible effect of the CTS_N input is the state of the CTS and DCTS bits, and an IRQ that can be generated when the control register's idcts bit is enabled. If the **Flow Control** hardware option is not enabled, the CTS bit always reads 0. Refer to "Flow Control" on page 6–5.
12 *(1)*	EOP	R	End of packet encountered. The EOP bit is set to 1 by one of the following events: ■ An EOP character is written to txdata ■ An EOP character is read from rxdata The EOP character is determined by the contents of the endofpacket register. The EOP bit stays set to 1 until it is explicitly cleared by a write to the status register. If the **Include End-of-Packet Register** hardware option is not enabled, the EOP bit always reads 0. Refer to "Streaming Data (DMA) Control" on page 6–6.

Note to Table 6–5:

(1) This bit is optional and may not exist in hardware.

control Register

The control register consists of individual bits, each controlling an aspect of the UART core's operation. The value in the control register can be read at any time.

Each bit in the control register enables an IRQ for a corresponding bit in the status register. When both a status bit and its corresponding interrupt-enable bit are 1, the core generates an IRQ.

The control register bits are shown in Table 6–6.

Table 6–6. control Register Bits

Bit	Name	Access	Description
0	IPE	RW	Enable interrupt for a parity error.
1	IFE	RW	Enable interrupt for a framing error.
2	IBRK	RW	Enable interrupt for a break detect.
3	IROE	RW	Enable interrupt for a receiver overrun error.
4	ITOE	RW	Enable interrupt for a transmitter overrun error.
5	ITMT	RW	Enable interrupt for a transmitter shift register empty.
6	ITRDY	RW	Enable interrupt for a transmission ready.
7	IRRDY	RW	Enable interrupt for a read ready.
8	IE	RW	Enable interrupt for an exception.
9	TRBK	RW	Transmit break. The TRBK bit allows an Avalon-MM master peripheral to transmit a break character over the TXD output. The TXD signal is forced to 0 when the TRBK bit is set to 1. The TRBK bit overrides any logic level that the transmitter logic would otherwise drive on the TXD output. The TRBK bit interferes with any transmission in process. The Avalon-MM master peripheral must set the TRBK bit back to 0 after an appropriate break period elapses.
10	IDCTS	RW	Enable interrupt for a change in CTS signal.
11 *(1)*	RTS	RW	Request to send (RTS) signal. The RTS bit directly feeds the RTS_N output. An Avalon-MM master peripheral can write the RTS bit at any time. The value of the RTS bit only affects the RTS_N output; it has no effect on the transmitter or receiver logic. Because the RTS_N output is logic negative, when the RTS bit is 1, a low logic-level (0) is driven on the RTS_N output. If the **Flow Control** hardware option is not enabled, the RTS bit always reads 0, and writing has no effect. Refer to "Flow Control" on page 6–5.
12	IEOP	RW	Enable interrupt for end-of-packet condition.

Note to Table 6–6:

(1) This bit is optional and may not exist in hardware.

divisor Register (Optional)

The value in the divisor register is used to generate the baud rate clock. The effective baud rate is determined by the formula:

Baud Rate = (Clock frequency) / (divisor + 1)

The divisor register is an optional hardware feature. If the **Baud Rate Can Be Changed By Software** hardware option is not enabled, the divisor register does not exist. In this case, writing divisor has no effect, and reading divisor returns an undefined value. For more information, refer to "Baud Rate Options" on page 6–4.

endofpacket Register (Optional)

The value in the `endofpacket` register determines the end-of-packet character for variable-length DMA transactions. After reset, the default value is zero, which is the ASCII null character (\0). For more information, refer to Table 6–5 on page 6–12 for the description for the `EOP` bit.

The `endofpacket` register is an optional hardware feature. If the **Include end-of-packet register** hardware option is not enabled, the `endofpacket` register does not exist. In this case, writing `endofpacket` has no effect, and reading returns an undefined value.

Interrupt Behavior

The UART core outputs a single IRQ signal to the Avalon-MM interface, which can connect to any master peripheral in the system, such as a Nios II processor. The master peripheral must read the `status` register to determine the cause of the interrupt.

Every interrupt condition has an associated bit in the `status` register and an interrupt-enable bit in the `control` register. When any of the interrupt conditions occur, the associated `status` bit is set to 1 and remains set until it is explicitly acknowledged. The IRQ output is asserted when any of the status bits are set while the corresponding interrupt-enable bit is 1. A master peripheral can acknowledge the IRQ by clearing the `status` register.

At reset, all interrupt-enable bits are set to 0; therefore, the core cannot assert an IRQ until a master peripheral sets one or more of the interrupt-enable bits to 1.

All possible interrupt conditions are listed with their associated status and control (interrupt-enable) bits in Table 6–5 on page 6–16 and Table 6–6 on page 6–18. Details of each interrupt condition are provided in the `status` bit descriptions.

Referenced Documents

This chapter references the following documents:

- *AN 350: Upgrading Nios Processor Systems to the Nios II Processor*
- *AN 351: Simulating Nios II Embedded Processor Designs*
- *Avalon Interface Specifications*
- *Nios II Software Developer's Handbook*
- *Timer Core* chapter in volume 5 of the *Quartus II Handbook*
- *AN 42: Metastability in Altera Devices*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 6–7 shows the revision history for this chapter.

Table 6–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Added description of a new parameter, **Synchronizer stages**.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

© March 2009 Altera Corporation

7. SPI Core

NII51011-9.0.0

Core Overview

SPI is an industry-standard serial protocol commonly used in embedded systems to connect microprocessors to a variety of off-chip sensor, conversion, memory, and control devices. The SPI core with Avalon® interface implements the SPI protocol and provides an Avalon Memory-Mapped (Avalon-MM) interface on the back end.

The SPI core can implement either the master or slave protocol. When configured as a master, the SPI core can control up to 32 independent SPI slaves. The width of the receive and transmit registers are configurable between 1 and 32 bits. Longer transfer lengths can be supported with software routines. The SPI core provides an interrupt output that can flag an interrupt whenever a transfer completes.

The SPI core is SOPC Builder ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Instantiating the SPI Core in SOPC Builder" on page 7–5
- "Device Support" on page 7–8
- "Software Programming Model" on page 7–8

Functional Description

The SPI core communicates using two data lines, a control line, and a synchronization clock:

- Master Out Slave In (`mosi`)—Output data from the master to the inputs of the slaves
- Master In Slave Out (`miso`)—Output data from a slave to the input of the master
- Serial Clock (`sclk`)—Clock driven by the master to slaves, used to synchronize the data bits
- Slave Select (`ss_n`)— Select signal (active low) driven by the master to individual slaves, used to select the target slave

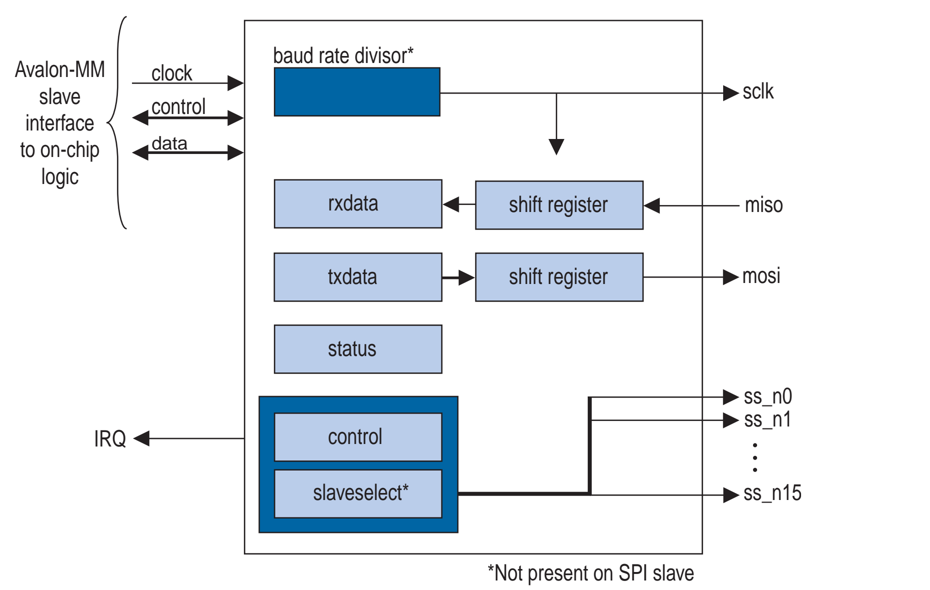
The SPI core has the following user-visible features:

- A memory-mapped register space comprised of five registers: `rxdata`, `txdata`, `status`, `control`, and `slaveselect`
- Four SPI interface ports: `sclk`, `ss_n`, `mosi`, and `miso`

The registers provide an interface to the SPI core and are visible via the Avalon-MM slave port. The `sclk`, `ss_n`, `mosi`, and `miso` ports provide the hardware interface to other SPI devices. The behavior of `sclk`, `ss_n`, `mosi`, and `miso` depends on whether the SPI core is configured as a master or slave.

Figure 7–1 shows a block diagram of the SPI core in master mode.

Figure 7–1. SPI Core Block Diagram



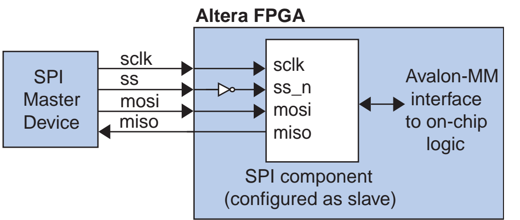
The SPI core logic is synchronous to the clock input provided by the Avalon-MM interface. When configured as a master, the core divides the Avalon-MM clock to generate the SCLK output. When configured as a slave, the core's receive logic is synchronized to SCLK input. The core's Avalon-MM interface is capable of Avalon-MM transfers with flow control. The SPI core can be used in conjunction with a DMA controller with flow control to automate continuous data transfers between, for example, the SPI core and memory.

For more details, refer to the *Interval Timer Core* chapter in volume 5 of the *Quartus II Handbook*.

Example Configurations

Figure 7–1 and Figure 7–2 show two possible configurations. In Figure 7–2, the SPI core provides a slave interface to an off-chip SPI master.

Figure 7–2. SPI Core Configured as a Slave



In Figure 7–1, the SPI core provides a master interface driving multiple off-chip slave devices. Each slave device in Figure 7–1 must tristate its `miso` output whenever its select signal is not asserted.

The ss_n signal is active-low. However, any signal can be inverted inside the FPGA, allowing the slave-select signals to be either active high or active low.

Transmitter Logic

The SPI core transmitter logic consists of a transmit holding register (txdata) and transmit shift register, each *n* bits wide. The register width *n* is specified at system generation time, and can be any integer value from 1 to 32. After a master peripheral writes a value to the txdata register, the value is copied to the shift register and then transmitted when the next operation starts.

The shift register and the txdata register provide double buffering during data transmission. A new value can be written into the txdata register while the previous data is being shifted out of the shift register. The transmitter logic automatically transfers the txdata register to the shift register whenever a serial shift operation is not currently in process.

In master mode, the transmit shift register directly feeds the mosi output. In slave mode, the transmit shift register directly feeds the miso output. Data shifts out LSB first or MSB first, depending on the configuration of the SPI core.

Receiver Logic

The SPI core receive logic consists of a receive holding register (rxdata) and receive shift register, each *n* bits wide. The register width *n* is specified at system generation time, and can be any integer value from 1 to 16. A master peripheral reads received data from the rxdata register after the shift register has captured a full *n*-bit value of data.

The shift register and the rxdata register provide double buffering while receiving data. The rxdata register can hold a previously received data value while subsequent new data is shifting into the shift register. The receiver logic automatically transfers the shift register content to the rxdata register when a serial shift operation completes.

In master mode, the shift register is fed directly by the miso input. In slave mode, the shift register is fed directly by the mosi input. The receiver logic expects input data to arrive LSB first or MSB first, depending on the configuration of the SPI core.

Master and Slave Modes

At system generation time, the designer configures the SPI core in either master mode or slave mode. The mode cannot be switched at runtime.

Master Mode Operation

In master mode, the SPI ports behave as shown in Table 7–1.

Table 7–1. Master Mode Port Configurations (Part 1 of 2)

Name	Direction	Description
mosi	output	Data output to slave(s)
miso	input	Data input from slave(s)

Table 7–1. Master Mode Port Configurations (Part 2 of 2)

Name	Direction	Description
sclk	output	Synchronization clock to all slaves
ss_n*M*	output	Slave select signal to slave *M*, where M is a number between 0 and 15.

In master mode, an intelligent host (for example, a microprocessor) configures the SPI core using the control and slaveselect registers, and then writes data to the txdata buffer to initiate a transaction. A master peripheral can monitor the status of the transaction by reading the status register. A master peripheral can enable interrupts to notify the host whenever new data is received (for example, a transfer has completed), or whenever the transmit buffer is ready for new data.

The SPI protocol is full duplex, so every transaction both sends and receives data at the same time. The master transmits a new data bit on the mosi output and the slave drives a new data bit on the miso input for each active edge of sclk. The SPI core divides the Avalon-MM system clock using a clock divider to generate the sclk signal.

When the SPI core is configured to interface with multiple slaves, the core has one ss_n signal for each slave, up to a maximum of sixteen slaves. During a transfer, the master asserts ss_n to each slave specified in the slaveselect register. Note that there can be no more than one slave transmitting data during any particular transfer, or else there will be a contention on the miso input. The number of slave devices is specified at system generation time.

Slave Mode Operation

In slave mode, the SPI ports behave as shown in Table 7–2.

Table 7–2. Slave Mode Port Configurations

Name	Direction	Description
mosi	input	Data input from the master
miso	output	Data output to the master
sclk	input	Synchronization clock
ss_n	input	Select signal

In slave mode, the SPI core simply waits for the master to initiate transactions. Before a transaction begins, the slave logic continuously polls the ss_n input. When the master asserts ss_n, the slave logic immediately begins sending the transmit shift register contents to the miso output. The slave logic also captures data on the mosi input, and fills the receive shift register simultaneously.

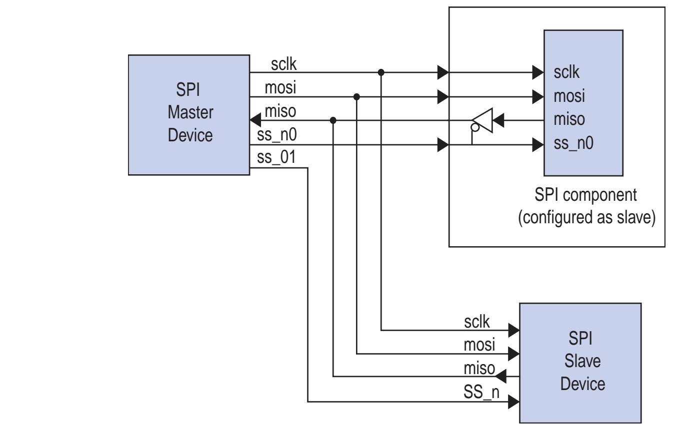
An intelligent host such as a microprocessor writes data to the txdata registers, so that it is transmitted the next time the master initiates an operation. A master peripheral reads received data from the rxdata register. A master peripheral can enable interrupts to notify the host whenever new data is received, or whenever the transmit buffer is ready for new data.

© March 2009 Altera Corporation

Multi-Slave Environments

When ss_n is not asserted, typical SPI cores set their miso output pins to high impedance. The Altera®-provided SPI slave core drives an undefined high or low value on its miso output when not selected. Special consideration is necessary to avoid signal contention on the miso output, if the SPI core in slave mode is connected to an off-chip SPI master device with multiple slaves. In this case, the ss_n input should be used to control a tristate buffer on the miso signal. Figure 7–3 shows an example of the SPI core in slave mode in an environment with two slaves.

Figure 7–3. SPI Core in a Multi-Slave Environment



Avalon-MM Interface

The SPI core's Avalon-MM interface consists of a single Avalon-MM slave port. In addition to fundamental slave read and write transfers, the SPI core supports Avalon-MM read and write transfers with flow control.

Instantiating the SPI Core in SOPC Builder

You can use the MegaWizard™ interface for the SPI core in SOPC Builder to configure the hardware feature set. The following sections describe the available options.

Master/Slave Settings

The designer can select either master mode or slave mode to determine the role of the SPI core. When master mode is selected, the following options are available: **Number of select (SS_n) signals, SPI clock rate**, and **Specify delay**.

Number of Select (SS_n) Signals

This setting specifies how many slaves the SPI master connects to. The range is 1 to 32. The SPI master core presents a unique ss_n signal for each slave.

SPI Clock (sclk) Rate

This setting determines the rate of the sclk signal that synchronizes data between master and slaves. The target clock rate can be specified in units of Hz, kHz or MHz. The SPI master core uses the Avalon-MM system clock and a clock divisor to generate sclk.

The actual frequency of `sclk` may not exactly match the desired target clock rate. The achievable clock values are:

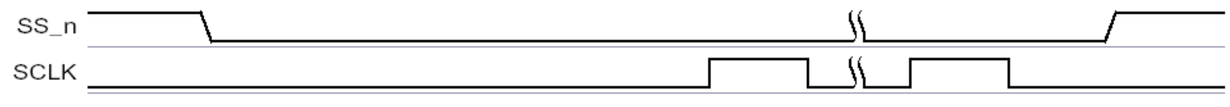
<Avalon-MM system clock frequency> / [2, 4, 6, 8, ...]

The actual frequency achieved will not be greater than the specified target value. For example, if the system clock frequency is 50 MHz and the target value is 25 MHz, the clock divisor is 2 and the actual `sclk` frequency achieves exactly 25 MHz.

Specify Delay

Turning on this option causes the SPI master to add a time delay between asserting the `ss_n` signal and shifting the first bit of data. This delay is required by certain SPI slave devices. If the delay option is on, you must also specify the delay time in units of ns, µs or ms. An example is shown in Figure 7–4.

Figure 7–4. Time Delay Between Asserting ss_n and Toggling sclk



The delay generation logic uses a granularity of half the period of `sclk`. The actual delay achieved is the desired target delay rounded up to the nearest multiple of half the `sclk` period, as shown in Equation 7–1 and Equation 7–2:

Equation 7–1.

$$p = \frac{1}{2} \times (\text{period of sclk})$$

Equation 7–2.

$$\text{actual delay} = \text{ceiling} \times \left(\frac{\text{desired delay}}{p}\right) \times p$$

Data Register Settings

The data register settings affect the size and behavior of the data registers in the SPI core. There are two data register settings:

- **Width**—This setting specifies the width of `rxdata`, `txdata`, and the receive and transmit shift registers. The range is from 1 to 32.
- **Shift direction**—This setting determines the direction that data shifts (MSB first or LSB first) into and out of the shift registers.

Timing Settings

The timing settings affect the timing relationship between the `ss_n`, `sclk`, `mosi` and `miso` signals. In this discussion the `mosi` and `miso` signals are referred to generically as *data*. There are two timing settings:

- **Clock polarity**—This setting can be 0 or 1. When clock polarity is set to 0, the idle state for `sclk` is low. When clock polarity is set to 1, the idle state for `sclk` is high.
- **Clock phase**—This setting can be 0 or 1. When clock phase is 0, data is latched on the leading edge of `sclk`, and data changes on trailing edge. When clock phase is 1, data is latched on the trailing edge of `sclk`, and data changes on the leading edge.

Figure 7–5 through Figure 7–8 demonstrate the behavior of signals in all possible cases of clock polarity and clock phase.

Figure 7–5. Clock Polarity = 0, Clock Phase = 0

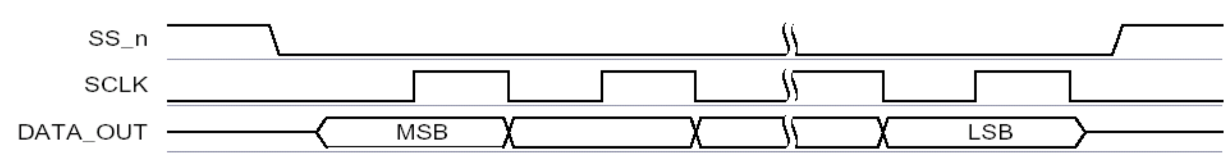


Figure 7–6. Clock Polarity = 0, Clock Phase = 1

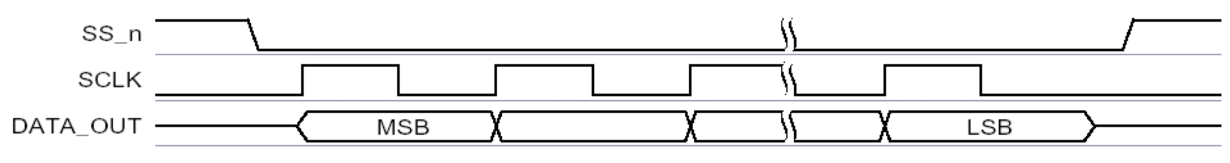


Figure 7–7. Clock Polarity = 1, Clock Phase = 0

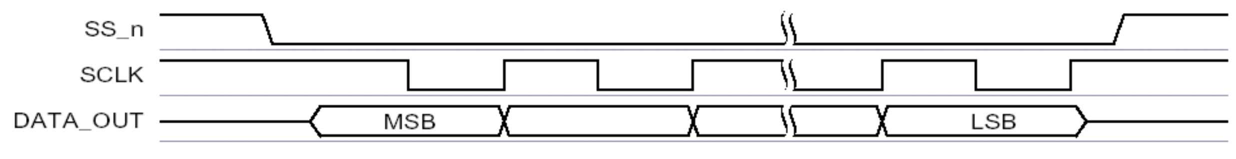
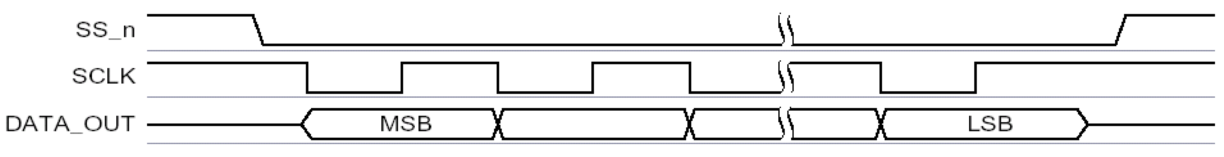


Figure 7–8. Clock Polarity = 1, Clock Phase = 1



Device Support

The SPI core supports all Altera® device families.

Software Programming Model

The following sections describe the software programming model for the SPI core, including the register map and software constructs used to access the hardware. For Nios® II processor users, Altera provides the HAL system library header file that defines the SPI core registers. The SPI core does not match the generic device model categories supported by the HAL, so it cannot be accessed via the HAL API or the ANSI C standard library. Altera provides a routine to access the SPI hardware that is specific to the SPI core.

Hardware Access Routines

Altera provides one access routine, `alt_avalon_spi_command()`, that provides general-purpose access to an SPI core configured as a master.

alt_avalon_spi_command()

Prototype:

```
int alt_avalon_spi_command(alt_u32 base, alt_u32 slave,
                           alt_u32 write_length,
                           const alt_u8* wdata,
                           alt_u32 read_length,
                           alt_u8* read_data,
                           alt_u32 flags)
```

Thread-safe: No.

Available from ISR: No.

Include: <altera_avalon_spi.h>

Description: `alt_avalon_spi_command()` is used to perform a control sequence on the SPI bus. This routine is designed for SPI masters of 8-bit data width or less. Currently, it does not support SPI hardware with data-width greater than 8 bits. A single call to this function writes a data buffer of arbitrary length out the MOSI port, and then reads back an arbitrary amount of data from the MISO port. The function performs the following actions:

(1) Asserts the slave select output for the specified slave. The first slave select output is numbered 0, the next is 1, etc.

(2) Transmits `write_length` bytes of data from `wdata` through the SPI interface, discarding the incoming data on MISO.

(3) Reads `read_length` bytes of data, storing the data into the buffer pointed to by `read_data`. MOSI is set to zero during the read transaction.

(4) De-asserts the slave select output, unless the flags field contains the value `ALT_AVALON_SPI_COMMAND_MERGE`. If you want to transmit from scattered buffers then you can call the function multiple times, specifying the merge flag on all the accesses except the last.

This function is not thread safe. If you want to access the SPI bus from more than one thread, you should use a semaphore or mutex to ensure that only one thread is executing within this function at any time.

Returns: The number of bytes stored in the `read_data` buffer.

Software Files

The SPI core is accompanied by the following software files. These files provide a low-level interface to the hardware.

- **altera_avalon_spi.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_spi.c**—This file implements low-level routines to access the hardware.

Register Map

An Avalon-MM master peripheral controls and communicates with the SPI core via the six 32-bit registers, shown in Table 7–3. The table assumes an *n*-bit data width for `rxdata` and `txdata`.

Table 7–3. Register Map for SPI Master Device

Internal Address	Register Name	32..11	10	9	8	7	6	5	4	3	2	1	0
0	`rxdata` *(1)*	`RXDATA` (n-1..0)											
1	`txdata` *(1)*	`TXDATA` (n-1..0)											
2	`status` *(2)*				`E`	`RRDY`	`TRDY`	`TMT`	`TOE`	`ROE`			
3	`control`		`SSO` *(3)*		`IE`	`IRRDY`	`ITRDY`		`ITOE`	`IROE`			
4	Reserved												
5	`slaveselect` *(3)*	Slave Select Mask											

Notes to Table 7–3:

(1) Bits 15 to *n* are undefined when *n* is less than 16.

(2) A write operation to the `status` register clears the `ROE`, `TOE`, and `E` bits.

(3) Present only in master mode.

Reading undefined bits returns an undefined value. Writing to undefined bits has no effect.

rxdata Register

A master peripheral reads received data from the `rxdata` register. When the receive shift register receives a full *n* bits of data, the `status` register's `RRDY` bit is set to 1 and the data is transferred into the `rxdata` register. Reading the `rxdata` register clears the `RRDY` bit. Writing to the `rxdata` register has no effect.

New data is always transferred into the `rxdata` register, whether or not the previous data was retrieved. If `RRDY` is 1 when data is transferred into the `rxdata` register (that is, the previous data was not retrieved), a receive-overrun error occurs and the `status` register's `ROE` bit is set to 1. In this case, the contents of `rxdata` are undefined.

txdata Register

A master peripheral writes data to be transmitted into the `txdata` register. When the `status` register's `TRDY` bit is 1, it indicates that the `txdata` register is ready for new data. The `TRDY` bit is set to 0 whenever the `txdata` register is written. The `TRDY` bit is set to 1 after data is transferred from the `txdata` register into the transmitter shift register, which readies the `txdata` holding register to receive new data.

A master peripheral should not write to the `txdata` register until the transmitter is ready for new data. If `TRDY` is 0 and a master peripheral writes new data to the `txdata` register, a transmit-overrun error occurs and the `status` register's `TOE` bit is set to 1. In this case, the new data is ignored, and the content of `txdata` remains unchanged.

© March 2009 Altera Corporation

As an example, assume that the SPI core is idle (that is, the txdata register and transmit shift register are empty), when a CPU writes a data value into the txdata holding register. The TRDY bit is set to 0 momentarily, but after the data in txdata is transferred into the transmitter shift register, TRDY returns to 1. The CPU writes a second data value into the txdata register, and again the TRDY bit is set to 0. This time the shift register is still busy transferring the original data value, so the TRDY bit remains at 0 until the shift operation completes. When the operation completes, the second data value is transferred into the transmitter shift register and the TRDY bit is again set to 1.

status Register

The status register consists of bits that indicate status conditions in the SPI core. Each bit is associated with a corresponding interrupt-enable bit in the control register, as discussed in "control Register" on page 7–12. A master peripheral can read status at any time without changing the value of any bits. Writing status does clear the ROE, TOE and E bits. Table 7–4 describes the individual bits of the status register.

Table 7–4. status Register Bits

#	Name	Description
3	ROE	**Receive-overrun error** The ROE bit is set to 1 if new data is received while the rxdata register is full (that is, while the RRDY bit is 1). In this case, the new data overwrites the old. Writing to the status register clears the ROE bit to 0.
4	TOE	**Transmitter-overrun error** The TOE bit is set to 1 if new data is written to the txdata register while it is still full (that is, while the TRDY bit is 0). In this case, the new data is ignored. Writing to the status register clears the TOE bit to 0.
5	TMT	**Transmitter shift-register empty** In master mode, the TMT bit is set to 0 when a transaction is in progress and set to 1 when the shift register is empty. In slave mode, the TMT bit is set to 0 when the slave is selected (SS_n is low) or when the SPI Slave register interface is not ready to receive data.
6	TRDY	**Transmitter ready** The TRDY bit is set to 1 when the txdata register is empty.
7	RRDY	**Receiver ready** The RRDY bit is set to 1 when the rxdata register is full.
8	E	**Error** The E bit is the logical OR of the TOE and ROE bits. This is a convenience for the programmer to detect error conditions. Writing to the status register clears the E bit to 0.

control Register

The control register consists of data bits to control the SPI core's operation. A master peripheral can read control at any time without changing the value of any bits.

Most bits (IROE, ITOE, ITRDY, IRRDY, and IE) in the control register control interrupts for status conditions represented in the status register. For example, bit 1 of status is ROE (receiver-overrun error), and bit 1 of control is IROE, which enables interrupts for the ROE condition. The SPI core asserts an interrupt request when the corresponding bits in status and control are both 1.

The control register bits are shown in Table 7–5.

Table 7–5. control Register Bits

#	Name	Description
3	IROE	Setting IROE to 1 enables interrupts for receive-overrun errors.
4	ITOE	Setting ITOE to 1 enables interrupts for transmitter-overrun errors.
6	ITRDY	Setting ITRDY to 1 enables interrupts for the transmitter ready condition.
7	IRRDY	Setting IRRDY to 1 enables interrupts for the receiver ready condition.
8	IE	Setting IE to 1 enables interrupts for any error condition.
10	SSO	Setting SSO to 1 forces the SPI core to drive its ss_n outputs, regardless of whether a serial shift operation is in progress or not. The slaveselect register controls which ss_n outputs are asserted. SSO can be used to transmit or receive data of arbitrary size, for example, greater than 32 bits.

After reset, all bits of the control register are set to 0. All interrupts are disabled and no ss_n signals are asserted.

slaveselect Register

The slaveselect register is a bit mask for the ss_n signals driven by an SPI master. During a serial shift operation, the SPI master selects only the slave device(s) specified in the slaveselect register.

The slaveselect register is only present when the SPI core is configured in master mode. There is one bit in slaveselect for each ss_n output, as specified by the designer at system generation time.

A master peripheral can set multiple bits of slaveselect simultaneously, causing the SPI master to simultaneously select multiple slave devices as it performs a transaction. For example, to enable communication with slave devices 1, 5, and 6, set bits 1, 5, and 6 of slaveselect. However, consideration is necessary to avoid signal contention between multiple slaves on their miso outputs.

Upon reset, bit 0 is set to 1, and all other bits are cleared to 0. Thus, after a device reset, slave device 0 is automatically selected.

Referenced Documents

This chapter references the following documents:

- *AN 350: Upgrading Nios Processor Systems to the Nios II Processor*
- *Interval Timer Core* chapter in volume 5 of the *Quartus II Handbook*

 © March 2009 Altera Corporation

Document Revision History

Table 7–6 shows the revision history for this chapter.

Table 7–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. Updated the width of the parameters and signals from 16 to 32.	—
May 2008 v8.0.0	Updated the description of the `TMT` bit.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

8. Optrex 16207 LCD Controller Core

NII51019-9.0.0

Core Overview

The Optrex 16207 LCD controller core with Avalon® Interface (LCD controller core) provides the hardware interface and software driver required for a Nios® II processor to display characters on an Optrex 16207 (or equivalent) 16×2-character LCD panel. Device drivers are provided in the HAL system library for the Nios II processor. Nios II programs access the LCD controller as a character mode device using ANSI C standard library routines, such as `printf()`. The LCD controller is SOPC Builder-ready, and integrates easily into any SOPC Builder-generated system.

The Nios II Embedded Design Suite (EDS) includes an Optrex LCD module and provide several ready-made example designs that display text on the Optrex 16207 via the LCD controller. For details about the Optrex 16207 LCD module, see the manufacturer's *Dot Matrix Character LCD Module User's Manual* available at www.optrex.com.

This chapter contains the following sections:

- "Functional Description"
- "Device and Tools Support" on page 8–2
- "Instantiating the Core in SOPC Builder" on page 8–2
- "Software Programming Model" on page 8–2

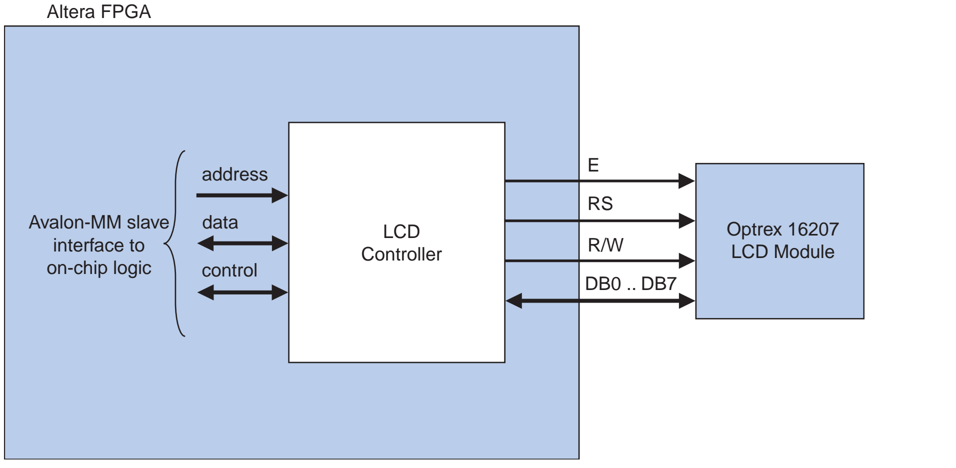
Functional Description

The LCD controller core consists of two user-visible components:

- Eleven signals that connect to pins on the Optrex 16207 LCD panel—These signals are defined in the Optrex 16207 data sheet.
  - `E`—Enable (output)
  - `RS`—Register Select (output)
  - `R/W`—Read or Write (output)
  - `DB0` through `DB7`—Data Bus (bidirectional)
- An Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to 4 registers.

Figure 8–1 shows a block diagram of the LCD controller core.

Figure 8–1. LCD Controller Block Diagram



Device and Tools Support

The LCD controller core supports all Altera device families. The LCD controller drivers support the Nios II processor.

Instantiating the Core in SOPC Builder

You can add the LCD controller core from the **System Contents** tab in SOPC Builder. In SOPC Builder, the LCD controller core has the name Character LCD (16×2, Optrex 16207). There are no user-configurable settings for this component.

Software Programming Model

This section describes the software programming model for the LCD controller.

HAL System Library Support

Altera provides HAL system library drivers for the Nios II processor that enable you to access the LCD controller using the ANSI C standard library functions. The Altera-provided drivers integrate into the HAL system library for Nios II systems. The LCD driver is a standard character-mode device, as described in the *Nios II Software Developer's Handbook*. Therefore, using `printf()` is the easiest way to write characters to the display.

The LCD driver requires that the HAL system library include the system clock driver.

Displaying Characters on the LCD

The driver implements VT100 terminal-like behavior on a miniature scale for the 16×2 screen. Characters written to the LCD controller are stored to an 80-column × 2-row buffer maintained by the driver. As characters are written, the cursor position is updated. Visible characters move the cursor position to the right. Any visible characters written to the right of the buffer are discarded. The line feed character (`\n`) moves the cursor down one line and to the left-most column.

The buffer is scrolled up as soon as a printable character is written onto the line below the bottom of the buffer. Rows do not scroll as soon as the cursor moves down to allow the maximum useful information in the buffer to be displayed.

If the visible characters in the buffer fit on the display, all characters are displayed. If the buffer is wider than the display, the display scrolls horizontally to display all the characters. Different lines scroll at different speeds, depending on the number of characters in each line of the buffer.

The LCD driver supports a small subset of ANSI and VT100 escape sequences that can be used to control the cursor position, and clear the display as shown in Table 8–1.

Table 8–1. Escape Sequence Supported by the LCD Controller

Sequence	Meaning
`BS (\b)`	Moves the cursor to the left by one character.
`CR (\r)`	Moves the cursor to the start of the current line.
`LF (\n)`	Moves the cursor to the start of the line and move it down one line.
`ESC( (\x1B)`	Starts a VT100 control sequence.
`ESC [ <y> ; <x> H`	Moves the cursor to the y, x position specified – positions are counted from the top left which is 1;1.
`ESC [ K`	Clears from current cursor position to end of line.
`ESC [ 2 J`	Clears the whole screen.

The LCD controller is an output-only device. Therefore, attempts to read from it returns immediately indicating that no characters have been received.

The LCD controller drivers are not included in the system library when the **Reduced device drivers** option is enabled for the system library. If you want to use the LCD controller while using small drivers for other devices, add the preprocessor option—`DALT_USE_LCD_16207` to the preprocessor options.

Software Files

The LCD controller is accompanied by the following software files. These files define the low-level interface to the hardware and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_lcd_16207_regs.h** — This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_lcd_16207.h**, **altera_avalon_lcd_16207.c** — These files implement the LCD controller device drivers for the HAL system library.

Register Map

The HAL device drivers make it unnecessary for you to access the registers directly. Therefore, Altera does not publish details about the register map. For more information, the **altera_avalon_lcd_16207_regs.h** file describes the register map, and the *Dot Matrix Character LCD Module User's Manual* from Optrex describes the register usage.

Interrupt Behavior

The LCD controller does not generate interrupts. However, the LCD driver's text scrolling feature relies on the HAL system clock driver, which uses interrupts for timing purposes.

Referenced Documents

This chapter references the *Nios II Software Developer's Handbook*.

Document Revision History

Table 8–2 shows the revision history for this chapter.

Table 8–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

9. PIO Core

NII51007-9.0.0

Core Overview

The parallel input/output (PIO) core with Avalon® interface provides a memory-mapped interface between an Avalon® Memory-Mapped (Avalon-MM) slave port and general-purpose I/O ports. The I/O ports connect either to on-chip user logic, or to I/O pins that connect to devices external to the FPGA.

The PIO core provides easy I/O access to user logic or external devices in situations where a "bit banging" approach is sufficient. Some example uses are:

- Controlling LEDs
- Acquiring data from switches
- Controlling display devices
- Configuring and communicating with off-chip devices, such as application-specific standard products (ASSP)

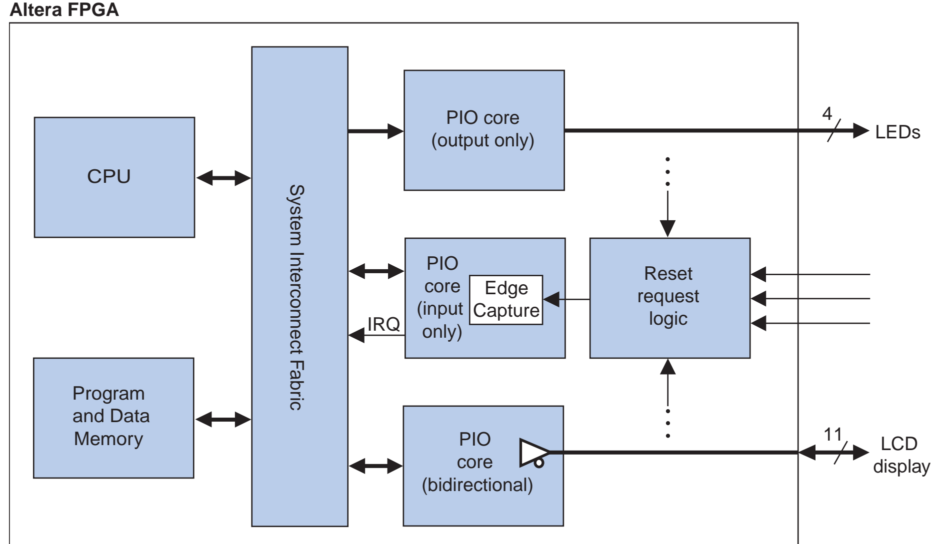
The PIO core interrupt request (IRQ) output can assert an interrupt based on input signals. The PIO core is SOPC Builder ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Example Configurations" on page 9–3
- "Instantiating the PIO Core in SOPC Builder" on page 9–4
- "Device Support" on page 9–5
- "Software Programming Model" on page 9–5

Functional Description

Each PIO core can provide up to 32 I/O ports. An intelligent host such as a microprocessor controls the PIO ports by reading and writing the register-mapped Avalon-MM interface. Under control of the host, the PIO core captures data on its inputs and drives data to its outputs. When the PIO ports are connected directly to I/O pins, the host can tristate the pins by writing control registers in the PIO core. Figure 9–1 shows an example of a processor-based system that uses multiple PIO cores to drive LEDs, capture edges from on-chip reset-request control logic, and control an off-chip LCD display.

Figure 9–1. An Example System Using Multiple PIO Cores



When integrated into an SOPC Builder-generated system, the PIO core has two user-visible features:

- A memory-mapped register space with four registers: `data`, `direction`, `interruptmask`, and `edgecapture`
- 1 to 32 I/O ports

The I/O ports can be connected to logic inside the FPGA, or to device pins that connect to off-chip devices. The registers provide an interface to the I/O ports via the Avalon-MM interface. See Table 9–2 on page 9–6 for a description of the registers.

Data Input and Output

The PIO core I/O ports can connect to either on-chip or off-chip logic. The core can be configured with inputs only, outputs only, or both inputs and outputs. If the core is used to control bidirectional I/O pins on the device, the core provides a bidirectional mode with tristate control.

The hardware logic is separate for reading and writing the data register. Reading the data register returns the value present on the input ports (if present). Writing data affects the value driven to the output ports (if present). These ports are independent; reading the data register does not return previously-written data.

Edge Capture

The PIO core can be configured to capture edges on its input ports. It can capture low-to-high transitions, high-to-low transitions, or both. Whenever an input detects an edge, the condition is indicated in the `edgecapture` register. The types of edges detected is specified at system generation time, and cannot be changed via the registers.

© March 2009 Altera Corporation

IRQ Generation

The PIO core can be configured to generate an IRQ on certain input conditions. The IRQ conditions can be either:

- *Level-sensitive*—The PIO core hardware can detect a high level. A NOT gate can be inserted external to the core to provide negative sensitivity.
- *Edge-sensitive*—The core's edge capture configuration determines which type of edge causes an IRQ

Interrupts are individually maskable for each input port. The interrupt mask determines which input port can generate interrupts.

Example Configurations

Figure 9–2 shows a block diagram of the PIO core configured with input and output ports, as well as support for IRQs.

Figure 9–2. PIO Core with Input Ports, Output Ports, and IRQ Support

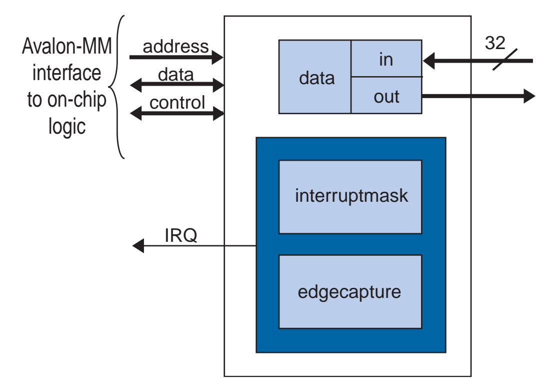
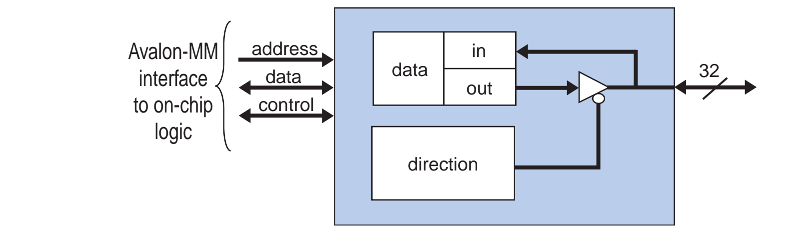


Figure 9–3 shows a block diagram of the PIO core configured in bidirectional mode, without support for IRQs.

Figure 9–3. PIO Core with Bidirectional Ports



Avalon-MM Interface

The PIO core's Avalon-MM interface consists of a single Avalon-MM slave port. The slave port is capable of fundamental Avalon-MM read and write transfers. The Avalon-MM slave port provides an IRQ output so that the core can assert interrupts.

Instantiating the PIO Core in SOPC Builder

Use the MegaWizard™ interface for the PIO core in SOPC Builder to configure the core. The following sections describe the available options.

Basic Settings

The **Basic Settings** page allows you to specify the width, direction and reset value of the I/O ports.

Width

The width of the I/O ports can be set to any integer value between 1 and 32.

Direction

You can set the port direction to one of the options shown in Table 9–1.

Table 9–1. Direction Settings

Setting	Description
Bidirectional (tristate) ports	In this mode, each PIO bit shares one device pin for driving and capturing data. The direction of each pin is individually selectable. To tristate an FPGA I/O pin, set the direction to input.
Input ports only	In this mode the PIO ports can capture input only.
Output ports only	In this mode the PIO ports can drive output only.
Both input and output ports	In this mode, the input and output ports buses are separate, unidirectional buses of *n* bits wide.

Output Port Reset Value

You can specify the reset value of the output ports. The range of legal values depends on the port width.

Output Register

The option **Enable individual bit set/clear output register** allows you to set or clear individual bits of the output port. When this option is turned on, two additional registers—`outset` and `outclear`—are implemented. You can use these registers to specify the output bit to set and clear.

Input Options

The **Input Options** page allows you to specify edge-capture and IRQ generation settings. The **Input Options** page is not available when **Output ports only** is selected on the **Basic Settings** page.

Edge Capture Register

Turn on **Synchronously capture** to include the edge capture register, `edgecapture`, in the core. The edge capture register allows the core to detect and generate an optional interrupt when an edge of the specified type occurs on an input port. The user must further specify the following features:

- Select the type of edge to detect:
  - **Rising Edge**
  - **Falling Edge**
  - **Either Edge**
- Turn on **Enable bit-clearing for edge capture register** to clear individual bit in the edge capture register. To clear a given bit, write 1 to the bit in the edge capture register.

Interrupt

Turn on **Generate IRQ** to assert an IRQ output when a specified event occurs on input ports. The user must further specify the cause of an IRQ event:

- **Level—**The core generates an IRQ whenever a specific input is high and interrupts are enabled for that input in the `interruptmask` register.
- **Edge**—The core generates an IRQ whenever a specific bit in the edge capture register is high and interrupts are enabled for that bit in the `interruptmask` register.

When **Generate IRQ** is off, the `interruptmask` register does not exist.

Simulation

The **Simulation** page allows you to specify the value of the input ports during simulation. Turn on **Hardwire PIO inputs in test bench** to set the PIO input ports to a certain value in the testbench, and specify the value in **Drive inputs to** field.

Device Support

The PIO core supports all Altera® device families.

Software Programming Model

This section describes the software programming model for the PIO core, including the register map and software constructs used to access the hardware. For Nios® II processor users, Altera provides the HAL system library header file that defines the PIO core registers. The PIO core does not match the generic device model categories supported by the HAL, so it cannot be accessed via the HAL API or the ANSI C standard library.

The Nios II Embedded Design Suite (EDS) provides several example designs that demonstrate usage of the PIO core. In particular, the **count_binary.c** example uses the PIO core to drive LEDs, and detect button presses using PIO edge-detect interrupts.

Software Files

The PIO core is accompanied by one software file, **altera_avalon_pio_regs.h**. This file defines the core's register map, providing symbolic constants to access the low-level hardware.

Register Map

An Avalon-MM master peripheral, such as a CPU, controls and communicates with the PIO core via the four 32-bit registers, shown in Table 9–2. The table assumes that the PIO core's I/O ports are configured to a width of *n* bits.

Table 9–2. Register Map for the PIO Core

Offset	Register Name		R/W	(n-1)	...	2	1	0
0	`data`	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				
1	`direction` *(1)*		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	`interruptmask` *(1)*		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	`edgecapture` *(1)*, *(2)*		R/W	Edge detection for each input port.				
4	`outset`		W	Specifies which bit of the output port to set.				
5	`outclear`		W	Specifies which output bit to clear.				

Notes to Table 9–2:

(1) This register may not exist, depending on the hardware configuration. If a register is not present, reading the register returns an undefined value, and writing the register has no effect.

(2) Writing any value to `edgecapture` clears all bits to 0.

data Register

Reading from `data` returns the value present at the input ports. If the PIO core hardware is configured in output-only mode, reading from `data` returns an undefined value.

Writing to `data` stores the value to a register that drives the output ports. If the PIO core hardware is configured in input-only mode, writing to `data` has no effect. If the PIO core hardware is in bidirectional mode, the registered value appears on an output port only when the corresponding bit in the `direction` register is set to 1 (output).

direction Register

The `direction` register controls the data direction for each PIO port, assuming the port is bidirectional. When bit *n* in `direction` is set to 1, port *n* drives out the value in the corresponding bit of the `data` register.

The `direction` register only exists when the PIO core hardware is configured in bidirectional mode. The mode (input, output, or bidirectional) is specified at system generation time, and cannot be changed at runtime. In input-only or output-only mode, the `direction` register does not exist. In this case, reading `direction` returns an undefined value, writing `direction` has no effect.

After reset, all bits of direction are 0, so that all bidirectional I/O ports are configured as inputs. If those PIO ports are connected to device pins, the pins are held in a high-impedance state. In bi-directional mode, to change the direction of the PIO port, reprogram the `direction` register.

interruptmask Register

Setting a bit in the `interruptmask` register to 1 enables interrupts for the corresponding PIO input port. Interrupt behavior depends on the hardware configuration of the PIO core. See "Interrupt Behavior" on page 9–7.

The `interruptmask` register only exists when the hardware is configured to generate IRQs. If the core cannot generate IRQs, reading `interruptmask` returns an undefined value, and writing to `interruptmask` has no effect.

After reset, all bits of `interruptmask` are zero, so that interrupts are disabled for all PIO ports.

edgecapture Register

Bit *n* in the `edgecapture` register is set to 1 whenever an edge is detected on input port *n*. An Avalon-MM master peripheral can read the `edgecapture` register to determine if an edge has occurred on any of the PIO input ports. If the option **Enable bit-clearing for edge capture register** is turned off, writing any value to the `edgecapture` register clears all bits in the register. Otherwise, writing a 1 to a particular bit in the register clears only that bit.

The type of edge(s) to detect is fixed in hardware at system generation time. The `edgecapture` register only exists when the hardware is configured to capture edges. If the core is not configured to capture edges, reading from `edgecapture` returns an undefined value, and writing to `edgecapture` has no effect.

outset and outclear Registers

You can use the `outset` and `outclear` registers to set and clear individual bits of the output port. For example, to set bit 6 of the output port, write `0x40` to the `outset` register. Writing `0x08` to the `outclear` register clears bit 3 of the output port.

These registers are only present when the option **Enable individual bit set/clear output register** is turned on.

Interrupt Behavior

The PIO core outputs a single IRQ signal that can connect to any master peripheral in the system. The master can read either the `data` register or the `edgecapture` register to determine which input port caused the interrupt.

When the hardware is configured for level-sensitive interrupts, the IRQ is asserted whenever corresponding bits in the `data` and `interruptmask` registers are 1. When the hardware is configured for edge-sensitive interrupts, the IRQ is asserted whenever corresponding bits in the `edgecapture` and `interruptmask` registers are 1. The IRQ remains asserted until explicitly acknowledged by disabling the appropriate bit(s) in `interruptmask`, or by writing to `edgecapture`.

Software Files

The PIO core is accompanied by the following software file. This file provide low-level access to the hardware. Application developers should not modify the file.

- **altera_avalon_pio_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used by device driver functions.

Document Revision History

Table 9–3 shows the revision history for this chapter.

Table 9–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Added a section on new registers, `outset` and `outclear`.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. Added the description for **Output Port Reset Value** and **Simulation** parameters.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

 © March 2009 Altera Corporation

10. Avalon-ST JTAG Interface Core

QII55008-9.0.0

Core Overview

The Avalon® Streaming (Avalon-ST) JTAG Interface core enables communication between SOPC Builder systems and JTAG hosts via Avalon-ST interface. Data is serially transferred on the JTAG interface, and presented on the Avalon-ST interface as bytes.

The SPI Slave/JTAG to Avalon Master Bridge is an example of how this core is used. For more information about the bridge, refer to the *SPI Slave/JTAG to Avalon Master Bridge Cores* chapter in volume 5 of the *Quartus II Handbook*.

The Avalon-ST JTAG Interface core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated systems.

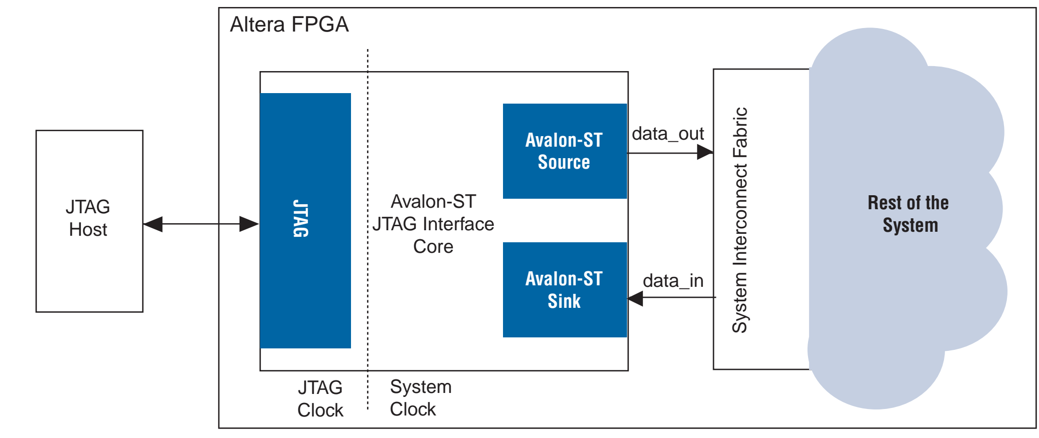
This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 10–3
- "Device Support" on page 10–3

Functional Description

Figure 10–1 shows a block diagram of the Avalon-ST JTAG Interface core in a typical system configuration.

Figure 10–1. SOPC Builder System with an Avalon-ST JTAG Interface Core



Interfaces

Table 10–1 shows the properties of the Avalon-ST interfaces.

Table 10–1. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Only supported on the sink interface.
Data Width	Data width = 8 bits; Bits per symbol = 8.
Channel	Not supported.
Error	Not used.
Packet	Not supported.

For more information about Avalon-ST interfaces, refer to the *Avalon Interface Specifications*.

Special characters

Table 10–2 lists the special characters recognized by the core.

Table 10–2. Special Characters

Character	Description
`0x4a`	Idle. Idle characters are inserted into data streams when there is no data to send.
`0x4d`	Idle escape. An idle escape character is inserted into data stream when the data to send is a special character, followed by the data which is XORed with `0x20`.

Operation

The Avalon-ST JTAG Interface core accepts incoming data in bits on its JTAG interface and packs the bits into bytes. After each byte is formed, the core checks for the following special characters:

- `0x4a`—Idle character. The core drops the idle character.
- `0x4d`—Escape character. The core drops the escape character, and `XOR`s the following byte with `0x20`.

Each valid byte is then transferred to the core's Avalon-ST source interface. As there are no means to backpressure this interface, you must ensure that sufficient storage is in place to avoid data loss.

In the opposite direction, the core serializes each byte received on its Avalon-ST sink interface and sends the bits to the JTAG interface. If there is no data on the sink interface, the core sends out idle characters. If the data is a special character, the core inserts an escape character and `XOR`s the data with `0x20`.

The core supports four operation modes. From the system console, you can set the instruction register (IR) to enable the following supported modes:

- Normal mode—The core works as a bridge between a JTAG host and an SOPC Builder system. Set the IR to 0 to enable this mode.
- Loopback—Data received by the core is sent back to the host. Set the IR to 1 to enable this mode.

- Troubleshoot—The core retrieves the value of the system reset and clock signals, and return them to the JTAG host. Set the IR to 2 to enable this mode.

A TimeQuest SDC file (**.sdc**) is provided to cut any paths internal to the core.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the Avalon-ST JTAG Interface core in SOPC Builder to add the core to a system. There are no user-configurable parameters for this core.

Device Support

The Avalon-ST JTAG Interface core supports all Altera® device families.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *SPI Slave/JTAG to Avalon Master Bridge Cores* chapter in volume 5 of the *Quartus II Handbook*

Document Revision History

Table 10–3 shows the revision history for this chapter.

Table 10–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

11. Avalon-ST Serial Peripheral Interface Core

QII55009-9.0.0

Core Overview

The Avalon® Streaming (Avalon-ST) Serial Peripheral Interface (SPI) core is an SPI slave that allows data transfers between SOPC Builder systems and off-chip SPI devices via Avalon-ST interfaces. Data is serially transferred on the SPI, and sent to and received from the Avalon-ST interface in bytes.

The SPI Slave to Avalon Master Bridge is an example of how this core is used. For more information on the bridge, refer to the *SPI Slave/JTAG to Avalon Master Bridge Cores* chapter in volume 5 of the *Quartus II Handbook*.

The Avalon-ST Serial Peripheral Interface core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system.

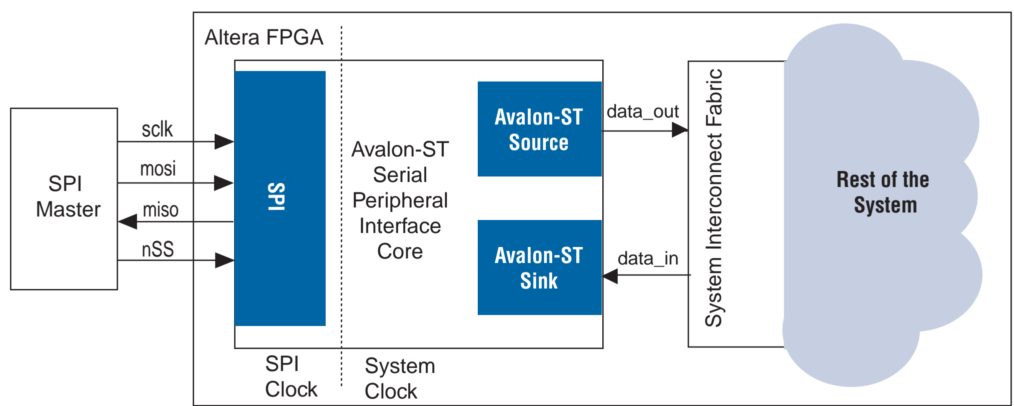
This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 11–3
- "Device Support" on page 11–3

Functional Description

Figure 11–1 shows a block diagram of the Avalon-ST Serial Peripheral Interface core in a typical system configuration.

Figure 11–1. SOPC Builder System with an Avalon-ST SPI Core



Interfaces

The serial peripheral interface is full-duplex and does not support backpressure. It supports SPI clock phase bit, CPHA = 1, and SPI clock polarity bit, CPOL = 0.

Table 11–1 shows the properties of the Avalon-ST interfaces.

Table 11–1. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Not supported.
Data Width	Data width = 8 bits; Bits per symbol = 8.
Channel	Not supported.
Error	Not used.
Packet	Not supported.

For more information about Avalon-ST interfaces, refer to the *Avalon Interface Specifications*.

Operation

The Avalon-ST SPI core waits for the `nSS` signal to be asserted low, signifying that the SPI master is initiating a transaction. The core then starts shifting in bits from the input signal `mosi`. The core packs the bits received on the SPI to bytes and checks for the following special characters:

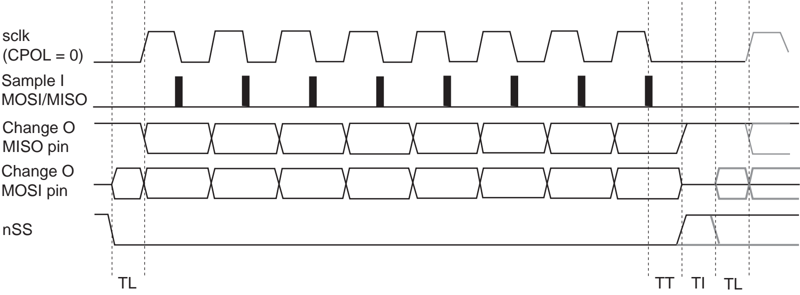
- `0x4a`—Idle character. The core drops the idle character.
- `0x4d`—Escape character. The core drops the escape character, and `XOR`s the following byte with `0x20`.

For each valid byte of data received, the core asserts the `valid` signal on its Avalon-ST source interface and presents the byte on the interface for a clock cycle.

At the same time, the core shifts data out from the Avalon-ST sink to the output signal `miso` beginning with from the most significant bit. If there is no data to shift out, the core shifts out idle characters (`0x4a`). If the data is a special character, the core inserts an escape character (`0x4d`) and XORs the data with `0x20`.

Figure 11–2 shows the SPI transfer protocol.

Figure 11–2. SPI Transfer Protocol



Notes to Figure 11–2:

(1) TL = The worst recovery time of `sclk` with respect with `nSS`.
(2) TT = The worst hold time for `MOSI` and `MISO` data.
(3) TI = The minimum width of a reset pulse required by Altera FPGA families.

Timing

The core requires a lead time (TL) between asserting the `nSS` signal and the SPI clock, and a lag time (TT) between the last edge of the SPI clock and deasserting the `nSS` signal. The `nSS` signal must be deasserted for a minimum idling time (TI) of one SPI clock between byte transfers. A TimeQuest SDC file (**.sdc**) is provided to remove false timing paths. The frequency of the SPI master's clock must be equal to or lower than the frequency of the core's clock.

Limitations

Daisy-chain configuration, where the output line `miso` of an instance of the core is connected to the input line `mosi` of another instance, is not supported.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the Avalon-ST SPI core in SOPC Builder to add the core to a system. The parameter **Number of synchronizer stages: Depth** allows you to specify the length of synchronization register chains. These register chains are used when a metastable event is likely to occur and the length specified determines the meantime before failure. The register chain length, however, affects the latency of the core.

For more information on metastability in Altera devices, refer to *AN 42: Metastability in Altera Devices*. For more information on metastability analysis and synchronization register chains, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Device Support

The Avalon-ST SPI core supports all Altera® device families.

Referenced Documents

This chapter references the following documents:

- *Avalon Interface Specifications*
- *SPI Slave/JTAG to Avalon Master Bridge Cores* chapter in volume 5 of the *Quartus II Handbook*
- *AN 42: Metastability in Altera Devices*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 11–2 shows the revision history for this chapter.

Table 11–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Added description of a new parameter, **Number of synchronizer stages: Depth**.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

 © March 2009 Altera Corporation

12. SPI Slave/JTAG to Avalon Master Bridge Cores

QII55011-9.0.0

Core Overview

The SPI Slave to Avalon® Master Bridge and the JTAG to Avalon Master Bridge cores provide a connection between host systems and SOPC Builder systems via the respective physical interfaces. Host systems can initiate Avalon Memory-Mapped (Avalon-MM) transactions by sending encoded streams of bytes via the cores' physical interfaces. The cores support reads and writes, but not burst transactions.

The SPI Slave to Avalon Master Bridge and the JTAG to Avalon Master Bridge are SOPC Builder-ready and integrates easily into any SOPC Builder-generated systems.

This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 12–3
- "Device Support" on page 12–3

Functional Description

Figure 12–1 shows a block diagram of the SPI Slave to Avalon Master Bridge core and its location in a typical system configuration.

Figure 12–1. SOPC Builder System with a SPI Slave to Avalon Master Bridge Core

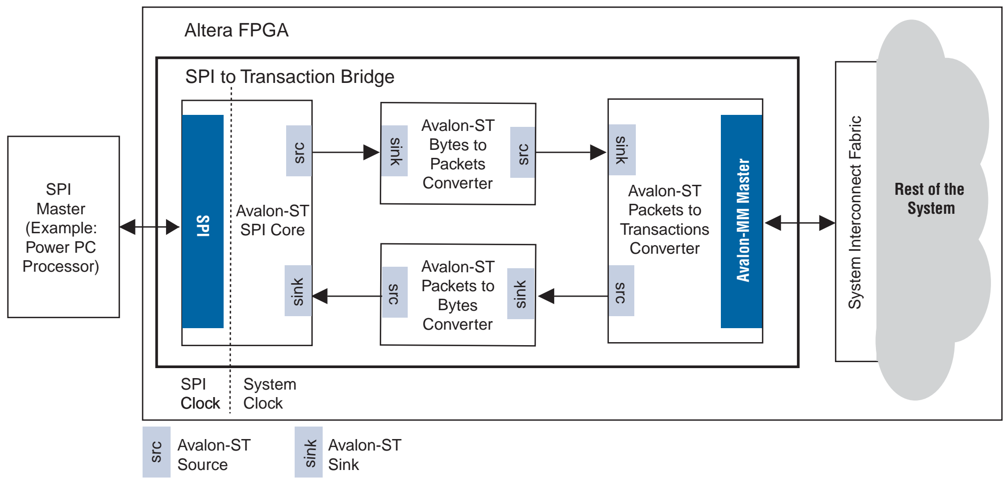
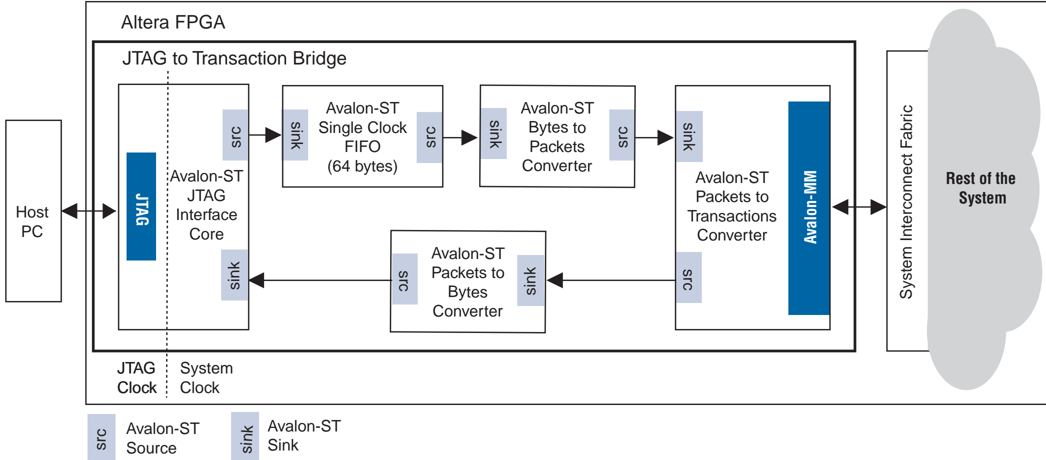


Figure 12–2 shows a block diagram of the JTAG to Avalon Master Bridge core and its location in a typical system configuration.

Figure 12–2. SOPC Builder System with a JTAG to Avalon Master Bridge Core



The SPI Slave to Avalon Master Bridge and the JTAG to Avalon Master Bridge cores accept encoded streams of bytes with transaction data on their respective physical interfaces and initiate Avalon-MM transactions on their Avalon-MM interfaces. Each bridge consists of the following cores, which are available as stand-alone components in SOPC Builder:

- **Avalon-ST Serial Peripheral Interface and Avalon-ST JTAG Interface**—Accepts incoming data in bits and packs them into bytes.
- **Avalon-ST Bytes to Packets Converter**—Transforms packets into encoded stream of bytes, and a likewise encoded stream of bytes into packets.
- **Avalon-ST Packets to Transactions Converter**—Transforms packets with data encoded according to a specific protocol into Avalon-MM transactions, and encodes the responses into packets using the same protocol.
- **Avalon-ST Single Clock FIFO**—Buffers data from the Avalon-ST JTAG Interface core. The FIFO is only used in the JTAG to Avalon Master Bridge.

For the bridges to successfully transform the incoming streams of bytes to Avalon-MM transactions, the streams of bytes must be constructed according to the protocols used by the cores.

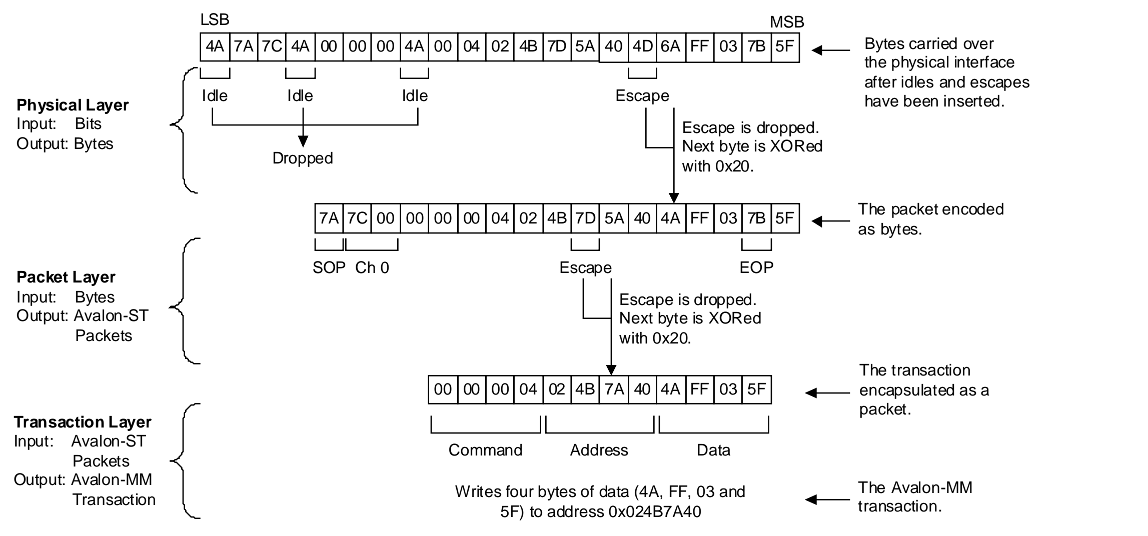
For more information about the protocol at each layer of the bridges and the single clock FIFO, refer to the following chapters:

- *Avalon-ST Serial Peripheral Interface Core* chapter in volume 5 of the *Quartus II Handbook*
- *Avalon-ST JTAG Interface Core* chapter in volume 5 of the *Quartus II Handbook*
- *Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores* chapter in volume 5 of the *Quartus II Handbook*

- *Avalon Packets to Transactions Converter Core* chapter in volume 5 of the *Quartus II Handbook*
- *Avalon-ST Single Clock and Dual Clock FIFO Cores* chapter in volume 5 of the *Quartus II Handbook*

The following example shows how a bytestream changes as it is transferred through the different layers in the bridges.

Figure 12–3. Bits to Avalon-MM Transaction



When the transaction is complete, the bridges send a response to the host system using the same protocol.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the SPI Slave to Avalon Master Bridge and the JTAG to Avalon Master Bridge in SOPC Builder to add the cores to a system. There are no user-configurable settings for the JTAG to Avalon Master Bridge core.

For the SPI Slave to Avalon Master Bridge core, the parameter **Number of synchronizer stages: Depth** allows you to specify the length of synchronization register chains. These register chains are used when a metastable event is likely to occur and the length specified determines the meantime before failure. The register chain length, however, affects the latency of the core.

For more information on metastability in Altera devices, refer to *AN 42: Metastability in Altera Devices*. For more information on metastability analysis and synchronization register chains, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Device Support

The SPI Slave to Avalon Master bridge supports all Altera® device families.

Referenced Documents

This chapter references the following documents:

- *Avalon-ST Serial Peripheral Interface Core* chapter in volume 5 of the *Quartus II Handbook*
- *Avalon-ST JTAG Interface Core* chapter in volume 5 of the *Quartus II Handbook*
- *Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores* chapter in volume 5 of the *Quartus II Handbook*
- *Avalon Packets to Transactions Converter Core* chapter in volume 5 of the *Quartus II Handbook*
- *Avalon-ST Single Clock and Dual Clock FIFO Cores* chapter in volume 5 of the *Quartus II Handbook*
- *AN 42: Metastability in Altera Devices*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 12–1 shows the revision history for this chapter.

Table 12–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Added description of a new parameter **Number of synchronizer stages: Depth**.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

13. PCI Lite Core

QII55010-9.0.0

Core Overview

The PCI Lite core is a protocol interface that translates PCI transactions to Avalon® Memory-Mapped (Avalon-MM) transactions with low latency and high throughput. The PCI Lite core uses the PCI-Avalon bridge to connect the PCI bus to the interconnect fabric, allowing you to easily create simple PCI systems that include one or more SOPC Builder components. This core has the following features:

- SOPC Builder ready
- PCI complexities, such as retry and disconnect are handled by the PCI/Avalon Bridge logic and transparent to the user
- Run-time configurable (dynamic) Avalon-to-PCI address translation
- Separate Avalon Memory-Mapped (Avalon-MM) slave ports for PCI bus access (PBA) and control register access (CRA)
- Support for Avalon-MM burst mode
- Common PCI and Avalon clock domains
- Option to increase PCI read performance by increasing the number of pending reads and maximum read burst size.

This chapter contains the following sections:

- "Performance and Resource Utilization"
- "Functional Description" on page 13–2
- "Instantiating the Core in SOPC Builder" on page 13–11
- "Device Support" on page 13–14
- "Simulation Considerations" on page 13–14

Performance and Resource Utilization

This section lists the resource utilization and performance data for supported devices when operating in the PCI Target-Only, and PCI Master/Target device modes for each of the application-specific performance settings.

The estimates are obtained by compiling the core using the Quartus® II software. Performance results vary depending on the parameters that you specify for the system module.

Table 13–1 shows the resource utilization and performance data for a Stratix® III device (EP3SE50F780C2). The performance of the MegaCore function in the Stratix IV family is similar to the Stratix III family.

Table 13–1. Memory Utilization and Performance Data for the Stratix III Family

PCI Device Mode	PCI Target	PCI Master	ALUTs *(2)*	Logic Register	M9K Memory Blocks	I/O Pins
Min *(1)*	Enabled	N/A	715	517	2	48
Max *(1)*	Enabled	Enabled	1,347	876	5	50

Notes to Table 13–1:

(1) **Min** = One BAR with minimum settings for each parameter.
Max = Three BARs with maximum settings for each parameter.

(2) The logic element (LE) count for the Stratix III family is based on the number of adaptive look-up tables (ALUTs) used for the design as reported by the Quartus II software.

Table 13–2 lists the resource utilization and performance data for a Cyclone III device (EP3C40F780C6).

Table 13–2. Memory Utilization and Performance Data for the Cyclone III Family

PCI Device Mode	PCI Target	PCI Master	Logic Elements	Logic Register	M4K Memory Blocks	I/O Pins
Min *(1)*	Enabled	N/A	1,057	511	2	48
Max *(1)*	Enabled	Enabled	2,027	878	5	50

Note to Table 13–2:

(1) **Min** = One BAR with minimum settings for each parameter.
Max = Three BARs with maximum settings for each parameter.

Functional Description

The following sections provide a functional description of the PCI Lite Core.

PCI-Avalon Bridge Blocks

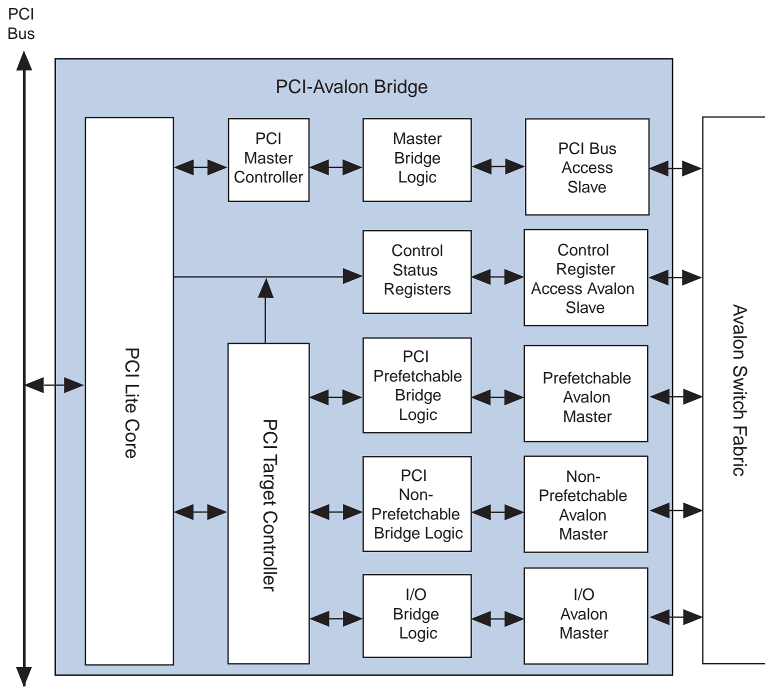
The PCI-Avalon bridge's blocks manage the connectivity for the following PCI operational modes:

- PCI Target-Only Peripheral
- PCI Master/Target Peripheral
- PCI Host-Bridge Device

Depending on the operational mode, the PCI-Avalon bridge uses some or all of the predefined Avalon-MM ports. Figure 13–1 shows a generic PCI-Avalon bridge block diagram, which includes the following blocks:

- Five predefined Avalon-MM ports
- Control registers
- PCI master controller (when applicable)
- PCI target controller

Figure 13–1. Generic PCI-Avalon Bridge Block Diagram



Avalon-MM Ports

The Avalon bridge comprises up to five predefined ports to communicate with the interconnect (depending on device operating mode).

This section discusses the five Avalon-MM ports:

- Prefetchable Avalon-MM master
- Non-Prefetchable Avalon-MM master
- I/O Avalon-MM master
- PCI bus access slave
- Control register access (CRA) Avalon-MM slave

Prefetchable Avalon-MM Master

The prefetchable Avalon-MM master port provides a high bandwidth PCI memory request access to Avalon-MM slave peripherals. This master port is capable of generating Avalon-MM burst transactions for PCI requests that hit a prefetchable base address register (BAR). You should only connect prefetchable Avalon-MM slaves to this port, typically RAM or ROM memory devices.

This port is optimized for high bandwidth transfers as a PCI target and it does not support single cycle transactions.

Non-Prefetchable Avalon-MM Master

The Non-Prefetchable Avalon-MM master port provides a low latency PCI memory request access to Avalon-MM slave peripherals. Burst operations are not supported on this master port. Only the exact amount of data needed to service the initial data phase is read from the interconnect. Therefore, the PCI byte enables (for the first data phase of the PCI read transaction) are passed directly to the interconnect.

This Avalon-MM master port is optimized for low latency access from PCI-to-Avalon-MM slaves. This is optimal for providing PCI target access to simple Avalon-MM peripherals.

I/O Avalon-MM Master

The I/O Avalon-MM master port provides a low latency PCI I/O request access to Avalon-MM slave peripherals. Burst operations are not supported on this master port. As only the exact amount of data needed to service the initial data phase is read from the interconnect, the PCI byte enables (for the first data phase of the PCI read transaction) are passed directly to the interconnect.

This Avalon-MM master port is also optimized for I/O access from PCI-to-Avalon-MM slaves for providing PCI target access to simple Avalon-MM peripherals.

PCI Bus Access Slave

This Avalon-MM slave port propagates the following transactions from the interconnect to the PCI bus:

- Single cycle memory read and write requests
- Burst memory read and write requests
- I/O read and write requests
- Configuration read and write requests

Burst requests from the interconnect are the only way to create burst transactions on the PCI bus.

This slave port is not implemented in the PCI Target-Only Peripheral mode.

Control Register Access (CRA) Avalon-MM Slave

This Avalon-MM slave port is used to access control registers in the PCI-Avalon bridge. To provide external PCI master access to these registers, one of the bridge's master ports must be connected to this port. There is no internal access inside the bridge from the PCI bus to these registers. You can only write to these registers from the interconnect. The `Control Register Access Avalon Slave` port is only enabled on Master/Target selection. The range of values supported by PCI CRA is 0x1000 to 0x1FFF. Depending on the system design, these values can be accessed by PCI processors, Avalon processors or both.

Table 13–3 on page 13–5 shows the instructions on how to use these values. The address translation table is writable via the `Control Register Access Avalon Slave` port. If the **Number of Address Pages** field is set to the maximum of **512**, `0x1FF8` contains `A2P_ADDR_MAP_LO511` and `0x1FFC` contains `A2P_ADDR_MAP_HI511`.

Each entry in the PCI address translation table is always 8 bytes wide. The lower order address bits that are treated as a pass through between Avalon-MM and PCI, and the number of pass-through bits, are defined by the size of page in the address translation table and are always forced to 0 in the hardware table. For example, if the page size is 4 KBytes, the number of pass-through bits is $\log_2$ (page size) = $\log_2$ (4 KBytes) = 12.

Refer to "Avalon-to-PCI Address Translation" on page 13–6 for more details.

Table 13–3. Avalon-to-PCI Address Translation Table – Address Range: 0x1000-0x1FFF

Address	Bit	Name	Access Mode	Description
0x1000	1:0	A2P_ADDR_SPACE0	W	Address space indication for entry 0. See Table 13–4 on page 13–7 for the definition of these bits.
	31:2	A2P_ADDR_MAP_LO0	W	Lower bits of Avalon-to-PCI address map entry 0. The pass through bits are not writable and are forced to 0.
0x1004	31:0	A2P_ADDR_MAP_HI0	W	Reserved.
0x1008	1:0	A2P_ADDR_SPACE1	W	Address Space indication for entry 1. See Table 13–4 on page 13–7 for the definition of these bits.
	31:2	A2P_ADDR_MAP_LO1	W	Lower bits of Avalon-to-PCI address map entry 1. Pass through bits are not writable and are forced to 0. This entry is only implemented if the number of pages in the address translation table is greater than 1.
0x100C	31:0	A2P_ADDR_MAP_HI1	W	Reserved.

Master and Target Performance

The performance of the PCI Lite core is designed to provide low-latency single-cycle and burst transactions.

Master Performance

The master provides high throughput for transactions initiated by Avalon-MM master devices to PCI target devices via the PCI bus master interface. Avalon-MM read transactions are implemented as latent read transfers. The PCI master device issues only one read transaction at a time.

The PCI bus access (PBA) handles the Avalon master transaction system interconnect hold state for 6 clock cycles. This is the maximum number of cycles supported by the PCI specification.

Target Performance

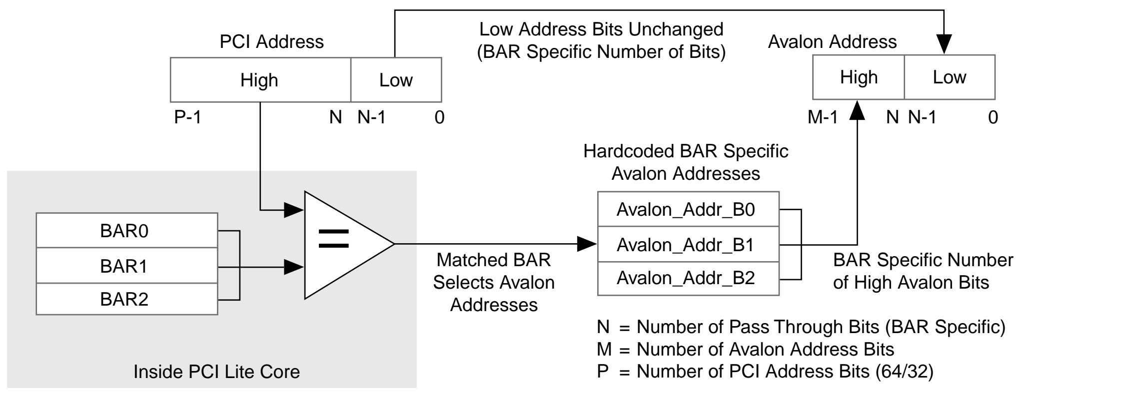
The target allows high throughput read/write operations to Avalon-MM slave peripherals. Read/write accesses to prefetchable base address registers (BARs) use dual-port buffers to enable burst transactions on both the PCI and Avalon-MM sides. This profile also allows access to the PCI BARs (Prefetchable, Non-Prefetchable, and I/O) to use their respective Avalon-MM master ports to initiate transfers to Avalon-MM slave peripherals. Prefetchable handles burst transaction and Non-Prefetchable and I/O handles only single-cycle transaction.

All PCI read transactions are completed as delayed reads. However, only one delayed read is accepted and processed at a time.

PCI-to-Avalon Address Translation

Figure 13–2 shows the PCI-to-Avalon address translation. The bits in the PCI address that are used in the BAR matching process are replaced by an Avalon-MM base address that is specific to that BAR.

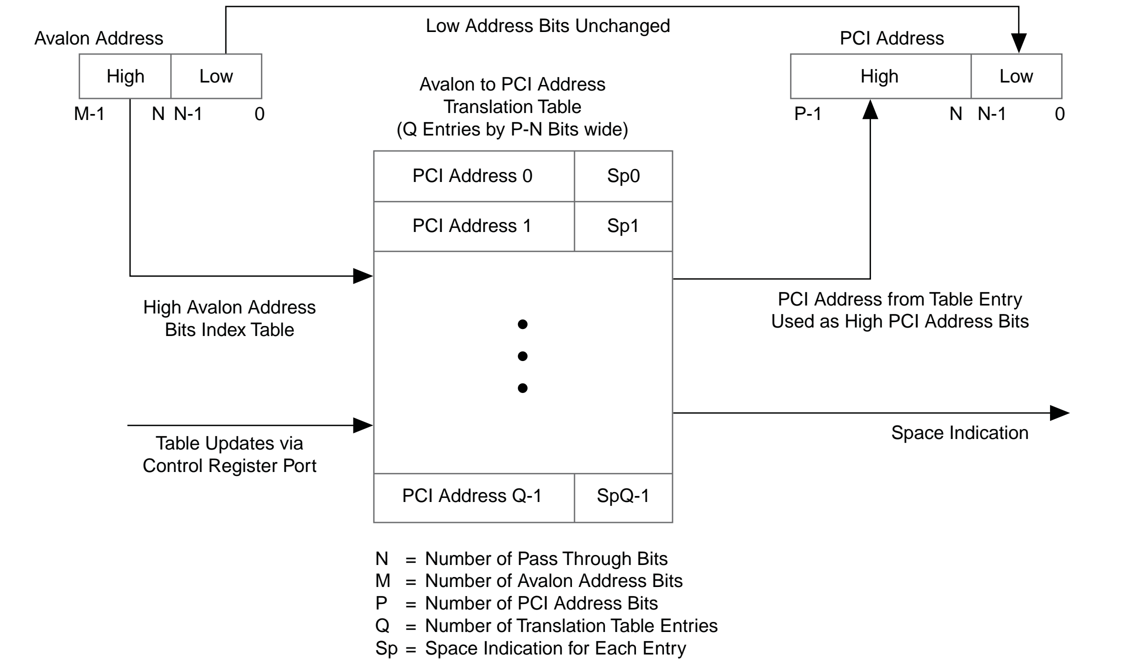
Figure 13–2. PCI-to-Avalon Address Translation



Avalon-to-PCI Address Translation

Avalon-to-PCI address translation is done through a translation table. Low order Avalon-MM address bits are passed to PCI unchanged; higher order Avalon-MM address bits are used to index into the address translation table. The value found in the table entry is used as the higher order PCI address bits. Figure 13–3 depicts this process.

Figure 13–3. Avalon-to-PCI Address Translation



The address size selections in the translation table determine both the number of entries in the Avalon-to-PCI address translation table, and the number of bits that are passed through the transaction table unchanged.

Each entry in the address translation table also has two address space indication bits, which specify the type of address space being mapped. If the type of address space being mapped is memory, the bits also indicate the resulting PCI address is a 32-bit address.

Table 13–4 shows the address space field's format of the address translation table entries.

Table 13–4. Address Space Bit Encodings

Address Space Indicator (Bits 1:0)	Description
`00`	Memory space, 32-bit PCI address. Address bits 63:32 of the translation table entries are ignored.
`01`	Reserved.
`10`	I/O space. The address from the translation table process is modified as described in Table 13–5.
`11`	Configuration space. The address from the translation table process is treated as a type 1 configuration address and is modified as described in Table 13–5.

If the space indication bits specify configuration or I/O space, subsequent modifications to the PCI address are performed. See Table 13–5.

Table 13–5. Configuration and I/O Space Address Modifications

Address Space	Modifications Performed
I/O	■ Address bits 2:0 are set to point to the first enabled byte according to the Avalon byte enables. (Bit 2 only needs to be modified when a 64-bit data path is in use.) ■ Address bits 31:3 are handled normally.
Configuration address bits 23:16 == 0 (bus number == 0)	■ Address bits 1:0 are set to `00` to indicate a type 0 configuration request. ■ Address bits 10:2 are passed through as normal. ■ Address bits 31:11 are set to be a one-hot encoding of the device number field (15:11) of the address from the translation table. For example, if the device number is `0x00`, address bit 11 is set to 1 and bits 31:12 are set to 0. If the device number is `0x01`, address bit 12 is set to 1 and bits 31:13, 11 are set to 0. ■ Address bits 31:24 of the original PCI address are ignored.
Configuration address bits 23:16 > 0 (bus number > 0)	■ Address bits 1:0 are set to `01` to indicate a type 1 configuration request. ■ Address bits 31:2 are passed through unchanged.

Avalon-To-PCI Read and Write Operation

The PCI Bus Access Slave port is a burst-capable slave that attempts to create PCI bursts that match the bursts requested from the interconnect.

The PCI-Avalon bridge is capable of handling bursts up to 512 bytes with a 32-bit PCI bus. In other words, the maximum supported Avalon-MM burst count is 128.

Bursts from Avalon-MM can be received on any boundary. However, when internal PCI-Avalon bridge bursts cross the Avalon-to-PCI address page boundary, they are broken into two pieces. Two bursts are used because the address translation can change at that boundary, requiring a different PCI address for the second portion of the burst with a burst count greater than 1.

Avalon-MM burst read requests are treated as if they are going to prefetchable PCI space. Therefore, if the PCI target space is non-prefetchable, you should not use read bursts.

Several factors control how Avalon-MM transactions (bursts or single cycle) are translated to PCI transactions. These cases are discussed in Table 13–6.

Table 13–6. Translation of Avalon Requests to PCI Requests

Data Path Width	Avalon Burst Count	Type of Operation	Avalon Byte Enables	Resulting PCI Operation and Byte Enables
32	1	Read or Write	Any value	Single data phase read or write, PCI byte enables identical to Avalon byte enables
32	>1	Read	Any value	Attempt to burst on PCI. All data phases have all PCI bytes enabled.
32	>1	Write	Any value	Attempt to burst on PCI. All data phases have PCI byte enables identical to the Avalon byte enables.

© March 2009 Altera Corporation

Avalon-to-PCI Write Requests

For write requests from the interconnect, the write request is pushed onto the PCI bus as a configuration write, I/O write, or memory write. When the Avalon-to-PCI command/write data buffer either has enough data to complete the full burst or 8 data phases (32 bytes on a 32-bit PCI bus) are exceeded, the PCI master controller issues the PCI write transaction.

The PCI write is issued to configuration, I/O, or memory space based on the address translation table. See "Avalon-to-PCI Address Translation" on page 13–6.

A PCI write burst can be terminated for various reasons. Table 13–7 describes the resulting action for the PCI master write request termination condition.

Table 13–7. PCI Master Write Request Termination Conditions

Termination condition	Resulting Action
Burst count satisfied	Normal master-initiated termination on PCI bus, command completes, and the master controller proceeds to the next command.
Latency timer expiring during configuration, I/O, or memory write command	Normal master-initiated termination on PCI bus, the continuation of the PCI write is requested from the master controller arbiter.
Avalon-to-PCI command/write data buffer running out of data	Normal master-initiated termination on the PCI bus. Master controller waits for the buffer to reach 8 DWORDs on a 32-bit PCI or 16 DWORDs on a 64-bit PCI, or there is enough data to complete the remaining burst count. Once enough data is available, the master controller arbiter continues wth the PCI write.
PCI target disconnect	The master controller arbiter attempts to initiate the PCI write until the transaction is successful.
PCI target retry	
PCI target-abort	The rest of the write data is read from the buffer and discarded.
PCI master-abort	

Avalon-to-PCI Read Requests

For read requests from the interconnect, the request is pushed on the PCI bus by a configuration read, I/O read, memory read, memory read line, or memory read multiple command. The PCI read is issued to configuration, I/O, or memory space based on the address translation table entry. See "Avalon-to-PCI Address Translation" on page 13–6.

If a memory space read request can be completed in a single data phase, it is issued as a memory read command. If the memory space read request spans more than one data phase but does not cross a cacheline boundary (as defined by the cacheline size register), it is issued as a memory read line command. If the memory space read request crosses a cache line boundary, it is issued as multiple memory read commands.

Read requests on PCI may initially be retried. Retries depend on the response time from the target. The master continues to retry until it gets the required data.

Table 13–8 shows PCI master read request termination conditions.

Table 13–8. PCI Master Read Request Termination Conditions

Termination Condition	Resulting Action
Burst count satisfied	Normal master initiated termination on the PCI bus. Master controller proceeds to the next command.
Latency timer expired	Normal master initiated termination on PCI bus. The continuation of the PCI read is made pending as a request from the master controller arbiter.
PCI target disconnect	The continuation of the PCI read is requested from the master controller arbiter.
PCI target retry	
PCI target-abort	Dummy data is returned to complete the Avalon-MM read request. The next operation is then attempted in a normal fashion.
PCI master-abort	

Ordering of Requests

The PCI-Avalon bridge handles the following types of requests:

- PMW—Posted memory write.
- DRR—Delayed read request.
- DWR—Delayed write request. DWRs are I/O or configuration write operation requests. The PCI-Avalon bridge does not handle DWRs as delayed writes.
  - As a PCI master, I/O or configuration writes are generated from posted Avalon-MM writes. If required to verify completion, you must issue a subsequent read request to the same target.
  - As a PCI target, configuration writes are the only requests accepted, which are never delayed. These requests are handled directly by the PCI core.
- DRC—Delayed read completion.
- DWC—Delayed write completion. These are never passed through to the core in either direction. Incoming configuration writes are never delayed. Delayed write completion status is not passed back at all.

Every single transaction that is initiated, locks the core until it is completed. Only then can a new transaction be accepted.

PCI Interrupt

When Avalon-MM asserts the `IRQ` signal, an interrupt on the PCI bus occurs. The Avalon-MM `IRQ` input causes a bit to be set in the PCI interrupt status register.

Instantiating the Core in SOPC Builder

Table 13–9 describes the parameters that can be configured in SOPC Builder for the PCI Lite core.

Table 13–9. Parameters for PCI Lite Core (Part 1 of 2)

Parameters	Legal Values	Description
Enable Master/Target Mode	On or Off	Turning this option **On** enables Master/Target mode. This option enables allows Avalon-MM master devices to access PCI target devices via the PCI bus master interface, and PCI bus master devices to access Avalon-MM slave devices via the PCI bus target interface. Turning this option **Off** means you have selected Target Only mode, which allows PCI bus mastering devices to access Avalon-MM slave devices via the PCI bus target interface.
Enable Host Bridge Mode	On or Off	Turning this option **On** enables this mode. In addition to the same features provided by the PCI Master/Target mode, Host Bridge Mode provides host bridge functionality including hardwiring the master enable bit to 1 in the PCI command register and allowing self-configuration. This value can only be set if the **Enable Master/Target Mode** option is turned **On**.
Number of Address Pages	2, 4, 8, or 16	The number of translation/pages supported by the device for Avalon to PCI address translation.
Size of Address Pages	12–27	The supported address size (in bits) that can be assigned to each map number entries.
Prefetchable BAR	On or Off	Turning this option **On** invokes a Prefetchable Master (PM) Bar in the PCI system. This option allows PCI-Avalon Bridge Lite to accept and process PM transactions.
Prefetchable BAR Size	10–31	The allowed reserved address range supported by the PM BAR. The reserved memory space is 1 KByte (10 bits) to 4 GBytes (31 bits).
Prefetchable BAR Avalon Address Translation Offset	*<BAR translation value>*	The direct translation of the value that hits the BAR and modified to a fixed address in the Avalon space. Refer to "PCI-to-Avalon Address Translation" on page 13–6.
Non-Prefetchable BAR	On or Off	Turning this option **On** invokes a Non-Prefetchable Master (NPM) Bar in the PCI system. This option allows the PCI-Avalon Bridge Lite to accept and process NPM transactions.
Non-Prefetchable BAR Size	10–31	Specifies the allowed reserved address range supported by the NPM BAR. The reserved memory space is 1 KByte (10 bits) to 4 GBytes (31 bits).
Non-Prefetchable BAR Avalon Address Translation Offset	*<BAR translation value>*	The direct translation of the value that hits the BAR and modified to a fixed address in the Avalon space. Refer to "PCI-to-Avalon Address Translation" on page 13–6.
I/O BAR	On or Off	Turning this option **On** enables an I/O BAR in the system. This option allows PCI-Avalon Bridge Lite to accept and process I/O type transactions.
I/O BAR Size	2–8	The allowed reserved address range supported by the I/O BAR. The reserved memory space is 4 bytes (2 bits) to 256 bytes (8 bits).
I/O BAR Avalon Address Translation Offset	*<BAR translation value>*	The direct translation of the value that hits the BAR and modified to a fixed address in the Avalon address space. Refer to "PCI-to-Avalon Address Translation" on page 13–6.

Table 13–9. Parameters for PCI Lite Core (Part 2 of 2)

Parameters	Legal Values	Description
Maximum Target Read Burst Size	1, 2, 4, 8, 16, 32, 64, or 128	Specifies the maximum FIFO depth that is used for reading. Larger values allow more reads to be read in a single transaction but also require more time to clear the FIFO content.
Device ID	<*register value*>	Device ID register. This parameter is a 16-bit hexadecimal value that sets the device ID register in the configuration space.
Vendor ID	<*register value*>	Vendor ID register.This parameter is a 16-bit read-only register that identifies the manufacturer of the device. The value of this register is assigned by the PCI Special Interest Group (SIG).
Class Code	<*register value*>	Class code register. This parameter is a 24-bit hexadecimal value that sets the class code register in the configuration space. The value entered for this parameter must be valid PCI SIG-assigned class code register value.
Revision ID	<*register value*>	Revision ID register. This parameter is an 8-bit read-only register that identifies the revision number of the device. The value of this register is assigned by the manufacturer.
Subsystem ID	<*register value*>	Subsystem ID register. This parameter is a 16-bit hexadecimal value that sets the subsystem ID register in the PCI configuration space. Any value can be entered for this parameter.
Subsystem Vendor ID	<*register value*>	Subsystem vendor ID register. This parameter is a 16-bit hexadecimal value that sets the subsystem vendor ID register in the PCI configuration space. The value for this parameter must be a valid PCI SIG-assigned vendor ID number.
Maximum Latency	<*register value*>	Maximum latency register. This parameter is an 8-bit hexadecimal value that sets the maximum latency register in the configuration space. This parameter must be set according to the guidelines in the PCI specifications. Only meaningful when the **Enable Master/Target Mode** option is turned **On**.
Minimum Grant	<*register value*>	Minimum grant register. This parameter is an 8-bit hexadecimal value that sets the minimum grant register in the PCI configuration space. This parameter must be set according to the guidelines in the PCI specifications. Only meaningful when the **Enable Master/Target Mode** option is turned **On**.

PCI Timing Constraint Files

The PCI Lite core supplies a Tcl timing constraint file for your target device family.

When run, the constraint file automatically sets the PCI Lite core assignments for your design such as PCI Lite core hierarchy, device family, density and package type used in your Quartus II project.

To run a PCI constraint file, perform the following steps:

1. Copy **pci_constraints.tcl** from **<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite**.

2. Update the pin list in the Tcl constraint file. Edit the `get_user_pin_name` procedure in the Tcl constraint file to match the default pin names. To edit the PCI constraint file, follow these steps:

   a. Locate the `get_user_pin_name` procedure. This procedure maps the default PCI pin names to user PCI pin names. The following lines are the first few lines of the procedure:

```
proc get_user_pin_name { internal_pin_name } {

    #---------------- Do NOT change ------------------------------- ---- Change -----
    array set map_user_pin_name_to_internal_pin_name {ad                          ad       }
```

   b. Edit the pin names under the `Change` header in the file to match the PCI pin names used in your Quartus II project. In the following example, the name `ad` is changed to `pci_ad`:

```
    #---------------- Do NOT change -------------------------------     ---- Change -----
    array set map_user_pin_name_to_internal_pin_name { ad                pci_ad         }
```

The Tcl constraint file uses the default PCI pin names to make assignments. When overwriting existing assignments, the Tcl constraint file checks the new assignment pin names against the default PCI pin names. You must update the assignment pin names if there is a mismatch between the assignment pin names and the default PCI pin names.

3. Source the constraint file by typing the following in the Quartus II Tcl Console window:

   `source pci_constraints.tcl ↵`

4. Add the PCI constraints to your project by typing the following command in the Quartus II Tcl Console window:

   `add_pci_constraints ↵`

   See "Additional Tcl Option" on page 13–13 for the option supported by the **add_pci_constraints** command.

   When you add the timing constraints file as described in Step 4 above, the Quartus II software generates a Synopsys Design Constraints (**.sdc**) file with the file name format, <*variation name*>**.sdc**. The Quartus II TimeQuest timing analyzer uses the constraints specified in this file.

For more information about **.sdc** files or TimeQuest timing analyzer, refer to Quartus II Help.

Additional Tcl Option

If you do not want to compile your project and prefer to skip analysis and synthesis, you can use the `-no_compile` option:

```
add_pci_contraints [-no_compile]
```

By default, the `add_pci_constraints` command performs analysis and synthesis in the Quartus II software to determine the hierarchy of your PCI Lite core design. You should only use this option if you have already performed analysis and synthesis or fully compiled your project prior to using this script.

Device Support

The PCI Lite core supports the Arria® GX, Arria II, Cyclone® III, Hardcopy® II, Stratix® III, and Stratix IV device families.

Simulation Considerations

The PCI Lite core includes a testbench that facilitates the design and verification of systems that implement the Altera PCI-Avalon bridge. The testbench only works for master systems and is provided in Verilog HDL only.

To use the PCI testbench, you must have a basic understanding of PCI bus architecture and operations.This section describes the features and applications of the PCI testbench to help you successfully design and verify your design.

Features

The PCI testbench includes the following features:

- Easy to use simulation environment for any standard Verilog HDL simulator
- Open source Verilog HDL files
- Flexible PCI bus functional model to verify your application that uses any PCI Lite core
- Simulates all basic PCI transactions including memory read/write operations, I/O read/write transactions, and configuration read/write transactions
- Simulates all abnormal PCI transaction terminations including target retry, target disconnect, target abort, and master abort
- Simulates PCI bus parking

Master Transactor (mstr_tranx)

The master transactor simulates the master behavior on the PCI bus. It serves as an initiator of PCI transactions for PCI testbench. The master transactor has three main sections:

- TASKS (Verilog HDL)
- INITIALIZATION
- USER COMMANDS

TASKS Sections

The TASKS (Verilog HDL) sections define the events that are executed for the user commands supported by the master transactor. The events written in the TASKS sections follow the phases of a standard PCI transaction as defined by the *PCI Local Bus Specification, Revision 3.0*, including:

- Address phase
- Turn-around phase (read transactions)
- Data phases
- Turn-around phase

The master transactor terminates the PCI transactions in the following cases:

- The PCI transaction has successfully transferred all the intended data.
- The PCI target terminates the transaction prematurely with a target retry, disconnect, or abort as defined in the *PCI Local Bus Specification, Revision 3.0*.
- A target does not claim the transaction resulting in a master abort.

The bus monitor informs the master transactor of a successful data transaction or a target termination. Refer to the source code, which shows you how the master transactor uses these termination signals from the bus monitor.

The PCI testbench master transactor TASKS sections implement basic PCI transaction functionality. If your application requires different functionality, modify the events to change the behavior of the master transactor. Additionally, you can create new procedures or tasks in the master transactor by using the existing events as an example.

INITIALIZATION Section

This user-defined section defines the parameters and reset length of your PCI bus on power-up. Specifically, the system should reset the bus and write the configuration space of the PCI agents. You can modify the master transactor INITIALIZATION section to match your system requirements by changing the time that the system reset is asserted and by modifying the data written in the configuration space of the PCI agents.

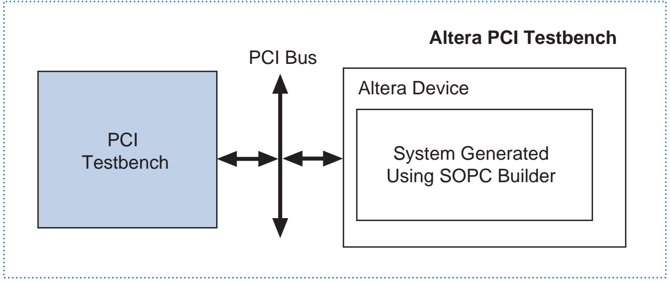
USER COMMANDS Section

The master transactor USER COMMANDS section contains the commands that initiate the PCI transactions you want to run for your tests. The list of events that are executed by these commands is defined in the TASKS sections. Customize the USER COMMANDS section to execute the sequence of commands needed to test your design.

Simulation Flow

This section describes the simulation flow using Altera PCI testbench. Figure 13–4 shows the block diagram of a typical verification environment using the PCI testbench.

Figure 13–4. Typical Verification Environment Using the PCI Testbench



The simulation flow using Altera PCI testbench comprises the following steps.

1. Use SOPC Builder to create your system. SOPC creates the <*variation name*_**system>_sim** folder in your project directory.
2. Source **pci_constraints.tcl**.
3. Copy **<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/verilog/pci_lite/trgt_tranx_mem_init.dat** to **<project_directory>/<variation name_system>_sim** folder.
4. Edit the top level HDL verilog files in the testbench. Insert the following lines just before `module test_bench`.

```
'include
"<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/
verilog/pci_lite/pci_tb.v"
'include
"<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/
verilog/pci_lite/clk_gen.v"
'include
"<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/
verilog/pci_lite/arbiter.v"
'include
"<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/
verilog/pci_lite/pull_up.v"
'include
"<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/
verilog/pci_lite/monitor.v"
'include
"<quartus_root>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/
verilog/pci_lite/trgt_tranx.v"
'include "mstr_tranx.v"
```

Modify **mstr_tranx.v** in your project directory to add the PCI transactions to your system. If you regenerate your system, SOPC Builder overwrites the testbench files in the **<sopc_system>_sim** directory. If you want the default testbench files, regenerate the system. Then resource **pci_constraints.tcl** or simply copy the **mstr_tranx.v** from **<quartus_ip>/ip/sopc_builder_ip/altera_avalon_pci_lite/pci_sim/verilog/pci_lite** into your project folder and repeat steps 3 and 4.

5. Set the initialization parameters, which are defined in the master transactor model source code. These parameters control the address space reserved by the target transactor model and other PCI agents on the PCI bus.
6. The master transactor defines the tasks (Verilog HDL) needed to initiate PCI transactions in your testbench. Add the commands that correspond to the transactions you want to implement in your tests to the master transactor model source code. At a minimum, you must add configuration commands to set the BAR for the target transactor model and write the configuration space of the PCI Lite core. Additionally, you can add commands to initiate memory or I/O transactions to the PCI Lite core.

7. Compile the files in your simulator, including the testbench modules and the files created by SOPC Builder.
8. Simulate the testbench for the desired time period.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 13–10 shows the revision history for this chapter.

Table 13–10. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. Edited the command errors in the **Simulation Flow** section.	—
May 2008 v8.0.0	Initial release.	—

Section II. On-Chip Storage Peripherals

This section describes on-chip storage peripherals provided for SOPC Builder systems.

This section includes the following chapters:

- Chapter 14, Avalon-ST Single Clock and Dual Clock FIFO Cores
- Chapter 15, On-Chip FIFO Memory Core
- Chapter 16, Avalon-ST Multi-Channel Shared Memory FIFO Core

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

14. Avalon-ST Single Clock and Dual Clock FIFO Cores

QII55014-9.0.0

Core Overview

The Avalon® Streaming (Avalon-ST) Single Clock and Avalon-ST Dual Clock FIFO cores are FIFO buffers which operate with a single clock and separate clocks for input and output ports, respectively. You can configure the cores to include Avalon Memory-Mapped (Avalon-MM) status interfaces to report the FIFO fill level.

The Avalon-ST Single Clock and Avalon-ST Dual Clock FIFO cores are SOPC Builder-ready and integrates easily into any SOPC Builder-generated systems.

This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 14–3
- "Device Support" on page 14–3
- "Software Programming Model" on page 14–4

Functional Description

Figure 14–1 and Figure 14–2 show block diagrams of the Avalon-ST Single Clock and Avalon-ST Dual Clock FIFO cores.

Figure 14–1. Avalon-ST Single Clock FIFO Core

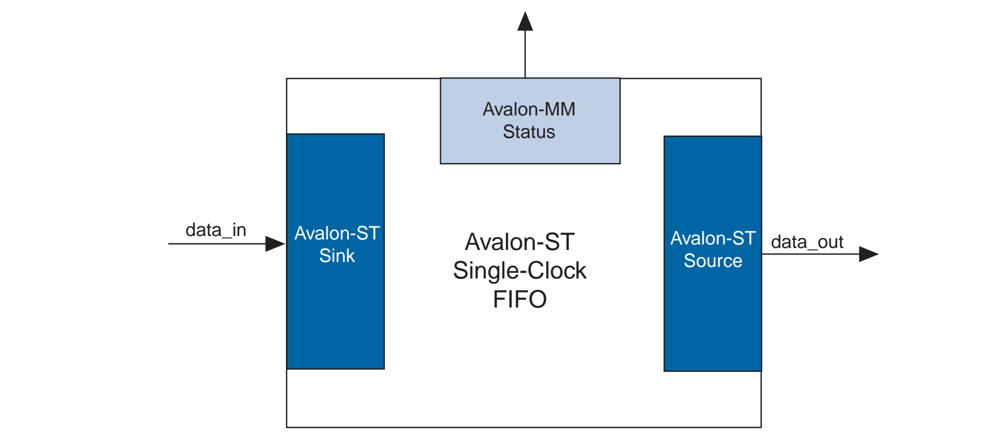


Figure 14–2. Avalon-ST Dual Clock FIFO Core

Interfaces

Table 14–1 shows the properties of the Avalon-ST interfaces.

Table 14–1. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Configurable.
Channel	Supported, up to 255 channels.
Error	Configurable.
Packet	Configurable.

For more information about Avalon-ST interfaces, refer to the *Avalon Interface Specifications*.

Operations

The Avalon-ST Single Clock and Avalon-ST Dual Clock FIFO cores are simple FIFO buffers with Avalon-ST input and output interfaces.

You can include an optional Avalon-MM status interface by setting the `Use_Fill_Level` parameter to 1. This interface reports the FIFO fill level. In the Dual Clock FIFO, you can implement separate status interfaces for the input and output clock domains.

Due to the latency of the clock crossing logic, the fill levels reported in the input and output clock domains may be different at any given instance. In both cases, the fill level is pessimistic for the clock domain; the fill level is reported high in the input clock domain and low in the output clock domain.

In the Avalon-ST Dual Clock FIFO, the FIFO has an output pipeline stage to improve f_{MAX}. This output stage is accounted for when calculating the output fill level, but not when calculating the input fill level. Hence, the best measure of the amount of data in the FIFO is given by the fill level in the output clock domain, while the fill level in the input clock domain represents the amount of space available in the FIFO (Available space = **FIFO depth** – input fill level).

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the Avalon-ST Single Clock and Avalon-ST Dual Clock FIFO cores in SOPC Builder to add the cores to a system.

Table 14–2 lists and describes the parameters you can configure.

Table 14–2. Configurable Parameters

Parameter	Legal Values	Description
Bits per symbol	1–32	The symbol width in bits.
Symbols per beat	1–32	The number of symbols transferred in a beat.
Error width	0–32	The width of the `error` signal.
FIFO depth	1–32	The FIFO depth. An output pipeline stage is added to the FIFO to increase performance, which increases the FIFO depth by one.
Use packets	0 or 1	Setting this parameter to 1 enables packet support on the Avalon-ST data interfaces.
Avalon-ST Single Clock FIFO Only		
Use fill level	0 or 1	Setting this parameter to 1 enables the Avalon-MM status interface.
Avalon-ST Dual Clock FIFO Only		
Use sink fill level	0 or 1	Setting this parameter to 1 enables the input clock domain Avalon-MM status interface.
Use source fill level	0 or 1	Setting this parameter to 1 enables the output clock domain Avalon-MM status interface.
Write pointer synchronizer length	2–8	The length of the write pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability while increasing the latency of the core.
Read pointer synchronizer length	2–8	The length of the read pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability.

For more information on metastability in Altera devices, refer to *AN 42: Metastability in Altera Devices*. For more information on metastability analysis and synchronization register chains, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Device Support

The Avalon-ST Single Clock and Avalon-ST Dual Clock FIFO cores support all Altera device families.

Software Programming Model

The following sections describe the software programming model for the Avalon-ST Single Clock and Avalon-ST Dual Clock FIFO cores.

HAL System Library Support

The Altera-provided driver implements a HAL device driver that integrates into the HAL system library for Nios II systems. HAL users should access the Avalon-ST Single Clock and Avalon-ST Dual Clock FIFO cores via the familiar HAL API and the ANSI C standard library.

Register Map

The Avalon-MM status interface reports the FIFO fill level. Table 14–3 shows the register map for the status interface of the cores.

Table 14–3. Register Map—Status Interface

Offset	Name	Access	Description
Base + 0	`Fill Level`	R	24-bit FIFO fill level. Bits 24 to 31 are unused.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 14–4 shows the revision history for this chapter.

Table 14–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Added description of new parameters, **Write pointer synchronizer length** and **Read pointer synchronizer length**.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

15. On-Chip FIFO Memory Core

QII55002-9.0.0

Core Overview

The on-chip FIFO memory core is a configurable component used to buffer data and provide flow control in an SOPC Builder system. The FIFO can operate with a single clock or with separate clocks for the input and output ports.

The input interface to the FIFO may be an Avalon® Memory Mapped (Avalon-MM) write slave or an Avalon Streaming (Avalon-ST) sink. The output interface can be an Avalon-ST source or an Avalon-MM read slave. The data is delivered to the output interface in the same order that it was received at the input interface, regardless of the value of channel, packet, frame, or any other signals.

In single clock mode, the on-chip FIFO memory includes an optional status interface that provides information about the fill-level of the FIFO. In dual clock mode, separate, optional status interfaces can be included for the input and output interfaces. The status interface also includes registers to set and control interrupts.

The on-chip FIFO memory core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. Device drivers are provided in the HAL system library allowing software to access the core using ANSI C.

This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 15–6
- "Instantiating the Core in SOPC Builder" on page 15–6
- "Software Programming Model" on page 15–8
- "Programming with the On-Chip FIFO Memory" on page 15–8
- "On-Chip FIFO Memory API" on page 15–13

Functional Description

The on-chip FIFO memory has four configurations:

- Avalon-MM write slave to Avalon-MM read slave
- Avalon-ST sink to Avalon-ST source
- Avalon-MM write slave to Avalon-ST source
- Avalon-ST sink to Avalon-MM read slave

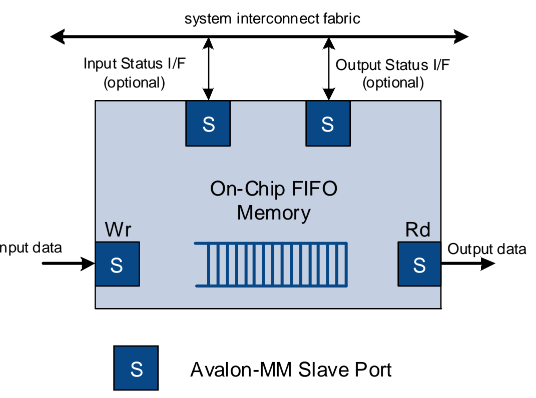
In all configurations, the input and output interfaces can use the optional backpressure signals to prevent underflow and overflow conditions. For the Avalon-MM interface, backpressure is implemented using the `waitrequest` signal. For Avalon-ST interfaces, backpressure is implemented using the `ready` and `valid` signals. For the on-chip FIFO memory, the delay between the sink asserts `ready` and the source drives valid data is one cycle.

Avalon-MM Write Slave to Avalon-MM Read Slave

In this mode, the FIFO's input is a zero-address-width Avalon-MM write slave. An Avalon-MM write master pushes data into the FIFO by writing to the input interface, and a read master (possibly the same master) pops data by reading from its output interface. The FIFO's input and output data must be the same width.

If **Allow backpressure** is turned on, the `waitrequest` signal is asserted whenever the `data_in` master tries to write to a full FIFO. `waitrequest` is only deasserted when there is enough space in the FIFO for a new transaction to complete. `waitrequest` is asserted for read operations when there is no data to be read from the FIFO, and is deasserted when the FIFO has data.

Figure 15–1. FIFO with Avalon-MM Input and Output Interfaces



Avalon-ST Sink to Avalon-ST Source

This FIFO has streaming input and output interfaces as illustrated in Figure 15–2. You can parameterize most aspects of the Avalon-ST interfaces including the **bits per symbol**, **symbols per beat**, and the width of `error` and `channel` signals. The input and output interfaces must be the same width. If **Allow backpressure** is on in the SOPC Builder MegaWizard, both interfaces use the `ready` and `valid` signals to indicate when space is available in the FIFO and when valid data is available.

For more information about the Avalon-ST interface protocol, refer to the *Avalon Interface Specifications*.

Figure 15–2. FIFO with Avalon-ST Input and Output Interfaces

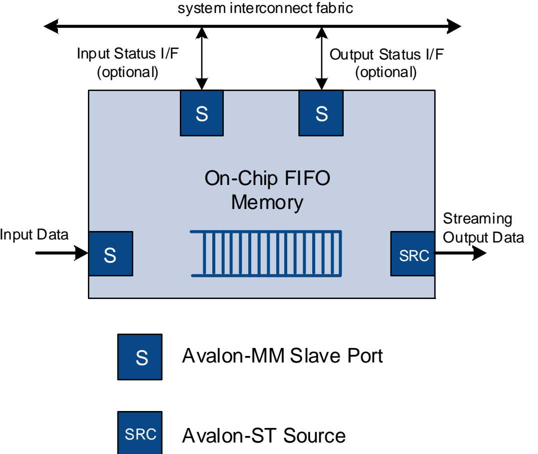
System Interconnect Fabric
Input Status I/F (optional)
Output Status I/F (optional)
S
S
On-Chip FIFO Memory
SNK
SRC
Streaming Output Data
SNK Avalon-ST Sink
SRC Avalon-ST Source
S Avalon-MM Slave Port

Avalon-MM Write Slave to Avalon-ST Source

In this mode, the FIFO's input is an Avalon-MM write slave with a width of 32 bits as shown in Figure 15–3. The Avalon-ST output (source) data width must also be 32 bits. You can configure output interface parameters, including: **bits per symbol**, **symbols per beat**, and the width of the `channel` and `error` signals. The FIFO performs the endian conversion to conform to the output interface protocol.

The signals that comprise this interface are mapped into bits in the Avalon's address space. If **Allow backpressure** is on, the input interface asserts `waitrequest` to indicate that the FIFO does not have enough space for the transaction to complete.

Figure 15–3. FIFO with Avalon-MM Input Interface and Avalon-ST Output Interface



The example memory map in Table 15–1 illustrates the layout of memory for a FIFO with a 32-bit Avalon-MM input interface and an Avalon-ST output interface. The output interface has 8-bit symbols, a 5-bit channel signal, and a 3-bit error signal, with packet support.

Table 15–1. Avalon-MM to Avalon-ST Memory Map

Offset	31 24	23 19	18 16	15 13	12 8	7 4	3 2	1	0
base + 0	Symbol 3	Symbol 2		Symbol 1		Symbol 0			
base + 1	reserved	reserved	`error`	reserved	`channel`	reserved	empty	`EOP`	`SOP`

If **Enable packet data** is off, the Avalon-MM write master writes all data at address offset 0 repeatedly to push data into the FIFO.

If **Enable packet data** is on, the Avalon-MM write master starts by writing the `SOP`, `error` (optional), `channel` (optional), `EOP`, and empty packet status information at address offset 1. Writing to address offset 1 does not push data into the FIFO. The Avalon-MM master then writes packet data to the FIFO repeatedly at address offset 0, pushing 8-bit symbols into the FIFO. Whenever a valid write occurs at address offset 0, the data and its respective packet information is pushed into the FIFO. Subsequent data is written at address offset 0 without the need to clear the `SOP`. Rewriting to address offset 1 is not required each time if the subsequent data to be pushed into the FIFO is not the end-of-packet data, as long as `error` and `channel` do not change.

At the end of each packet, the Avalon-MM master writes to the address at offset 1 to set the `EOP` bit to 1, before writing the last symbol of the packet at offset 0. The write master uses the empty field to indicate the number of unused symbols at the end of the transfer. If the last packet data is not aligned with the **symbols per beat**, the empty field indicates the number of empty symbols in the last packet data. For example, if the Avalon-ST interface has **symbols per beat** of 4, and the last packet only has 3 symbols, the empty field will be 1, indicating that one symbol (the least significant symbol in the memory map) is empty.

Avalon-ST Sink to Avalon-MM Read Slave

In this mode, the FIFO's input is an Avalon-ST sink and the output is an Avalon-MM read slave with a width of 32 bits (Figure 15–4). The Avalon-ST input (sink) data width must also be 32 bits. You can configure input interface parameters, including: **bits per symbol**, **symbols per beat**, and the width of the `channel` and `error` signals. The FIFO performs the endian conversion to conform to the output interface protocol.

An Avalon-MM master reads the data from the FIFO. The signals are mapped into bits in the Avalon's address space. If **Allow backpressure** is on in the SOPC Builder MegaWizard, the input (sink) interface uses the `ready` and `valid` signals to indicate when space is available in the FIFO and when valid data is available. For the output interface, `waitrequest` is asserted for read operations when there is no data to be read from the FIFO. It is deasserted when the FIFO has data to send.

Figure 15–4. FIFO with Avalon-ST Input and Avalon-MM Output

As shown in Table 15–2, the memory map for the Avalon-ST to Avalon-MM slave FIFO is exactly the same as for Avalon-MM to Avalon-ST FIFO.

Table 15–2. Avalon-ST to Avalon-MM Memory Map

Offset	31 24	23 19	18 16	15 13	12 8	7 4	3 2	1	0
base + 0	Symbol 3	Symbol 2		Symbol 1		Symbol 0			
base + 1	reserved	reserved	`error`	reserved	`channel`	reserved	empty	`EOP`	`SOP`

If **Enable packet data** is off, read data repeatedly at address offset 0 to pop the data from the FIFO.

If **Enable packet data** is on, the Avalon-MM read master starts reading from address offset 0. If the read is valid, that is, the FIFO is not empty, both data and packet status information are popped from the FIFO. The packet status information is obtained by reading at address offset 1. Reading from address offset 1 does not pop data from the FIFO. The `error`, `channel`, `SOP`, `EOP` and empty fields are available at address offset 1 to determine the status of the packet data read from address offset 0.

The empty field indicates the number of empty symbols in the data field. For example, if the Avalon-ST interface has symbols-per-beat of 4, and the last packet data only has 1 symbol, then the empty field will be 3 to indicate that 3 symbols (the 3 least significant symbols in the memory map) are empty.

Status Interface

The FIFO provides two optional status interfaces, one for the master writing to the input interface and a second for the read master reading from the output interface. For FIFOs that operate in a single domain, a single status interface is sufficient to monitor the status of the FIFO. For FIFOs using a dual clocking scheme, a second status interface using the output clock is necessary to accurately monitor the status of the FIFO in both clock domains.

Clocking Modes

When single clock mode is used, the FIFO being used is SCFIFO. When dual-clock mode is chosen, the FIFO being used is DCFIFO. In dual-clock mode, input data and write-side status interfaces use the write side clock domain; the output data and read-side status interfaces use the read-side clock domain.

Device Support

The on-chip FIFO memory supports all Altera® device families except the Hardcopy® series.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the on-chip FIFO memory in SOPC Builder to specify the core configuration. The following sections describe the available options.

FIFO Settings

The following sections outline the settings that pertain to the FIFO as a whole.

Depth

Depth indicates the depth of the FIFO, in Avalon-ST beats or Avalon-MM words. The default depth is 16. When dual clock mode is used, the actual FIFO depth is equal to depth-3. This is due to clock crossing and to avoid FIFO overflow.

Clock Settings

The two options are **Single clock mode** and **Dual clock mode**. In **Single clock mode**, all interface ports use the same clock. In **Dual clock mode**, input data and input side status are on the input clock domain. Output data and output side status are on the output clock domain.

Status Port

The optional status ports are Avalon-MM slaves. To include the optional input side status interface, turn on **Create status interface for input** on the SOPC Builder MegaWizard. For FIFOs whose input and output ports operate in separate clock domains, you can include a second status interface by turning on **Create status interface for output**. Turning on **Enable IRQ for status ports** adds an interrupt signal to the status ports.

FIFO Implementation

This option determines if the FIFO is built from registers or embedded memory blocks. The default is to construct the FIFO from embedded memory blocks.

Interface Parameters

The following sections outline the options for the input and output interfaces.

Input

Available input interfaces are **Avalon-MM** write slave and **Avalon-ST** sink.

Output

Available output interfaces are **Avalon-MM** read slave and **Avalon-ST** source.

Allow Backpressure

When **Allow backpressure** is on, an Avalon-MM interface includes the `waitrequest` signal which is asserted to prevent a master from writing to a full FIFO or reading from an empty FIFO. An Avalon-ST interface includes the `ready` and `valid` signals to prevent underflow and overflow conditions.

Avalon-MM Port Settings

Valid **Data widths** are 8, 16, and 32 bits.

If Avalon-MM is selected for one interface and Avalon-ST for the other, the data width is fixed at 32 bits.

The Avalon-MM interface accesses data 4 bytes at a time. For data widths other than 32 bits, be careful of potential overflow and underflow conditions.

Avalon-ST Port Settings

The following parameters allow you to specify the size and error handling of the Avalon-ST port or ports:

- **Bits per symbol**
- **Symbols per beat**
- **Channel width**
- **Error width**

If the symbol size is not a power of two, it is rounded up to the next power of two. For example, if the **bits per symbol** is 10, the symbol will be mapped to a 16-bit memory location. With 10-bit symbols, the maximum number of **symbols per beat** is two.

Enable packet data provides an option for packet transmission.

Software Programming Model

The following sections describe the software programming model for the on-chip FIFO memory core, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides HAL system library drivers that enable you to access the on-chip FIFO memory core using its HAL API.

HAL System Library Support

The Altera-provided driver implements a HAL device driver that integrates into the HAL system library for Nios II systems. HAL users should access the on-chip FIFO memory via the familiar HAL API, rather than accessing the registers directly.

Software Files

Altera provides the following software files for the on-chip FIFO memory core:

- **altera_avalon_fifo_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_fifo_util.h**—This file defines functions to access the on-chip FIFO memory core hardware. It provides utilities to initialize the FIFO, read and write status, enable flags and read events.
- **altera_avalon_fifo.h**—This file provides the public interface to the on-chip FIFO memory
- **altera_avalon_fifo_util.c**—This file implements the utilities listed in **altera_avalon_fifo_util.h**.

Programming with the On-Chip FIFO Memory

This section describes the low-level software constructs for manipulating the on-chip FIFO memory core hardware. Table 15–3 lists all of the available functions.

Table 15–3. On-Chip FIFO Memory Functions (Part 1 of 2)

Function Name	Description
`altera_avalon_fifo_init()`	Initializes the FIFO.
`altera_avalon_fifo_read_status()`	Returns the integer value of the specified bit of the status register. To read all of the bits at once, use the `ALTERA_AVALON_FIFO_STATUS_ALL` mask.
`altera_avalon_fifo_read_ienable()`	Returns the value of the specified bit of the interrupt enable register. To read all of the bits at once, use the `ALTERA_AVALON_FIFO_EVENT_ALL` mask.
`altera_avalon_fifo_read_almostfull()`	Returns the value of the `almostfull` register.
`altera_avalon_fifo_read_almostempty()`	Returns the value of the `almostempty` register.
`altera_avalon_fifo_read_event()`	Returns the value of the specified bit of the event register. All of the event bits can be read at once by using the `ALTERA_AVALON_FIFO_STATUS_ALL` mask.
`altera_avalon_fifo_read_level()`	Returns the fill level of the FIFO.
`altera_avalon_fifo_clear_event()`	Clears the specified bits and the event register and performs error checking.

Table 15–3. On-Chip FIFO Memory Functions (Part 2 of 2)

Function Name	Description
altera_avalon_fifo_write_ienable()	Writes the specified bits of the interruptenable register and performs error checking.
altera_avalon_fifo_write_almostfull()	Writes the specified value to the almostfull register and performs error checking.
altera_avalon_fifo_write_almostempty()	Writes the specified value to the almostempty register and performs error checking.
altera_avalon_fifo_write_fifo()	Writes the specified data to the write_address.
altera_avalon_fifo_write_other_info()	Writes the packet status information to the write_address. Only valid when the **Enable packet data** option is turned on.
altera_avalon_fifo_read_fifo()	Reads data from the specified read_address.
altera_avalon_fifo_read__other_info()	Reads the packet status information from the specified read_address. Only valid when the **Enable packet data** option is turned on.

Software Control

Table 15–4 provides the register map for the status register. The layout of status register for the input and output interfaces is identical.

Table 15–4. FIFO Status Register Memory Map

offset	31 24	23 16	15 8	7	6	5	4	3	2	1	0
base	fill_level										
base + 1						i_status					
base + 2						event					
base + 3						interrupt enable					
base + 4	almostfull										
base + 5	almostempty										

Table 15–5 outlines the use of the various fields of the status register.

Table 15–5. FIFO Status Field Descriptions (Part 1 of 2)

Field	Type	Description
fill_level	RO	The instantaneous fill level of the FIFO, provided in units of symbols for a FIFO with an Avalon-ST FIFO and words for an Avalon-MM FIFO.
i_status	RO	A 6-bit register that shows the FIFO's instantaneous status. See Table 15–6 for the meaning of each bit field.
event	RW1C	A 6-bit register with exactly the same fields as i_status. When a bit in the i_status register is set, the same bit in the event register is set. The bit in the event register is only cleared when software writes a 1 to that bit.
interruptenable	RW	A 6-bit interrupt enable register with exactly the same fields as the event and i_status registers. When a bit in the event register transitions from a 0 to a 1, and the corresponding bit in interruptenable is set, the master Is interrupted.

Table 15–5. FIFO Status Field Descriptions (Part 2 of 2)

Field	Type	Description
almostfull	RW	A threshold level used for interrupts and status. Can be written by the Avalon-MM status master at any time. The default threshold value for DCFIFO is Depth-4. The default threshold value for SCFIFO is Depth-1. The valid range of the threshold value is from 1 to the default. 1 is used when attempting to write a value smaller than 1. The default is used when attempting to write a value larger than the default.
almostempty	RW	A threshold level used for interrupts and status. Can be written by the Avalon-MM status master at any time. The default threshold value for DCFIFO is 1. The default threshold value for SCFIFO is 1. The valid range of the threshold value is from 1 to the maximum allowable almostfull threshold. 1 is used when attempting to write a value smaller than 1. The maximum allowable is used when attempting to write a value larger than the maximum allowable.

Table 15–6 describes the instantaneous status bits.

Table 15–6. Status Bit Field Descriptions

Bit(s)	Name	Description
0	FULL	Has a value of 1 if the FIFO is currently full.
1	EMPTY	Has a value of 1 if the FIFO is currently empty.
2	ALMOSTFULL	Has a value of 1 if the fill level of the FIFO is greater than the almostfull value.
3	ALMOSTEMPTY	Has a value of 1 if the fill level of the FIFO is less than the almostempty value.
4	OVERFLOW	Is set to 1 for 1 cycle every time the FIFO overflows. The FIFO overflows when an Avalon write master writes to a full FIFO. OVERFLOW is only valid when **Allow backpressure** is off.
5	UNDERFLOW	Is set to 1 for 1 cycle every time the FIFO underflows. The FIFO underflows when an Avalon read master reads from an empty FIFO. UNDERFLOW is only valid when **Allow backpressure** is off.

Table 15–7 lists the bit fields of the event register. These fields are identical to those in the status register and are set at the same time; however, these fields are only cleared when software writes a one to clear (W1C). The event fields can be used to determine if a particular event has occurred.

Table 15–7. Event Bit Field Descriptions

Bit(s)	Name	Description
1	E_FULL	Has a value of 1 if the FIFO has been full and the bit has not been cleared by software.
0	E_EMPTY	Has a value of 1 if the FIFO has been empty and the bit has not been cleared by software.
3	E_ALMOSTFULL	Has a value of 1 if the fill level of the FIFO has been greater than the almostfull threshold value and the bit has not been cleared by software.
2	E_ALMOSTEMPTY	Has a value of 1 if the fill level of the FIFO has been less than the almostempty value and the bit has not been cleared by software.
4	E_OVERFLOW	Has a value of 1 if the FIFO has overflowed and the bit has not been cleared by software.
5	E_UNDERFLOW	Has a value of 1 if the FIFO has underflowed and the bit has not been cleared by software.

Table 15–8 provides a mask for the six STATUS fields. When a bit in the event register transitions from a zero to a one, and the corresponding bit in the interruptenable register is set, the master is interrupted.

Table 15–8. InterruptEnable Bit Field Descriptions

Bit(s)	Name	Description
1	`IE_FULL`	Enables an interrupt if the FIFO is currently full.
0	`IE_EMPTY`	Enables an interrupt if the FIFO is currently empty.
3	`IE_ALMOSTFULL`	Enables an interrupt if the fill level of the FIFO is greater than the value of the `almostfull` register.
2	`IE_ALMOSTEMPTY`	Enables an interrupt if the fill level of the FIFO is less than the value of the `almostempty` register.
4	`IE_OVERFLOW`	Enables an interrupt if the FIFO overflows. The FIFO overflows when an Avalon write master writes to a full FIFO.
5	`IE_UNDERFLOW`	Enables an interrupt if the FIFO underflows. The FIFO underflows when an Avalon read master reads from an empty FIFO.
6	`ALL`	Enables all 6 status conditions to interrupt.

Macros to access all of the registers are defined in **altera_avalon_fifo_regs.h.** For example, this file includes the following macros to access the `status` register.

```
#define ALTERA_AVALON_FIFO_LEVEL_REG          0
#define ALTERA_AVALON_FIFO_STATUS_REG         1
#define ALTERA_AVALON_FIFO_EVENT_REG          2
#define ALTERA_AVALON_FIFO_IENABLE_REG        3
#define ALTERA_AVALON_FIFO_ALMOSTFULL_REG     4
#define ALTERA_AVALON_FIFO_ALMOSTEMPTY_REG    5
```

For a complete list of predefined macros and utilities to access the on-chip FIFO hardware, see:
<install_dir>\quartus\sopc_builder\components\altera_avalon_fifo\HAL\inc\alatera_avalon_fifo.h and
<install_dir>\quartus\sopc_builder\components\altera_avalon_fifo\HAL\inc\alatera_avalon_fifo_util.h.

Software Example

Example 15–1 shows sample codes for the core.

Example 15–1. Sample Code for the On-Chip FIFO Memory (Part 1 of 2)

```
/***********************************************************************/
//Includes
#include "altera_avalon_fifo_regs.h"
#include "altera_avalon_fifo_util.h"
#include "system.h"
#include "sys/alt_irq.h"
#include <stdio.h>
#include <stdlib.h>

#define ALMOST_EMPTY 2
#define ALMOST_FULL OUTPUT_FIFO_OUT_FIFO_DEPTH-5

volatile int input_fifo_wrclk_irq_event;

void print_status(alt_u32 control_base_address)
{
   printf("--------------------------------------\n");
   printf("LEVEL = %u\n", altera_avalon_fifo_read_level(control_base_address) );
   printf("STATUS = %u\n", altera_avalon_fifo_read_status(control_base_address,
ALTERA_AVALON_FIFO_STATUS_ALL) );
   printf("EVENT = %u\n", altera_avalon_fifo_read_event(control_base_address,
ALTERA_AVALON_FIFO_EVENT_ALL) );
   printf("IENABLE = %u\n", altera_avalon_fifo_read_ienable(control_base_address,
ALTERA_AVALON_FIFO_IENABLE_ALL) );
   printf("ALMOSTEMPTY = %u\n",
altera_avalon_fifo_read_almostempty(control_base_address) );
   printf("ALMOSTFULL = %u\n\n",
altera_avalon_fifo_read_almostfull(control_base_address));
}

static void handle_input_fifo_wrclk_interrupts(void* context, alt_u32 id)
{
   /* Cast context to input_fifo_wrclk_irq_event's type. It is important
   * to declare this volatile to avoid unwanted compiler optimization.
   */
   volatile int* input_fifo_wrclk_irq_event_ptr = (volatile int*) context;

   /* Store the value in the FIFO's irq history register in *context. */
   *input_fifo_wrclk_irq_event_ptr =
altera_avalon_fifo_read_event(INPUT_FIFO_IN_CSR_BASE, ALTERA_AVALON_FIFO_EVENT_ALL);
   printf("Interrupt Occurs for %#x\n", INPUT_FIFO_IN_CSR_BASE);
   print_status(INPUT_FIFO_IN_CSR_BASE);

   /* Reset the FIFO's IRQ History register. */
   altera_avalon_fifo_clear_event(INPUT_FIFO_IN_CSR_BASE,
ALTERA_AVALON_FIFO_EVENT_ALL);
}

/* Initialize the fifo */
static int init_input_fifo_wrclk_control()
{
   int return_code = ALTERA_AVALON_FIFO_OK;

   /* Recast the IRQ History pointer to match the alt_irq_register() function
   * prototype. */
   void* input_fifo_wrclk_irq_event_ptr = (void*) &input_fifo_wrclk_irq_event;
   /* Enable all interrupts. */
```

Example 15–1. Sample Code for the On-Chip FIFO Memory (Part 2 of 2)

```
    /* Clear event register, set enable all irq, set almostempty and
    almostfull threshold */
    return_code = altera_avalon_fifo_init(INPUT_FIFO_IN_CSR_BASE,
                                           0, // Disabled interrupts
                                           ALMOST_EMPTY,
                                           ALMOST_FULL);
    /* Register the interrupt handler. */
    alt_irq_register( INPUT_FIFO_IN_CSR_IRQ,
input_fifo_wrclk_irq_event_ptr, handle_input_fifo_wrclk_interrupts );
    return return_code;
}
```

On-Chip FIFO Memory API

This section describes the application programming interface (API) for the on-chip FIFO memory core.

altera_avalon_fifo_init()

Prototype:	`int altera_avalon_fifo_init(alt_u32 address, alt_u32 ienable, alt_u32 emptymark, alt_u32 fullmark)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave `ienable`—the value to write to the `interruptenable` register `emptymark`—the value for the almost empty threshold level `fullmark`—the value for the almost full threshold level
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_EVENT_CLEAR_ERROR` for clear errors, `ALTERA_AVALON_FIFO_IENABLE_WRITE_ERROR` for interrupt enable write errors, `ALTERA_AVALON_FIFO_THRESHOLD_WRITE_ERROR` for errors writing the `almostfull` and `almostempty` registers.
Description:	Clears the `event` register, writes the `interruptenable` register, and sets the `almostfull` register and `almostempty` registers.

altera_avalon_fifo_read_status()

Prototype:	`int altera_avalon_fifo_read_status(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave `mask`—masks the read value from the `status` register
Returns:	Returns the masked bits of the addressed register.
Description:	Gets the addressed register bits—the `AND` of the value of the addressed register and the mask.

altera_avalon_fifo_read_ienable()

Prototype: `int altera_avalon_fifo_read_ienable(alt_u32 address, alt_u32 mask)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

`mask`—masks the read value from the `interruptenable` register

Returns: Returns the logical AND of the `interruptenable` register and the mask.

Description: Gets the logical AND of the `interruptenable` register and the mask.

altera_avalon_fifo_read_almostfull()

Prototype: `int altera_avalon_fifo_read_almostfull(alt_u32 address)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

Returns: Returns the value of the `almostfull` register.

Description: Gets the value of the `almostfull` register.

altera_avalon_fifo_read_almostempty()

Prototype: `int altera_avalon_fifo_read_almostempty(alt_u32 address)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

Returns: Returns the value of the `almostempty` register.

Description: Gets the value of the `almostempty` register.

altera_avalon_fifo_read_event()

Prototype: `int altera_avalon_fifo_read_event(alt_u32 address, alt_u32 mask)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

`mask`—masks the read value from the `event` register

Returns: Returns the logical AND of the `event` register and the mask.

Description: Gets the logical AND of the `event` register and the mask. To read single bits of the event register use the single bit masks, for example: `ALTERA_AVALON_FIFO_FIFO_EVENT_F_MSK`.
To read the entire `event` register use the full mask: `ALTERA_AVALON_FIFO_EVENT_ALL`.

altera_avalon_fifo_read_level()

Prototype:	`int altera_avalon_fifo_read_level(alt_u32 address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave
Returns:	Returns the fill level of the FIFO.
Description:	Gets the fill level of the FIFO.

altera_avalon_fifo_clear_event()

Prototype:	`int altera_avalon_fifo_clear_event(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave `mask`—the mask to use for bit-clearing (1 means clear this bit, 0 means do not clear)
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_EVENT_CLEAR_ERROR` if unsuccessful.
Description:	Clears the specified bits of the `event` register.

altera_avalon_fifo_write_ienable()

Prototype:	`int altera_avalon_fifo_write_ienable(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control slave `mask`—the value to write to the `interruptenable` register. See **altera_avalon_fifo_regs.h** for individual interrupt bit masks.
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_IENABLE_WRITE_ERROR` if unsuccessful.
Description:	Writes the specified bits of the `interruptenable` register.

altera_avalon_fifo_write_almostfull()

Prototype: `int altera_avalon_fifo_write_almostfull(alt_u32 address, alt_u32 data)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

`data`—the value for the almost full threshold level

Returns: Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_THRESHOLD_WRITE_ERROR` if unsuccessful.

Description: Writes `data` to the `almostfull` register.

altera_avalon_fifo_write_almostempty()

Prototype: `int altera_avalon_fifo_write_almostempty(alt_u32 address, alt_u23 data)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `address`—the base address of the FIFO control slave

`data`—the value for the almost empty threshold level

Returns: Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_THRESHOLD_WRITE_ERROR` if unsuccessful.

Description: Writes `data` to the `almostempty` register.

altera_avalon_write_fifo()

Prototype: `int altera_avalon_write_fifo(alt_u32 write_address, alt_u32 ctrl_address, alt_u32 data)`

Thread-safe: No.

Available from ISR: No.

Include: <**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>

Parameters: `write_address`—the base address of the FIFO write slave

`ctrl_address`—the base address of the FIFO control slave

`data`—the value to write to address offset 0 for Avalon-MM to Avalon-ST transfers, the value to write to the single address available for Avalon-MM to Avalon-MM transfers. See the *Avalon Interface Specifications* for the data ordering.

Returns: Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_FULL` if unsuccessful.

Description: Writes `data` to the specified address if the FIFO is not full.

altera_avalon_write_other_info()

Prototype:	`int altera_avalon_write_other_info(alt_u32 write_address, alt_u32 ctrl_address, alt_u32 data)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`write_address`—the base address of the FIFO write slave `ctrl_address`—the base address of the FIFO control slave `data`—the packet status information to write to address offset 1 of the Avalon interface. See the *Avalon Interface Specifications* for the ordering of the packet status information.
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_FULL` if unsuccessful.
Description:	Writes the packet status information to the `write_address`. Only valid when **Enable packet data** is on.

altera_avalon_fifo_read_fifo()

Prototype:	`int altera_avalon_fifo_read_fifo(alt_u32 read_address, alt_u32 ctrl_address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`read_address`—the base address of the FIFO read slave `ctrl_address`—the base address of the FIFO control slave
Returns:	Returns the data from address offset 0, or 0 if the FIFO is empty.
Description:	Gets the data addressed by `read_address`.

altera_avalon_fifo_read_other_info()

Prototype:	`int altera_avalon_fifo_read_other_info(alt_u32 read_address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`read_address`—the base address of the FIFO read slave
Returns:	Returns the packet status information from address offset 1 of the Avalon interface. See the *Avalon Interface Specifications* for the ordering of the packet status information.
Description:	Reads the packet status information from the specified `read_address`. Only valid when **Enable packet data** is on.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 15–9 shows the revision history for this chapter.

Table 15–9. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Updated the description of the function altera_avalon_fifo_read_status().	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

16. Avalon-ST Multi-Channel Shared Memory FIFO Core

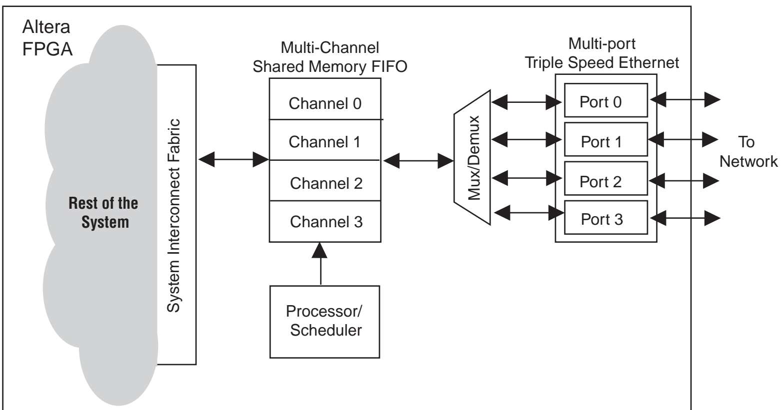
QII55015-9.0.0

Core Overview

The Avalon® Streaming (Avalon-ST) Multi-Channel Shared Memory FIFO core is a FIFO buffer with Avalon-ST data interfaces. The core, which supports up to 16 channels, is a contiguous memory space with dedicated segments of memory allocated for each channel. Data is delivered to the output interface in the same order it was received on the input interface for a given channel.

Figure 16–1 shows an example of how the core is used in a system. In this example, the core is used to buffer data going into and coming from a four-port Triple Speed Ethernet MegaCore function. A processor, if used, can request data for a particular channel to be delivered to the Triple Speed Ethernet MegaCore function.

Figure 16–1. Multi-Channel Shared Memory FIFO in a System—An Example



The Avalon-ST Multi-Channel Shared FIFO core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated systems.

This chapter contains the following sections:

- "Performance and Resource Utilization" on page 16–2
- "Functional Description" on page 16–3
- "Instantiating the Core in SOPC Builder" on page 16–5
- "Device Support" on page 16–5
- "Software Programming Model" on page 16–5

Performance and Resource Utilization

This section lists the resource utilization and performance data for various Altera device families. The estimates are obtained by compiling the core using the Quartus® II software.

Table 16–1 shows the resource utilization and performance data for a Stratix II GX device (EP2SGX130GF1508I4).

Table 16–1. Memory Utilization and Performance Data for Stratix II GX Devices

Channels	ALUTs	Logic Registers	Memory Blocks			f_{MAX} (MHz)
			M512	M4K	M-RAM	
4	559	382	0	0	1	> 125
12	1617	1028	0	0	6	> 125

Table 16–2 shows the resource utilization and performance data for a Stratix III device (EP3SL340F1760C3). The performance of the MegaCore function in Stratix IV devices is similar to Stratix III devices.

Table 16–2. Memory Utilization and Performance Data for Stratix III Devices

Channels	ALUTs	Logic Registers	Memory Blocks			f_{MAX} (MHz)
			M9K	M144K	MLAB	
4	557	345	37	0	0	> 125
12	1741	1028	0	24	0	> 125

Table 16–3 shows the resource utilization and performance data for a Cyclone III device (EP3C120F780I7).

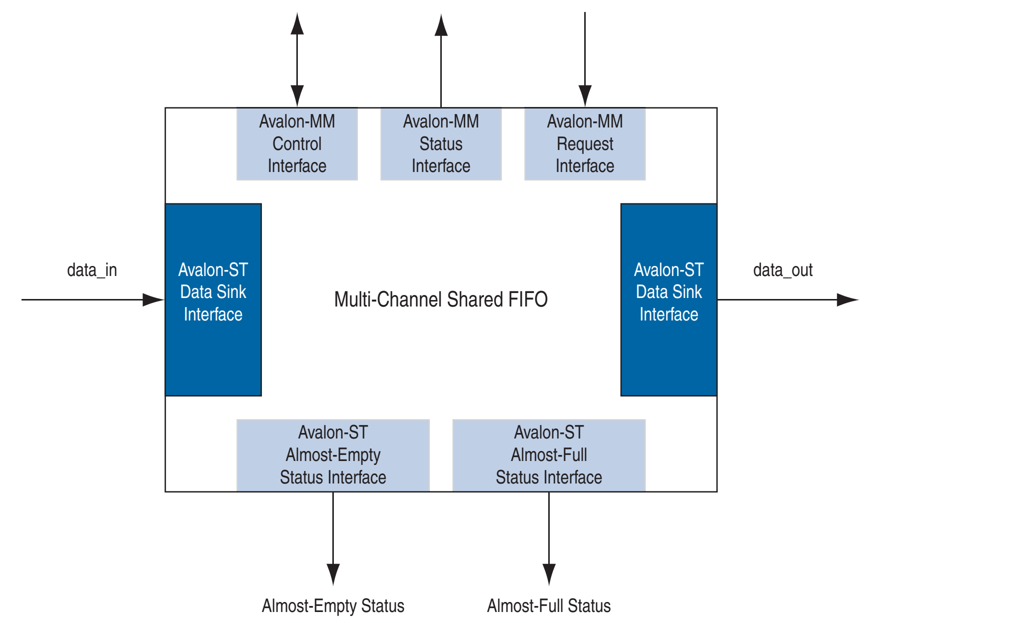
Table 16–3. Memory Utilization and Performance Data for Cyclone III Devices

Channels	Total Logic Elements	Total Registers	Memory M9K	f_{MAX} (MHz)
4	711	346	37	> 125
12	2284	1029	412	> 125

Functional Description

Figure 16–2 shows a block diagram of the Avalon-ST Multi-Channel Shared FIFO core.

Figure 16–2. Avalon-ST Multi-Channel Shared Memory FIFO Core



Interfaces

This section describes the core's interfaces.

Avalon-ST Interfaces

The core includes Avalon-ST interfaces for transferring data and almost-full status.

Table 16–4 shows the properties of the Avalon-ST data interfaces.

Table 16–4. Properties of Avalon-ST Interfaces

Feature	Property	
	Data Interfaces	Status Interfaces
Backpressure	Ready latency = 0.	Not supported.
Data Width	Configurable.	Data width = 2 bits. Symbols per beat = 1.
Channel	Supported, up to 16 channels.	Supported, up to 16 channels.
Error	Configurable.	Not used.
Packet	Supported.	Not supported.

Avalon-MM Interfaces

The core can have up to three Avalon-MM interfaces:

- **Avalon-MM control interface**—Allows master peripherals to set and access almost-full and almost-empty thresholds. The same set of thresholds is used by all channels.
- **Avalon-MM status interface—**Provides the FIFO fill level for a given channel. The FIFO fill level represents the amount of data in the FIFO at any given time. The fill level is available on the `readdata` bus one clock cycle after the read request is received.
- **Avalon-MM request interface—**Allows master peripherals to request data for a given channel. This interface is implemented only when the parameter **Use Request** is set to 1. The `request_address` signal contains the channel number. Only one FIFO entry is returned for each request.

For more information about Avalon interfaces, refer to the *Avalon Interface Specifications*.

Operation

The Avalon-ST Multi-Channel Shared FIFO core allocates dedicated memory segments within the FIFO for each channel, and is implemented such that the memory segments occupy a single memory block. The depth of each memory segment is determined by the parameter **FIFO depth**. If the core is configured to support more than one channel, the Avalon-MM request interface must be implemented to allow master peripherals to request data for a specific channel. Otherwise, only channel 0 is accessible.

When a request is received on the core's Avalon-MM request interface, the requested data is available on the Avalon-ST data source interface after three clock cycles. Only one word of data can be requested at a time. The core delivers the data to the Avalon-ST data source interface after a full packet is received.

The core does not implement any mechanism to accept incoming requests while processing. Once the core starts processing a request, incoming requests are dropped until the current one completes and data is transferred to the requesting component. Packets received on the Avalon-ST sink interface are dropped if the error signal is asserted.

You can configure almost-full thresholds to manage FIFO overflow. The current threshold status for each channel is available from the core's Avalon-ST status interfaces in a round-robin fashion. For example, if the threshold status for channel 0 is available on the interface in clock cycle *n*, the threshold status for channel 1 is available in clock cycle *n+1* and so forth.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the Avalon-ST Multi-Channel Shared FIFO core in SOPC Builder to add the core to a system.

Table 16–5 lists and describes the parameters you can configure.

Table 16–5. Configurable Parameters

Parameter	Legal Values	Description
Number of channels	1, 2, 4, 8, and 16	The total number of channels supported on the Avalon-ST data interfaces.
Symbols per beat	1–32	The number of symbols transferred in a beat on the Avalon-ST data interfaces
Bits per symbol	1–32	The symbol width in bits on the Avalon-ST data interfaces.
Error width	0–32	The width of the `error` signal on the Avalon-ST data interfaces.
FIFO depth	$2–2^{32}$	The depth of each memory segment allocated for a channel. The value must be a multiple of 2.
Use request	0 or 1	Setting this parameter to 1 implements the Avalon-MM request interface. If the request interface is disabled, only channel 0 can be used.
Address width	1–32	The width of the FIFO address. This parameter is determined by the parameter **FIFO depth**; **FIFO depth** = $2^{\text{Address Width}}$.

Device Support

The Avalon-ST Multi-Channel Shared FIFO core supports all Altera device families.

Software Programming Model

The following sections describe the software programming model for the Avalon-ST Multi-Channel Shared FIFO core.

HAL System Library Support

The Altera-provided driver implements a HAL device driver that integrates into the HAL system library for Nios II systems. HAL users should access the Avalon-ST Multi-Channel Shared FIFO core via the familiar HAL API and the ANSI C standard library.

Register Map

You can update and access the FIFO thresholds via the Avalon-MM control interface. Table 16–6 shows the register map for the control interface.

Table 16–6. Control Interface Register Map

Offset	Name	Access	Description
Base + 0	`Almost_Full_Threshold`	RW	The value of the primary almost-full threshold. The bit `Almost_full_data[0]` on the Avalon-ST almost-full status interface is set to 1 when the FIFO level is greater than or equal to this threshold.
Base + 8	`Almost_Full2_Threshold`	RW	The value of the secondary almost-full threshold. The bit `Almost_full_data[1]` on the Avalon-ST almost-full status interface is set to 1 when the FIFO level is greater than or equal to this threshold.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 16–7 shows the revision history for this chapter.

Table 16–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

Section III. Transport and Communication

This section describes communication and transport peripherals provided for SOPC Builder systems.

This section includes the following chapters:

- Chapter 17, Avalon Streaming Channel Multiplexer and Demultiplexer Cores
- Chapter 18, Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores
- Chapter 19, Avalon Packets to Transactions Converter Core
- Chapter 20, Avalon-ST Round Robin Scheduler Core

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

17. Avalon Streaming Channel Multiplexer and Demultiplexer Cores

QII55004-9.0.0

Core Overview

The Avalon® streaming (Avalon-ST) channel multiplexer core receives data from a number of input interfaces and multiplexes the data into a single output interface, using the optional `channel` signal to indicate which input the output data is from. The Avalon-ST channel demultiplexer core receives data from a channelized input interface and drives that data to multiple output interfaces, where the output interface is selected by the input `channel` signal.

The multiplexer and demultiplexer can transfer data between interfaces on cores that support the unidirectional flow of data. The multiplexer and demultiplexer allow you to create multiplexed or de-multiplexer datapaths without having to write custom HDL code to perform these functions. The multiplexer includes a round-robin scheduler. Both cores are SOPC Builder-ready and integrate easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Multiplexer" on page 17–2
- "Demultiplexer" on page 17–4
- "Device Support" on page 17–6
- "Hardware Simulation Considerations" on page 17–6
- "Software Programming Model" on page 17–6

Resource Usage and Performance

Resource utilization for the cores depends upon the number of input and output interfaces, the width of the datapath and whether the streaming data uses the optional packet protocol. For the multiplexer, the parameterization of the scheduler also effects resource utilization. Table 17–1 provides estimated resource utilization for eleven different configurations of the multiplexer.

Table 17–1. Multiplexer Estimated Resource Usage and Performance (Part 1 of 2)

No. of Inputs	Data Width	Scheduling Size (Cycles)	Stratix® II and Stratix II GX (Approximate LEs)		Cyclone® II		Stratix	
			f_{MAX} (MHz)	ALM Count	f_{MAX} (MHz)	Logic Cells	f_{MAX} (MHz)	Logic Cells
2	Y	1	500	31	420	63	422	80
2	Y	2	500	36	417	60	422	58
2	Y	32	451	43	364	68	360	49
8	Y	2	401	150	257	233	228	298
8	Y	32	356	151	219	207	211	123
16	Y	2	262	333	174	533	170	284
16	Y	32	310	337	161	471	157	277
2	N	1	500	23	400	48	422	52

Table 17–1. Multiplexer Estimated Resource Usage and Performance (Part 2 of 2)

No. of Inputs	Data Width	Scheduling Size (Cycles)	Stratix® II and Stratix II GX (Approximate LEs)		Cyclone® II		Stratix	
			f_{MAX} (MHz)	ALM Count	f_{MAX} (MHz)	Logic Cells	f_{MAX} (MHz)	Logic Cells
2	N	9	500	30	420	52	422	56
11	N	9	292	275	197	397	182	287
16	N	9	262	295	182	441	179	224

Table 17–2 provides estimated resource utilization for six different configurations of the demultiplexer. The core operating frequency varies with the device, the number of interfaces and the size of the datapath.

Table 17–2. Demultiplexer Estimated Resource Usage

No. of Inputs	Data Width (Symbols per Beat)	Stratix II (Approximate LEs)		Cyclone II		Stratix II GX (Approximate LEs)	
		f_{MAX} (MHz)	ALM Count	f_{MAX} (MHz)	Logic Cells	f_{MAX} (MHz)	Logic Cells
2	1	500	53	400	61	399	44
15	1	349	171	235	296	227	273
16	1	363	171	233	294	231	290
2	2	500	85	392	97	381	71
15	2	352	247	213	450	210	417
16	2	328	280	218	451	222	443

Multiplexer

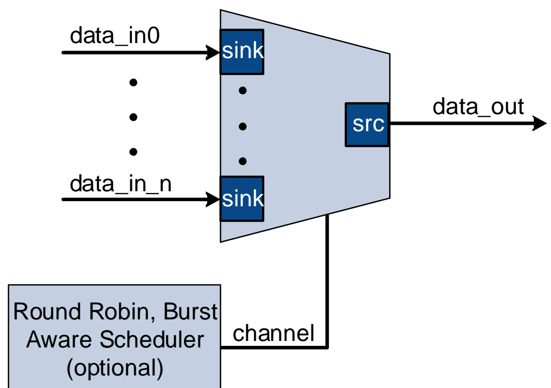
This section describes the hardware structure and functionality of the multiplexer component.

Functional Description

The Avalon-ST multiplexer takes data from a number of input data interfaces, and multiplexes the data onto a single output interface. The multiplexer includes a simple, round-robin scheduler that selects from the next input interface that has data. Each input interface has the same width as the output interface, so that all other input interfaces are backpressured when the multiplexer is carrying data from a different input interface.

The multiplexer includes an optional `channel` signal that enables each input interface to carry channelized data. When the `channel` signal is present on input interfaces, the multiplexer adds $\log_2$(num_input_interfaces) bits to make the output channel signal, such that the output channel signal has all of the bits of the input channel plus the bits required to indicate which input interface each cycle of data is from. These bits are appended to either the most or least significant bits of the output `channel` signal as specified in the SOPC Builder MegaWizard™ Interface.

Figure 17–1. Multiplexer



The internal scheduler considers one input interface at a time, selecting it for transfer. Once an input interface has been selected, data from that input interface is sent until one of the following scenarios occurs:

- The specified number of cycles have elapsed.
- The input interface has no more data to send and `valid` is deasserted on a ready cycle.
- When packets are supported, `endofpacket` is asserted.

Input Interfaces

Each input interface is an Avalon-ST data interface that optionally supports packets. The input interfaces are identical; they have the same symbol and data widths, error widths, and channel widths.

Output Interface

The output interface carries the multiplexed data stream with data from all of the inputs. The symbol, data, and error widths are the same as the input interfaces. The width of the `channel` signal is the same as the input interfaces, with the addition of the bits needed to indicate the input each datum was from.

Instantiating the Multiplexer in SOPC Builder

Use the MegaWizard Interface for the multiplexer core in SOPC Builder to specify the core configuration. The following sections list the available options in the MegaWizard Interface.

Functional Parameters

You can configure the following options for the multiplexer:

- **Number of Input Ports**—The number of input interfaces that the multiplexer supports. Valid values are 2–16.
- **Scheduling Size (Cycles)**—The number of cycles that are sent from a single channel before changing to the next channel.

- **Use Packet Scheduling**—When this option is on, the multiplexer only switches the selected input interface on packet boundaries. Hence, packets on the output interface are not interleaved.
- **Use high bits to indicate source port**—When this option is on, the high bits of the output channel signal are used to indicate the input interface that the data came from. For example, if the input interfaces have 4-bit channel signals, and the multiplexer has 4 input interfaces, the output interface has a 6-bit channel signal. If this parameter is true, bits [5:4] of the output channel signal indicate the input interface the data is from, and bits [3:0] are the channel bits that were presented at the input interface.

Output Interface

You can configure the following options for the output interface:

- **Data Bits Per Symbol**—The number of bits per symbol for the input and output interfaces. Valid values are 1–32 bits.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat (transfer). Valid values are 1–32.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Channel Signal Width (bits)**—The number of bits used for the `channel` signal for input interfaces. A value of 0 indicates that input interfaces do not have channels. A value of 4 indicates that up to 16 channels share the same input interface. The input channel can have a width between 0–31 bits. A value of 0 means that the optional `channel` signal is not used.
- **Error Signal Width (bits)**—The width of the `error` signal for input and output interfaces. A value of 0 means the `error` signal is not used.

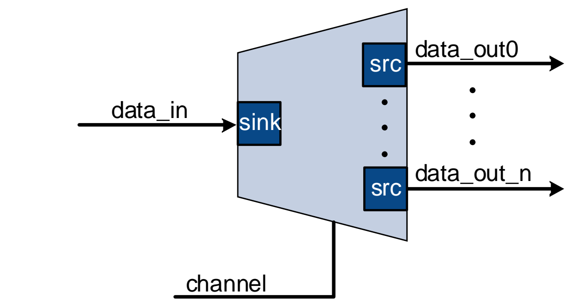
Demultiplexer

This section describes the hardware structure and functionality of the demultiplexer component.

Functional Description

That Avalon-ST demultiplexer takes data from a channelized input data interface and provides that data to multiple output interfaces, where the output interface selected for a particular transfer is specified by the input `channel` signal. The data is delivered to the output interfaces in the same order it was received at the input interface, regardless of the value of `channel`, `packet`, `frame`, or any other signal. Each of the output interfaces has the same width as the input interface, so each output interface is idle when the demultiplexer is driving data to a different output interface. The demultiplexer uses $\log_2$(num_output_interfaces) bits of the `channel` signal to select the output to which to forward the data; the remainder of the channel bits are forwarded to the appropriate output interface unchanged.

Figure 17–2. Demultiplexer



Input Interface

Each input interface is an Avalon-ST data interface that optionally supports packets.

Output Interfaces

Each output interface carries data from a subset of channels from the input interface. Each output interface is identical; all have the same symbol and data widths, error widths, and channel widths. The symbol, data, and error widths are the same as the input interface. The width of the `channel` signal is the same as the input interface, without the bits that were used to select the output interface.

Instantiating the Demultiplexer in SOPC Builder

Use the MegaWizard Interface for the demultiplexer core in SOPC Builder to specify the core configuration. The following sections list the available options in the MegaWizard Interface.

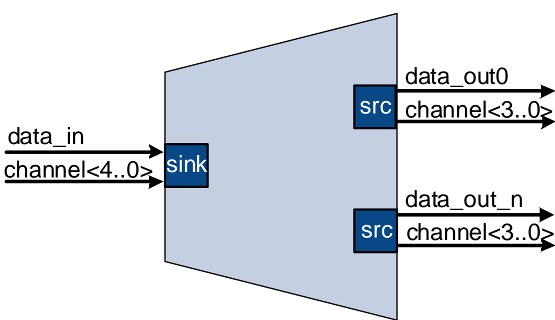
Functional Parameters

You can configure the following options for the demultiplexer as a whole:

- **Number of Output Ports**—The number of output interfaces that the multiplexer supports Valid values are 2–16.
- **High channel bits select output**—When this option is on, the high bits of the input channel signal are used by the de-multiplexing function and the low order bits are passed to the output. When this option is off, the low order bits are used and the high order bits are passed through.

  The following example illustrates the significance of the location of these signals. In Figure 17–3 there is one input interface and two output interfaces. If the low-order bits of the channel signal select the output interfaces, the even channels goes to channel 0 and the odd channels goes to channel 1. If the high-order bits of the channel signal select the output interface, channels 0–7 goes to channel 0 and channels 8–15 goes to channel 1.

Figure 17–3. Select Bits for Demultiplexer



Input Interface

You can configure the following options for the input interface:

- **Data Bits Per Symbol**—The number of bits per symbol for the input and output interfaces. Valid values are 1 to 32 bits.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat (transfer). Valid values are 1 to 32.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Channel Signal Width (bits)**—The number of bits used for the `channel` signal for output interfaces. A value of 0 means that output interfaces do not use the optional `channel` signal.
- **Error Signal Width (bits)**—The width of the `error` signal for input and output interfaces. A value of 0 means the `error` signal is not unused.

Device Support

The Avalon Streaming Channel Multiplexer and Demultiplexer cores support all Altera device families.

Hardware Simulation Considerations

The multiplexer and demultiplexer components do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard SOPC Builder simulation flow to simulate the component design files inside an SOPC Builder system.

Software Programming Model

The multiplexer and demultiplexer components do not have any user-visible control or status registers. Therefore, software cannot control or configure any aspect of the multiplexer or de-multiplexer at run-time. The components cannot generate interrupts.

Document Revision History

Table 17–3 shows the revision history for this chapter.

Table 17–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. Added parameter **Include Packet Support**.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

18. Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores

QII55012-9.0.0

Core Overview

The Avalon® Streaming (Avalon-ST) Bytes to Packets and Packets to Bytes Converter cores allow an arbitrary stream of packets to be carried over a byte interface, by encoding packet-related control signals such as `startofpacket` and `endofpacket` into byte sequences.The Avalon-ST Packets to Bytes Converter core encodes packet control and payload as a stream of bytes. The Avalon-ST Bytes to Packets Converter core accepts an encoded stream of bytes, and converts it into a stream of packets.

The SPI Slave to Avalon Master Bridge and JTAG to Avalon Master Bridge are examples of how the cores are used. For more information about the bridge, refer to the *SPI Slave/JTAG to Avalon Master Bridge Cores chapter* in volume 5 of the *Quartus II Handbook*.

Both of these cores are SOPC Builder-ready and integrate easily into any SOPC Builder-generated system.

This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 18–3
- "Device Support" on page 18–4

Functional Description

Figure 18–1 and Figure 18–2 show block diagrams of the Avalon-ST Bytes to Packets and Packets to Bytes Converter cores.

Figure 18–1. Avalon-ST Bytes to Packets Converter Core

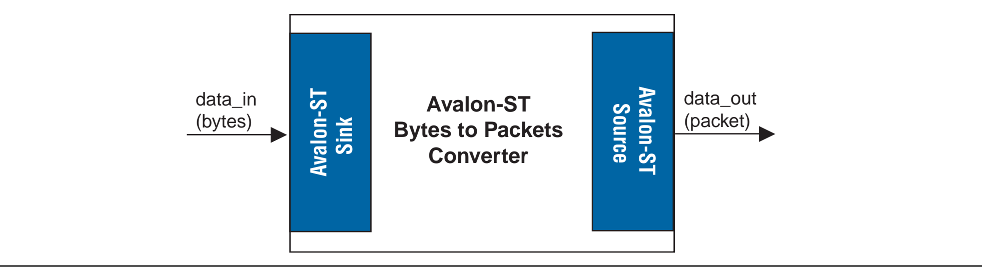
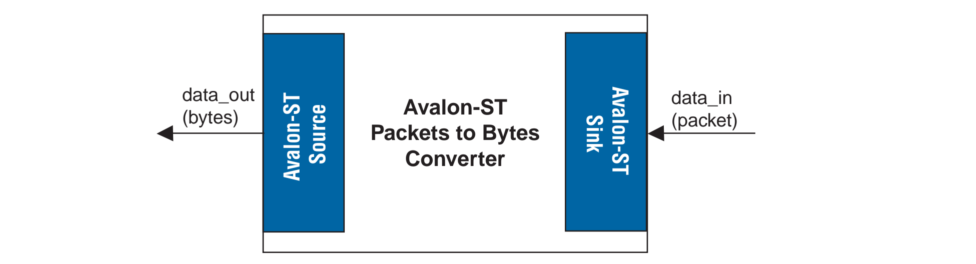


Figure 18–2. Avalon-ST Packets to Bytes Converter Core



Interfaces

Table 18–1 shows the properties of the Avalon-ST interfaces.

Table 18–1. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Data width = 8 bits; Bits per symbol = 8.
Channel	Supported, up to 255 channels.
Error	Not used.
Packet	Supported only on the Avalon-ST Bytes to Packet Converter core's source interface and the Avalon-ST Packet to Bytes Converter core's sink interface.

For more information about Avalon-ST interfaces, refer to the *Avalon Interface Specifications*.

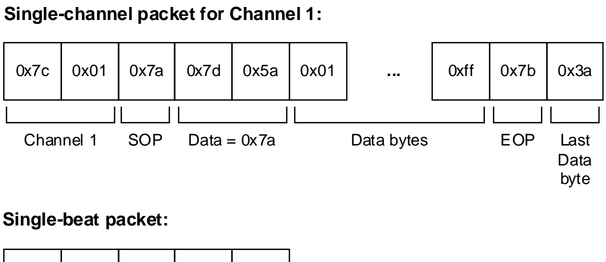
Operation—Avalon-ST Bytes to Packets Converter Core

The Avalon-ST Bytes to Packets Converter core receives streams of bytes and transforms them into packets. When parsing incoming bytestreams, the core decodes special characters in the following manner, with higher priority operations listed first:

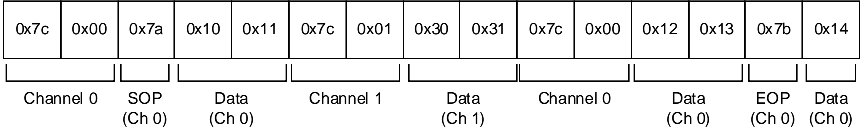
- Escape (`0x7d`)—The core drops the byte. The next byte is XORed with `0x20`.
- Start of packet (`0x7a`)—The core drops the byte and marks the next payload byte as the start of a packet by asserting the `startofpacket` signal on the Avalon-ST source interface.
- End of packet (`0x7b`)—The core drops the byte and marks the following byte as the end of a packet by asserting the `endofpacket` signal on the Avalon-ST source interface. For single beat packets, both the `startofpacket` and `endofpacket` signals are asserted in the same clock cycle.
- Channel number indicator (`0x7c`)—The core drops the byte and takes the next non-special character as the channel number.

Figure 18–3 shows examples of bytestreams.

Figure 18–3. Examples of Bytestreams



Interleaved channels in a packet:



Operation—Avalon-ST Packets to Bytes Converter Core

The Avalon-ST Packets to Bytes Converter core receives packetized data and transforms the packets to bytestreams. The core constructs outgoing bytestreams by inserting appropriate special characters in the following manner and sequence:

- If the `startofpacket` signal on the core's source interface is asserted, the core inserts the following special characters:
  - Channel number indicator (`0x7c`).
  - Channel number, escaping it if required.
  - Start of packet (`0x7a`).
- If the `endofpacket` signal on the core's source interface is asserted, the core inserts an end of packet (`0x7b`) before the last byte of data.
- If the `channel` signal on the core's source interface changes to a new value within a packet, the core inserts a channel number indicator (`0x7c`) followed by the new channel number.
- If a data byte is a special character, the core inserts an escape (`0x7d`) followed by the data XORed with `0x20`.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the Avalon-ST Bytes to Packets and Packets to Bytes Converter cores in SOPC Builder to add the core to a system. There are no user-configurable parameters for this core.

Device Support

The Avalon-ST Bytes to Packets and Packets to Bytes Converter cores support all Altera device families.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 18–2 shows the revision history for this chapter.

Table 18–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

 © March 2009 Altera Corporation

19. Avalon Packets to Transactions Converter Core

QII55013-9.0.0

Core Overview

The Avalon® Packets to Transactions Converter core receives streaming data from upstream components and initiates Avalon Memory-Mapped (Avalon-MM) transactions. The core then returns Avalon-MM transaction responses to the requesting components.

The SPI Slave to Avalon Master Bridge and JTAG to Avalon Master Bridge are examples of how this core is used. For more information on the bridge, refer to the *SPI Slave/JTAG to Avalon Master Bridge Cores* chapter in volume 5 of the *Quartus II Handbook*.

The Avalon Packets to Transactions Converter core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated systems.

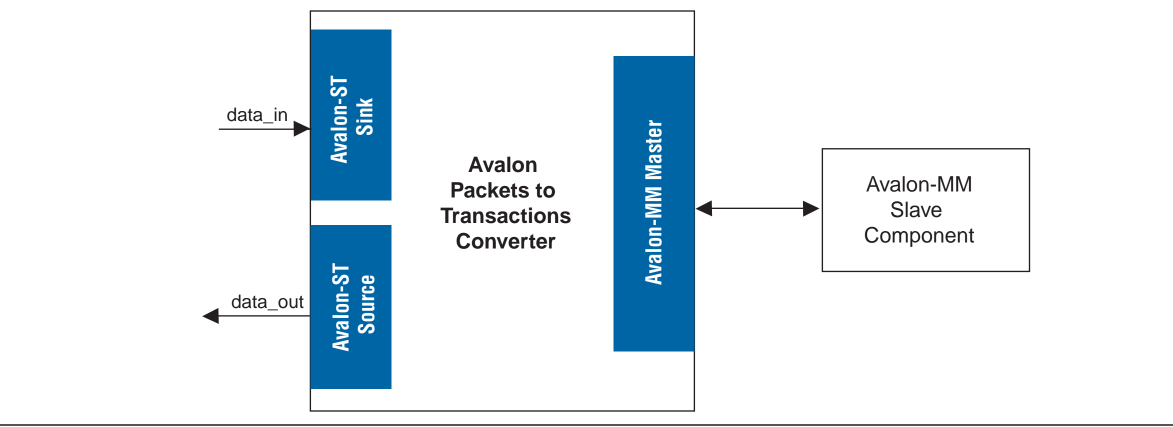
This chapter contains the following sections:

- "Functional Description"
- "Instantiating the Core in SOPC Builder" on page 19–4
- "Device Support" on page 19–4

Functional Description

Figure 19–1 shows a block diagram of the Avalon Packets to Transactions Converter core.

Figure 19–1. Avalon Packets to Transactions Converter Core



Interfaces

Table 19–1 shows the properties of the Avalon-ST interfaces.

Table 19–1. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Data width = 8 bits; Bits per symbol = 8.
Channel	Not supported.
Error	Not used.
Packet	Supported.

The Avalon-MM master interface supports read and write transactions. The data width is set to 32 bits and burst transactions are not supported.

For more information about Avalon-ST interfaces, refer to *Avalon Interface Specifications*.

Operation

The Avalon Packets to Transactions Converter core receives streams of packets on its Avalon-ST sink interface and initiates Avalon-MM transactions. Upon receiving transaction responses from Avalon-MM slaves, the core transforms the responses to packets and returns them to the requesting components via its Avalon-ST source interface. The core does not report Avalon-ST errors.

Packet Formats

The core expects incoming data streams to be in the format shown in Table 19–2. A response packet is returned for every write transaction. The core also returns a response packet if a no transaction (0x7f) is received. An invalid transaction code is regarded as a no transaction. For read transactions, the core simply returns the data read.

Table 19–2. Packet Formats

Byte	Field	Description
Transaction Packet Format		
0	`Transaction code`	Type of transaction. See Table 19–1.
1	Reserved	Reserved for future use.
[3:2]	`Size`	Transaction size in bytes. For write transactions, the size indicates the size of the `data` field. For read transactions, the size indicates the total number of bytes to read.
[7:4]	`Address`	32-bit address for the transaction.
[n:8]	`Data`	Transaction data; data to be written for write transactions.
Response Packet Format		
0	`Transaction code`	The transaction code with the most significant bit inversed.
1	Reserved	Reserved for future use.
[4:2]	`Size`	Total number of bytes read/written successfully.

Supported Transactions

Table 19–3 lists the Avalon-MM transactions supported by the core.

Table 19–3. Transaction Supported

Transaction Code	Avalon-MM Transaction	Description
`0x00`	Write, non-incrementing address.	Writes data to the given address until the total number of bytes written to the same word address equals to the value specified in the `size` field.
`0x04`	Write, incrementing address.	Writes transaction data starting at the given address.
`0x10`	Read, non-incrementing address.	Reads 32 bits of data from the given address until the total number of bytes read from the same address equals to the value specified in the `size` field.
`0x14`	Read, incrementing address.	Reads the number of bytes specified in the `size` field starting from the given address.
`0x7f`	No transaction.	No transaction is initiated. You can use this transaction type for testing purposes. Although no transaction is initiated on the Avalon-MM interface, the core still returns a response packet for this transaction code.

The core can handle only a single transaction at a time. The `ready` signal on the core's Avalon-ST sink interface is asserted only when the current transaction is completely processed.

No internal buffer is implemented on the data paths. Data received on the Avalon-ST interface is forwarded directly to the Avalon-MM interface and vice-versa. Asserting the `waitrequest` signal on the Avalon-MM interface backpressures the Avalon-ST sink interface. In the opposite direction, if the Avalon-ST source interface is backpressured, the `read` signal on the Avalon-MM interface is not asserted until the backpressure is alleviated. Backpressuring the Avalon-ST source in the middle of a read could result in data loss. In such cases, the core returns the data that is successfully received.

A transaction is considered complete when the core receives an EOP. For write transactions, the actual data size is expected to be the same as the value of the `size` field. Whether or not both values agree, the core always uses the EOP to determine the end of data.

Malformed Packets

The following are examples of malformed packets:

- Consecutive start of packet (SOP)—An SOP marks the beginning of a transaction. If an SOP is received in the middle of a transaction, the core drops the current transaction without returning a response packet for the transaction, and initiates a new transaction. This effectively handles packets without an end of packet(EOP).
- Unsupported transaction codes—The core treats unsupported transactions as a no transaction.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the Avalon Packets to Transactions Converter core in SOPC Builder to add the core to a system. There are no user-configurable settings for this core.

Device Support

The Avalon Packets to Transactions Converter core supports all Altera device families.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 19–4 shows the revision history for this chapter.

Table 19–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

20. Avalon-ST Round Robin Scheduler Core

QII55016-9.0.0

Core Overview

Avalon® Streaming (Avalon-ST) components in SOPC Builder provide a channel interface to stream data from multiple channels into a single component. In a multi-channel Avalon-ST component that stores data, the component can store data either in the sequence that it comes in (FIFO) or in segments according to the channel. When data is stored in segments according to channels, a scheduler is needed to schedule the read operations from that particular component. The most basic of the schedulers is the Avalon-ST Round Robin Scheduler core.

The Avalon-ST Round Robin Scheduler core is SOPC Builder-ready and can integrate easily into any SOPC Builder-generated systems.

This chapter contains the following sections:

- "Performance and Resource Utilization"
- "Functional Description" on page 20–2
- "Instantiating the Core in SOPC Builder" on page 20–4
- "Device Support" on page 20–4

Performance and Resource Utilization

This section lists the resource utilization and performance data for various Altera® device families. The estimates are obtained by compiling the core using the Quartus® II software.

Table 20–1 shows the resource utilization and performance data for a Stratix® II GX device (EP2SGX130GF1508I4).

Table 20–1. Resource Utilization and Performance Data for Stratix II GX Devices

Number of Channels	ALUTs	Logic Registers	Memory M512/M4K/ M-RAM	f_{MAX} (MHz)
4	7	7	0/0/0	> 125
12	25	17	0/0/0	> 125
24	62	30	0/0/0	> 125

Table 20–2 shows the resource utilization and performance data for a Stratix III device (EP3SL340F1760C3). The performance of the MegaCore® function in Stratix IV devices is similar to Stratix III devices.

Table 20–2. Resource Utilization and Performance Data for Stratix III Devices

Number of Channels	ALUTs	Logic Registers	Memory M9K/M144K/ MLAB	f_{MAX} (MHz)
4	7	7	0/0/0	> 125
12	25	17	0/0/0	> 125
24	67	30	0/0/0	> 125

Table 20–3 shows the resource utilization and performance data for a Cyclone® III device (EP3C120F780I7).

Table 20–3. Resource Utilization and Performance Data for Cyclone III Devices

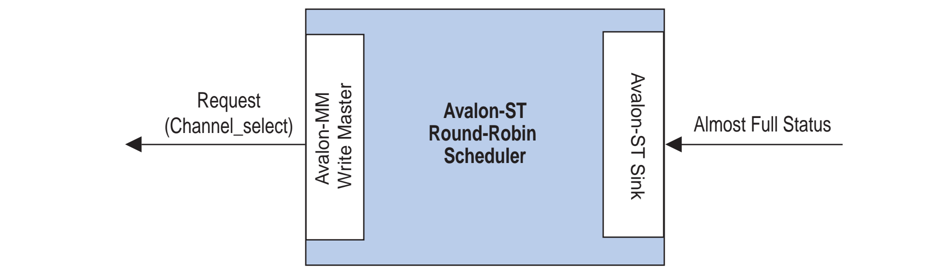
Number of Channels	Total Logic Elements	Total Registers	Memory M9K	f_{MAX} (MHz)
4	12	7	0	> 125
12	32	17	0	> 125
24	71	30	0	> 125

Functional Description

The Avalon-ST Round Robin Scheduler core controls the read operations from a multi-channel Avalon-ST component that buffers data by channels. It reads the almost-full threshold values from the multiple channels in the multi-channel component and issues the read request to the Avalon-ST source according to a round-robin scheduling algorithm.

Figure 20–1 shows the block diagram of the Avalon-ST Round Robin Scheduler.

Figure 20–1. Avalon-ST Round Robin Scheduler Block Diagram



Interfaces

The following interfaces are available in the Avalon-ST Round Robin Scheduler core:

- Almost-Full Status Interface
- Request Interface

Almost-Full Status Interface

The Almost-Full Status interface is an Avalon-ST sink interface. Table 20–4 describes the almost-full interface.

Table 20–4. Avalon-ST Interface Feature Support

Feature	Property
Backpressure	Not supported
Data Width	Data width = 1; Bits per symbol = 1
Channel	Maximum channel = 32; Channel width = 5
Error	Not supported
Packet	Not supported

The interface collects the almost-full status from the sink components for all the channels in the sequence provided.

Request Interface

The Request Interface is an Avalon Memory-Mapped (MM) Write Master interface. This interface requests data from a specific channel. The Avalon-ST Round Robin Scheduler core cycles through all of the channels it supports and schedules data to be read.

Operations

If a particular channel is almost full, the Avalon-ST Round Robin Scheduler will not schedule data to be read from that channel in the source component.

The Avalon-ST Round Robin Scheduler only requests 1 beat of data from a channel at each transaction. To request 1 beat of data from channel *n*, the scheduler writes the value `1` to address ($4 \times n$). For example, if the scheduler is requesting data from channel 3, the scheduler writes `1` to address `0xC`.

At every clock cycle, the Avalon-ST Round Robin Scheduler requests data from the next channel. Therefore, if the Avalon-ST Round Robin Scheduler starts requesting from channel 1, at the next clock cycle, it requests from channel 2. The Avalon-ST Round Robin Scheduler does not request data from a particular channel if the almost-full status for the channel is asserted. In this case, one clock cycle is used without a request transaction.

The Avalon-ST Round Robin Scheduler cannot determine if the requested component is able to service the request transaction. The component asserts `waitrequest` when it cannot accept new requests.

Table 20–5 shows the list of ports for the Avalon-ST Round Robin Scheduler core:

Table 20–5. Ports for the Avalon-ST Round Robin Scheduler (Part 1 of 2)

Signal	Direction	Description
Clock and Reset		
`clk`	In	Clock reference.
`reset_n`	In	Asynchronous active low reset.
Avalon-MM Request Interface		
`request_address` ($\log_2$ `Max_Channels-1:0`)	Out	The write address used to signal the channel the request is for.
`request_write`	Out	Write enable signal.

Table 20–5. Ports for the Avalon-ST Round Robin Scheduler (Part 2 of 2)

Signal	Direction	Description
request_writedata	Out	The amount of data requested from the particular channel. This value is always fixed at 1.
request_waitrequest	In	Wait request signal, used to pause the scheduler when the slave cannot accept a new request.
Avalon-ST Almost-Full Status Interface		
almost_full_valid	In	Indicates that almost_full_channel and almost_full_data are valid.
almost_full_channel (Channel_Width-1:0)	In	Indicates the channel for the current status indication.
almost_full_data ($\log_2$ Max_Channels-1:0)	In	A 1-bit signal that is asserted high to indicate that the channel indicated by almost_full_channel is almost full.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the Avalon-ST Round Robin Scheduler core in SOPC Builder to specify the core's configuration. Table 20–6 describes the parameters that can be configured for the Avalon-ST Round Robin Scheduler component.

Table 20–6. Parameters for Avalon-ST Round Robin Scheduler Component

Parameters	Values	Description
Number of channels	2–32	Specifies the number of channels the Avalon-ST Round Robin Scheduler supports.
Use almost-full status	0–1	Specifies whether the almost-full interface is used. If the interface is not used, the core always requests data from the next channel at the next clock cycle.

Device Support

The Avalon-ST Round Robin Scheduler core supports supports all Altera device families.

Document Revision History

Table 20–7 shows the revision history for this chapter.

Table 20–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Initial release.	—

 © March 2009 Altera Corporation

Section IV. Peripherals

This section describes multiprocessor coordination peripherals provided by Altera® for SOPC Builder systems. These components provide reliable mechanisms for multiple Nios® II processors to communicate with each other, and coordinate operations.

This section includes the following chapters:

- Chapter 21, Scatter-Gather DMA Controller Core
- Chapter 22, DMA Controller Core
- Chapter 23, Video Sync Generator and Pixel Converter Cores
- Chapter 24, Interval Timer Core
- Chapter 25, System ID Core
- Chapter 26, Mutex Core
- Chapter 27, Mailbox Core

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

21. Scatter-Gather DMA Controller Core

QII55003-9.0.0

Core Overview

The Scatter-Gather Direct Memory Access (SG-DMA) controller core implements high-speed data transfer between two components. You can use the SG-DMA controller core to transfer data from:

- Memory to memory
- Data stream to memory
- Memory to data stream

The SG-DMA controller core transfers and merges non-contiguous memory to a continuous address space, and vice versa. The core reads a series of descriptors that specify the data to be transferred.

For applications requiring more than one DMA channel, multiple instantiations of the core can provide the required throughput. Each SG-DMA controller has its own series of descriptors specifying the data transfers. A single software module controls all of the DMA channels.

The SG-DMA controller core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. For the Nios® II processor, device drivers are provided in the HAL system library, allowing software to access the core using the provided driver.

Example Systems

Figure 21–1 shows a SG-DMA controller core in a block diagram for the DMA subsystem of a printed circuit board. The SG-DMA core in the FPGA reads streaming data from an internal streaming component and writes data to an external memory. A Nios II processor provides overall system control.

Figure 21–1. SG-DMA Controller Core with Streaming Peripheral and External Memory

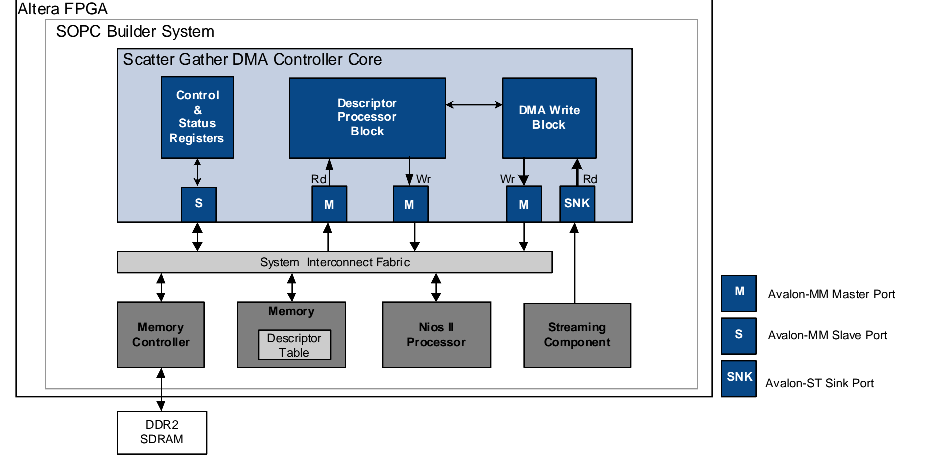


Figure 21–2 shows a different use of the SG-DMA controller core, where the core transfers data between an internal and external memory. The host processor and memory are connected to a system bus, typically either a PCI Express or Serial RapidIO™.

Figure 21–2. SG-DMA Controller Core with Internal and External Memory

Comparison of SG-DMA Controller Core and DMA Controller Core

The SG-DMA controller core provides a significant performance enhancement over the previously available DMA controller core, which could only queue one transfer at a time. Using the DMA Controller core, a CPU had to wait for the transfer to complete before writing a new descriptor to the DMA slave port. Transfers to non-contiguous memory could not be linked; consequently, the CPU overhead was substantial for small transfers, degrading overall system performance. In contrast, the SG-DMA controller core reads a series of descriptors from memory that describe the required transactions and performs all of the transfers without additional intervention from the CPU.

In This Chapter

This chapter contains the following sections:

- "Functional Description" on page 21–3
- "Device Support" on page 21–9
- "Instantiating the Core in SOPC Builder" on page 21–9
- "Simulation Considerations" on page 21–10
- "Software Programming Model" on page 21–10
- "Programming with SG-DMA Controller" on page 21–15

Resource Usage and Performance

Resource utilization for the core is 600–1400 logic elements, depending upon the width of the datapath, the parameterization of the core, the device family, and the type of data transfer. Table 21–1 provides the estimated resource usage for a SG-DMA controller core used for memory to memory transfer. The core is configurable and the resource utilization varies with the configuration specified.

Table 21–1. SG-DMA Estimated Resource Usage

Datapath	Cyclone® II	Stratix® (LEs)	Stratix II (ALUTs)
8-bit datapath	850	650	600
32-bit datapath	1100	850	700
64-bit datapath	1250	1250	800

The core operating frequency varies with the device and the size of the datapath. Table 21–2 provides an example of expected performance for SG-DMA cores instantiated in several different device families.

Table 21–2. SG-DMA Peak Performance

Device	Datapath	f_{MAX}	Throughput
Cyclone II	64 bits	150 MHz	9.6 Gbps
Cyclone III	64 bits	160 MHz	10.2 Gbps
Stratix II/Stratix II GX	64 bits	250 MHz	16.0 Gbps
Stratix III	64 bits	300 MHz	19.2 Gbps

Functional Description

The SG-DMA controller core comprises three major blocks: descriptor processor, DMA read, and DMA write. These blocks are combined to create three different configurations:

- Memory to memory
- Memory to stream
- Stream to memory

The type of devices you are transferring data to and from determines the configuration to implement. Examples of memory-mapped devices are PCI, PCIe and most memory devices. The Triple Speed Ethernet MAC, DSP MegaCore functions and many video IPs are examples of streaming devices. A recompilation is necessary each time you change the configuration of the SG-DMA controller core.

Functional Blocks and Configurations

The following sections describe each functional block and configuration.

Descriptor Processor

The descriptor processor reads descriptors from the descriptor list via its Avalon-MM read master port and pushes commands into the command FIFOs of the DMA read and write blocks. Each command includes the following fields to specify a transfer:

- Source address
- Destination address
- Number of bytes to transfer
- Increment read address after each transfer
- Increment write address after each transfer
- Generate start of packet (SOP) and end of packet (EOP)

After each command is processed by the DMA read or write block, a *status token* containing information about the transfer such as the number of bytes actually written is returned to the descriptor processor, where it is written to the respective fields in the descriptor.

DMA Read Block

The DMA read block is used in memory-to-memory and memory-to-stream configurations. The block performs the following operations:

- Reads commands from the input command FIFO.
- Reads a block of memory via the Avalon-MM read master port for each command.
- Pushes data into the data FIFO.

If burst transfer is enabled, an internal read FIFO with a depth of twice the maximum read burst size is instantiated. The DMA read block initiates burst reads only when the read FIFO has sufficient space to buffer the complete burst.

DMA Write Block

The DMA write block is used in memory-to-memory and stream-to-memory configurations. The block reads commands from its input command FIFO. For each command, the DMA write block reads data from its Avalon-ST sink port and writes it to the Avalon-MM master port.

If burst transfer is enabled, an internal write FIFO with a depth of twice the maximum write burst size is instantiated. Each burst write transfers a fixed amount of data equals to the write burst size, except for the last burst. In the last burst, the remaining data is transferred even if the amount of data is less than the write burst size.

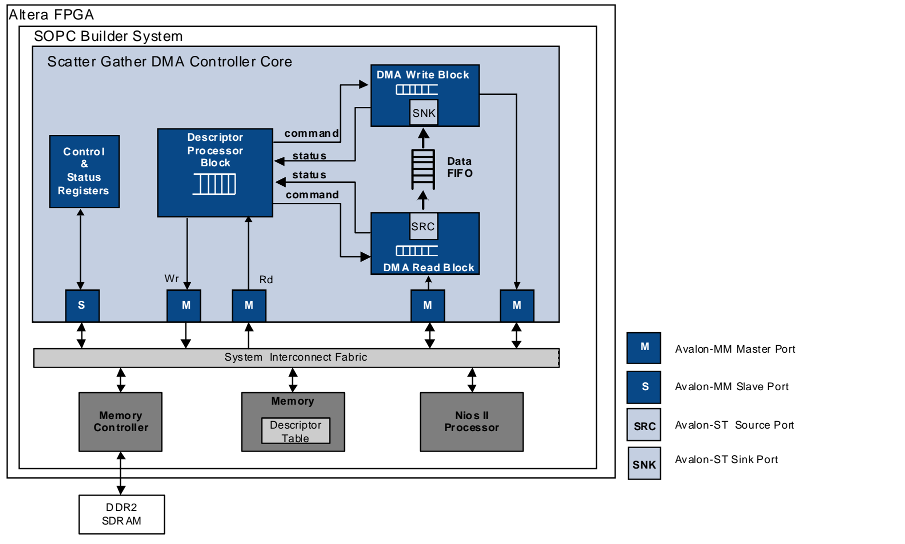
 © March 2009 Altera Corporation

Memory-to-Memory Configuration

Memory-to-memory configurations include all three blocks: descriptor processor, DMA read, and DMA write. An internal FIFO is also included to provide buffering and flow control for data transferred between the DMA read and write blocks.

Figure 21–3 illustrates one possible memory-to-memory configuration with an internal Nios II processor and descriptor list.

Figure 21–3. Example of Memory-to-Memory Configuration

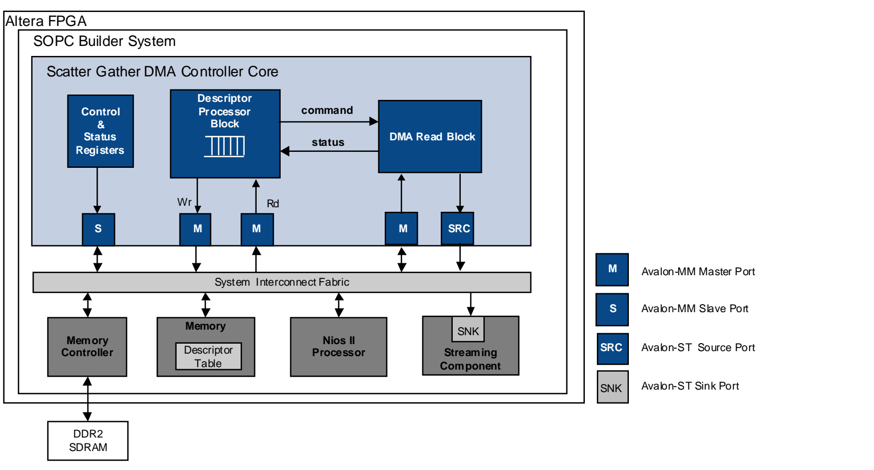


Memory-to-Stream Configuration

Memory-to-stream configurations include the descriptor processor and DMA read blocks. Figure 21–4 illustrates a memory-to-stream configuration.

In this example, the Nios II processor and descriptor table are in the FPGA. Data from an external DDR2 SDRAM is read by the SG-DMA controller and written to an on-chip streaming peripheral.

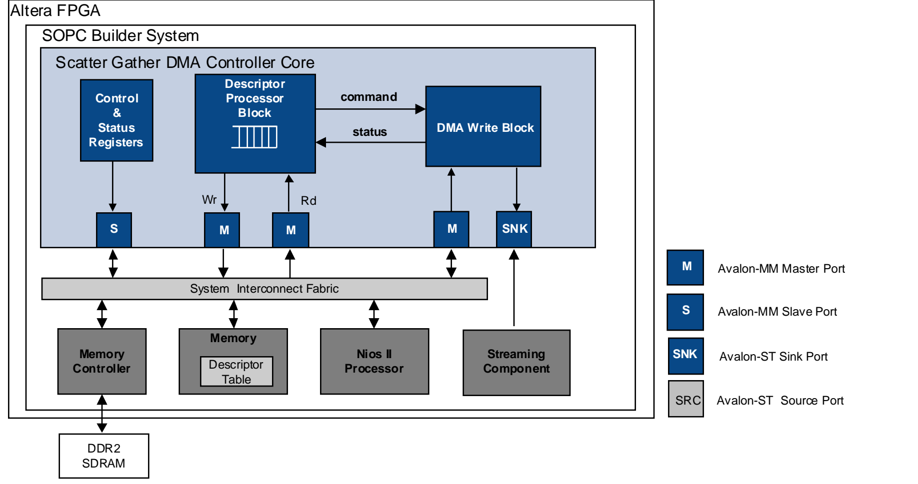
Figure 21–4. Example of Memory-to-Stream Configuration



Stream-to-Memory Configuration

Stream-to-memory configurations include the descriptor processor and DMA write blocks. This configuration is similar to the memory-to-stream configuration as Figure 21–5 illustrates.

Figure 21–5. Example of Memory-to-Stream Configuration



DMA Descriptors

DMA descriptors specify data transfers to be performed. The SG-DMA core uses a dedicated interface to read and write the descriptors. These descriptors, which are stored as a linked list, can be stored on an on-chip or off-chip memory and can be arbitrarily long.

Storing the descriptor list in an external memory frees up resources in the FPGA; however, an external descriptor list increases the overhead involved when the descriptor processor reads and updates the list. The SG-DMA core has an internal FIFO to store descriptors read from memory, which allows the core to perform descriptor read, execute, and write back operations in parallel, hiding the descriptor access and processing overhead.

The descriptors must be initialized and aligned on a 32-bit boundary. The last descriptor in the list must have its `OWNED_BY_HW` bit set to 0 because the core relies on a cleared `OWNED_BY_HW` bit to stop processing.

See "DMA Descriptors" on page 21–13 for the structure of the DMA descriptor.

Descriptor Processing

The following steps describe how the DMA descriptors are processed:

1. Software builds the descriptor linked list. See "Building and Updating Descriptor List" on page 21–8 for more information on how to build and update the descriptor linked list.
2. Software writes the address of the first descriptor to the `next_descriptor_pointer` register and initiates the transfer by setting the `RUN` bit in the `control` register to 1. See "Software Programming Model" on page 21–10 for more information on the registers.

   On the next clock cycle following the assertion of the `RUN` bit, the core sets the `BUSY` bit in the `status` register to 1 to indicate that descriptor processing is executing.
3. The descriptor processor block reads the address of the first descriptor from the `next_descriptor_pointer` register and pushes the retrieved descriptor into the command FIFO, which feeds commands to both the DMA read and write blocks. As soon as the first descriptor is read, the block reads the next descriptor and pushes it into the command FIFO. One descriptor is always read in advance thus maximizing throughput.
4. The core performs the data transfer.
   - In memory-to-memory configurations, the DMA read block receives the source address from its command FIFO and starts reading data to fill the FIFO on its stream port until the specified number of bytes are transferred. The DMA read block pauses when the FIFO is full until the FIFO has enough space to accept more data.

     The DMA write block gets the destination address from its command FIFO and starts writing until the specified number of bytes are transferred. If the data FIFO ever empties, the write block pauses until the FIFO has more data to write.
   - In memory-to-stream configurations, the DMA read block reads from the source address and transfers the data to the core's streaming port until the specified number of bytes are transferred or the end of packet is reached. The block uses the end-of-packet indicator for transfers with an unknown transfer size.
   - In stream-to-memory configurations, the DMA write block reads from the core's streaming port and writes to the destination address. The block continues reading until the specified number of bytes are transferred.
5. The descriptor processor block receives a status from the DMA read or write block and updates the `DESC_CONTROL`, `DESC_STATUS`, and `ACTUAL_BYTES_TRANSFERRED` fields in the descriptor. The `OWNED_BY_HW` bit in the `DESC_CONTROL` field is cleared unless the `PARK` bit is set to 1.

Once the core starts processing the descriptors, software must not update descriptors with `OWNED_BY_HW` bit set to 1. It is only safe for software to update a descriptor when its `OWNED_BY_HW` bit is cleared.

The SG-DMA core continues processing the descriptors until an error condition occurs and the STOP_DMA_ER bit is set to 1, or a descriptor with a cleared OWNED_BY_HW bit is encountered.

Building and Updating Descriptor List

Altera recommends the following method of building and updating the descriptor list:

1. Build the descriptor list and terminate the list with a non-hardware owned descriptor (OWNED_BY_HW = 0). The list can be arbitrarily long.
2. Set the interrupt IE_CHAIN_COMPLETED.
3. Write the address of the first descriptor in the first list to the next_descriptor_pointer register and set the RUN bit to 1 to initiate transfers.
4. While the core is processing the first list, build a second list of descriptors.
5. When the SD-DMA controller core finishes processing the first list, an interrupt is generated. Update the next_descriptor_pointer register with the address of the first descriptor in the second list. Clear the RUN bit and the status register. Set the RUN bit back to 1 to resume transfers.
6. If there are new descriptors to add, always add them to the list which the core is not processing. For example, if the core is processing the first list, add new descriptors to the second list and so forth.

This method ensures that the descriptors are not updated when the core is processing them. Because the method requires a response to the interrupt, a high-latency interrupt may cause a problem in systems where stalling data movement is not possible.

Error Conditions

The SG-DMA core has a configurable error width. Error signals are connected directly to the Avalon-ST source or sink to which the SG-DMA core is connected. Table 21–3 lists the error signals when the core is operating in the memory-to-stream configuration and connected to the transmit FIFO interface of the Altera Triple-Speed Ethernet MegaCore® function.

Table 21–3. Avalon-ST Transmit Error Types

Signal Type	Description
TSE_transmit_error[0]	Transmit Frame Error. Asserted to indicate that the transmitted frame should be viewed as invalid by the Ethernet MAC. The frame is then transferred onto the GMII interface with an error code during the frame transfer.

Table 21–4 lists the error signals when the core is operating in the stream-to-memory configuration and connected to the transmit FIFO interface of the Triple-Speed Ethernet MegaCore function.

Table 21–4. Avalon-ST Receive Error Types

Signal Type	Description
TSE_receive_error[0]	Receive Frame Error. This signal indicates that an error has occurred. It is the logical OR of receive errors 1 through 5.
TSE_receive_error[1]	Invalid Length Error. Asserted when the received frame has an invalid length as defined by the IEEE 802.3 standard.
TSE_receive_error[2]	CRC Error. Asserted when the frame has been received with a CRC-32 error.
TSE_receive_error[3]	Receive Frame Truncated. Asserted when the received frame has been truncated due to receive FIFO overflow.
TSE_receive_error[4]	Received Frame corrupted due to PHY error. (The PHY has asserted an error on the receive GMII interface.)
TSE_receive_error[5]	Collision Error. Asserted when the frame was received with a collision.

Each streaming core has a different set of error codes. Refer to the respective user guides for the codes.

Device Support

The SG-DMA Controller core supports all Altera device families.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the SG-DMA Controller core in SOPC Builder to add the core to a system.

The SG-DMA controller core should be given a higher priority (lower IRQ value) than most of the components in a system to ensure high througput.

Table 21–5 lists and describes the parameters you can configure.

Table 21–5. Configurable Parameters (Part 1 of 2)

Parameter	Legal Values	Description
Transfer mode	Memory To Memory Memory To Stream Stream To Memory	Configuration to use. For more information about these configurations, see "Memory-to-Memory Configuration" on page 21–5
Enable bursting on descriptor read master	On/Off	If this option is on, the descriptor processor block uses Avalon-MM bursting when fetching descriptors and writing them back in memory. With 32-bit read and write ports, the descriptor processor block can fetch the 256-bit descriptor by performing 8-word burst as opposed to eight individual single-word transactions.
Allow unaligned transfers	On/Off	If this option is on, the core allows accesses to non-word-aligned addresses. This option doesn't apply for burst transfers. Unaligned transfers require extra logic that may negatively impact system performance.
Enable burst transfers	On/Off	Turning on this option enables burst reads and writes.

Table 21–5. Configurable Parameters (Part 2 of 2)

Parameter	Legal Values	Description
Read burstcount signal width	1–16	The width of the read `burstcount` signal. This value determines the maximum burst read size.
Write burstcount signal width	1–16	The width of the write `burstcount` signal. This value determines the maximum burst write size.
Data width	8, 16, 32, 64	The data width in bits for the Avalon-MM read and write ports.
Source error width	0–7	The width of the `error` signal for the Avalon-ST source port.
Sink error width	0 – 7	The width of the `error` signal for the Avalon-ST sink port.
Data transfer FIFO depth	2, 4, 8, 16, 32, 64	The depth of the internal data FIFO in memory-to-memory configurations with burst transfers disabled.

Simulation Considerations

Signals for hardware simulation are automatically generated as part of the Nios II simulation process available in the Nios II IDE.

Software Programming Model

The following sections describe the software programming model for the SG-DMA controller core.

HAL System Library Support

The Altera-provided driver implements a HAL device driver that integrates into the HAL system library for Nios II systems. HAL users should access the SG-DMA controller core via the familiar HAL API and the ANSI C standard library.

Software Files

The SG-DMA controller core provides the following software files. These files provide low-level access to the hardware and drivers that integrate into the Nios II HAL system library. Application developers should not modify these files.

- **altera_avalon_sgdma_regs.h**—defines the core's register map, providing symbolic constants to access the low-level hardware
- **altera_avalon_sgdma.h**—provides definitions for the Altera Avalon SG-DMA buffer control and status flags.
- **altera_avalon_sgdma.c**—provides function definitions for the code that implements the SG-DMA controller core.
- **altera_avalon_sgdma_descriptor.h**—defines the core's descriptor, providing symbolic constants to access the low-level hardware.

Register Maps

The SG-DMA controller core has three registers accessible from its Avalon-MM interface; `status`, `control` and `next_descriptor_pointer`. Software can configure the core and determines its current status by accessing the registers.

The control/status register has a 32-bit interface without byte-enable logic, and therefore cannot be properly accessed by a master with narrower data width than itself. To ensure correct operation of the core, always access the register with a master that is at least 32 bits wide.

Table 21–6 lists and describes the registers.

Table 21–6. Register Map

32-bit Word Offset	Register Name	Reset Value	Description
base + 0	status	0	This register indicates the core's current status such as what caused the last interrupt and if the core is still processing descriptors. See Table 21–4 on page 21–9 for the status register map.
base + 4	control	0	This register specifies the core's behavior such as what triggers an interrupt and when the core is started and stopped. The host processor can configure the core by setting the register bits accordingly. See Table 21–4 on page 21–9 for the control register map.
base + 8	next_descriptor_pointer	0	This register contains the address of the next descriptor to process. Set this register to the address of the first descriptor as part of the system initialization sequence. Altera recommends that user applications clear the RUN bit in the control register and wait until the BUSY bit of the status register is set to 0 before reading this register.

Table 21–7 provides a bit map for the control register.

Table 21–7. Control Register Bit Map (Part 1 of 2)

Bit	Bit Name	Access	Description
0	IE_ERROR	R/W	When this bit is set to 1, the core generates an interrupt if an Avalon-ST error occurs during descriptor processing. *(1)*
1	IE_EOP_ENCOUNTERED	R/W	When this bit is set to 1, the core generates an interrupt if an EOP is encountered during descriptor processing. *(1)*
2	IE_DESCRIPTOR_COMPLETED	R/W	When this bit is set to 1, the core generates an interrupt after each descriptor is processed. *(1)*
3	IE_CHAIN_COMPLETED	R/W	When this bit is set to 1, the core generates an interrupt after the last descriptor in the list is processed, that is when the core encounters a descriptor with a cleared OWNED_BY_HW bit. *(1)*
4	IE_GLOBAL	R/W	Global signal to enable all interrupts.
5	RUN	R/W	Set this bit to 1 to start the descriptor processor block which subsequently initiates DMA transactions. Prior to setting this bit to 1, ensure that the register next_descriptor_pointer is updated with the address of the first descriptor to process. The core continues to process descriptors in its queue as long as this bit is 1. Clear this bit to stop the core from processing the next descriptor in its queue. If this bit is cleared in the middle of processing a descriptor, the core completes the processing before stopping. The host processor can then modify the remaining descriptors and restart the core.

Table 21–7. Control Register Bit Map (Part 2 of 2)

Bit	Bit Name	Access	Description
6	STOP_DMA_ER	R/W	Set this bit to 1 to stop the core when an Avalon-ST error is encountered during a DMA transaction. This bit applies only to stream-to-memory configurations.
7	IE_MAX_DESC_PROCESSED	R/W	Set this bit to 1 to generate an interrupt after the number of descriptors specified by MAX_DESC_PROCESSED are processed.
8 .. 15	MAX_DESC_PROCESSED	R/W	Specifies the number of descriptors to process before the core generates an interrupt.
16	SW_RESET	R/W	Software can reset the core by writing to this bit twice. Upon the second write, the core is reset. The logic which sequences the software reset process then resets itself automatically. Executing a software reset when a DMA transfer is active may result in permanent bus lockup until the next system reset. Hence, Altera recommends that you use the software reset as your last resort.
17	PARK	R/W	Seting this bit to 0 causes the SG-DMA controller core to clear the OWNED_BY_HW bit in the descriptor after each descriptor is processed. If the PARK bit is set to 1, the core does not clear the OWNED_BY_HW bit, thus allowing the same descriptor to be processed repeatedly without software intervention. You also need to set the last descriptor in the list to point to the first one.
18..30	Reserved		
31	CLEAR_INTERRUPT	R/W	Set this bit to 1 to clear pending interrupts.

Note to Table 21–11:

(1) All interrupts are generated only after the descriptor is updated.

Table 21–8 provides a bit map for the status register. Altera recommends that you read the status register only after the RUN bit in the control register is cleared.

Table 21–8. Status Register Bit Map (Part 1 of 2)

Bit	Bit Name	Access	Description
0	ERROR	R/C *(1) (2)*	A value of 1 indicates that an Avalon-ST error was encountered during a transfer.
1	EOP_ENCOUNTERED	R/C	A value of 1 indicates that the transfer was terminated by an end-of-packet (EOP) signal generated on the Avalon-ST source interface. This condition is only possible in stream-to-memory configurations.
2	DESCRIPTOR_COMPLETED	R/C *(1) (2)*	A value of 1 indicates that a descriptor was processed to completion.
3	CHAIN_COMPLETED	R/C *(1) (2)*	A value of 1 indicates that the core has completed processing the descriptor chain.

Table 21–8. Status Register Bit Map (Part 2 of 2)

Bit	Bit Name	Access	Description
4	BUSY	R *(1) (3)*	A value of 1 indicates that descriptors are being processed. This bit is set to 1 on the next clock cycle after the RUN bit is asserted and does not get cleared until one of the following event occurs: ■ Descriptor processing completes and the RUN bit is cleared. ■ An error condition occurs, the STOP_DMA_ER bit is set to 1 and the processing of the current descriptor completes.
5 .. 31	Reserved		

Notes to Table 21–8:

(1) This bit must be cleared after a read is performed. Write one to clear this bit.

(2) This bit is updated by hardware after each DMA transfer completes. It remains set until software writes one to clear.

(3) This bit is continuously updated by the hardware.

DMA Descriptors

Table 21–9 shows the structure a DMA descriptor entry. See “Data Structure” on page 21–15 for the structure definition.

Table 21–9. DMA Descriptor Structure

Byte Offset	Field Names			
	31 24	23 16	15 8	7 0
base	source			
base + 4	Reserved			
base + 8	destination			
base + 12	Reserved			
base + 16	next_desc_ptr			
base + 20	Reserved			
base + 24	Reserved		bytes_to_transfer	
base + 28	desc_control	desc_status	actual_bytes_transferred	

Table 21–10 describes the each field in a descriptor entry.

Table 21–10. DMA Descriptor Field Description (Part 1 of 2)

Field Name	Access	Description
source	R/W	Specifies the address of data to be read. This address is set to 0 if the input interface is an Avalon-ST interface.
destination	R/W	Specifies the address to which data should be written. This address is set to 0 if the output interface is an Avalon-ST interface.
next_desc_ptr	R/W	Specifies the address of the next descriptor in the linked list.
bytes_to_transfer	R/W	Specifies the number of bytes to transfer. If this field is 0, the SG-DMA controller core continues transferring data until it encounters an EOP.

Table 21–10. DMA Descriptor Field Description (Part 2 of 2)

Field Name	Access	Description
actual_bytes_transferred	R	Specifies the number of bytes that are successfully transferred by the core. This field is updated after the core processes a descriptor.
desc_status	R/W	This field is updated after the core processes a descriptor. See Table 21–12 on page 21–14 for the bit map of this field.
desc_control	R/W	Specifies the behavior of the core. This field is updated after the core processes a descriptor. See Table 21–11 on page 21–14 for descriptions of each bit.

Table 21–1 provides a bit map for the desc_control field.

Table 21–11. DESC_CONTROL Bit Map

Bit (s)	Field Name	Access	Description
0	GENERATE_EOP	W	When this bit is set to 1,the DMA read block asserts the EOP signal on the final word.
1	READ_FIXED_ADDRESS	R/W	This bit applies only to Avalon-MM read master ports. When this bit is set to 1, the DMA read block does not increment the memory address. When this bit is set to 0, the read address increments after each read.
2	WRITE_FIXED_ADDRESS	R/W	This bit applies only to Avalon-MM write master ports. When this bit is set to 1, the DMA write block does not increment the memory address. When this bit is set to 0, the write address increments after each write. In memory-to-stream configurations, the DMA read block generates a start-of-packet (SOP) on the first word when this bit is set to 1.
[6:3]	Reserved	—	—
7	OWNED_BY_HW	R/W	This bit determines whether hardware or software has write access to the current register. When this bit is set to 1, the core can update the descriptor and software should not access the descriptor due to the possibility of race conditions. Otherwise, it is safe for software to update the descriptor.

After completing a DMA transaction, the descriptor processor block updates the desc_status field to indicate how the transaction proceeded. Table 21–1 provides the bit map of this field.

Table 21–12. DESC_STATUS Bit Map

Bit	Bit Name	Access	Description
[7:0]	ERROR_0 .. ERROR_7	R	Each bit represents an error that occurred on the Avalon-ST interface. The context of each error is defined by the component connected to the Avalon-ST interface.

Timeouts

The SG-DMA controller does not implement internal counters to detect stalls. Software can instantiate a timer component if this functionality is required.

Programming with SG-DMA Controller

This section describes the device and descriptor data structures, and the application programming interface (API) for the SG-DMA controller core.

Data Structure

Figure 21–6 shows the data structure for the device.

Figure 21–6. Device Data Structure

```
typedef struct alt_sgdma_dev
{
  alt_llist                 llist;               // Device linked-list entry
  const char                *name;               // Name of SGDMA in SOPC System
  void                      *base;               // Base address of SGDMA
  alt_u32                   *descriptor_base;    // reserved
  alt_u32                   next_index;          // reserved
  alt_u32                   num_descriptors;     // reserved
  alt_sgdma_descriptor      *current_descriptor; // reserved
  alt_sgdma_descriptor      *next_descriptor;    // reserved
  alt_avalon_sgdma_callback callback;            // Callback routine pointer
  void                      *callback_context;   // Callback context pointer
  alt_u32                   chain_control;       // Value OR'd into control reg
} alt_sgdma_dev;
```

Figure 21–7 shows the data structure for the descriptors.

Figure 21–7. Descriptor Data Structure

```
typedef struct {
    alt_u32   *read_addr;
    alt_u32   read_addr_pad;

    alt_u32   *write_addr;
    alt_u32   write_addr_pad;

    alt_u32   *next;
    alt_u32   next_pad;

    alt_u16   bytes_to_transfer;
    alt_u8    read_burst; /* Reserved field. Set to 0. */
    alt_u8    write_burst;/* Reserved field. Set to 0. */

    alt_u16   actual_bytes_transferred;
    alt_u8    status;
    alt_u8    control;

} alt_avalon_sgdma_packed alt_sgdma_descriptor;
```

SG-DMA API

Table 21–13 lists all functions provided and briefly describes each.

Table 21–13. Function List

Name	Description
alt_avalon_sgdma_do_async_transfer()	Starts a non-blocking transfer of a descriptor chain.
alt_avalon_sgdma_do_sync_transfer()	Starts a blocking transfer of a descriptor chain. This function blocks both before transfer if the controller is busy and until the requested transfer has completed.
alt_avalon_sgdma_construct_mem_to_mem_desc()	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-MM to Avalon-MM transfer.
alt_avalon_sgdma_construct_stream_to_mem_desc()	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-ST to Avalon-MM transfer. The function automatically terminates the descriptor chain with a NULL descriptor.
alt_avalon_sgdma_construct_mem_to_stream_desc()	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-MM to Avalon-ST transfer.
alt_avalon_sgdma_check_descriptor_status()	Reads the status of a given descriptor.
alt_avalon_sgdma_register_callback()	Associates a user-specific callback routine with the SG-DMA interrupt handler.
alt_avalon_sgdma_start()	Starts the DMA engine. This is not required when alt_avalon_sgdma_do_async_transfer() and alt_avalon_sgdma_do_sync_transfer() are used.
alt_avalon_sgdma_stop()	Stops the DMA engine. This is not required when alt_avalon_sgdma_do_async_transfer() and alt_avalon_sgdma_do_sync_transfer() are used.
alt_avalon_sgdma_open()	Returns a pointer to the SG-DMA controller with the given name.

alt_avalon_sgdma_do_async_transfer()

Prototype: int alt_avalon_do_async_transfer(alt_sgdma_dev *dev, alt_sgdma_descriptor *desc)

Thread-safe: No.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*dev`—a pointer to an SG-DMA device structure.

`*desc`—a pointer to a single, constructed descriptor. The descriptor must have its “next” descriptor field initialized either to a non-ready descriptor, or to the next descriptor in the chain.

Returns: Returns 0 success. Other return codes are defined in **errno.h**.

Description: Set up and begin a non-blocking transfer of one or more descriptors or a descriptor chain. If the SG-DMA controller is busy at the time of this call, the routine immediately returns `EBUSY`; the application can then decide how to proceed without being blocked. If a callback routine has been previously registered with this particular SG-DMA controller, the transfer is set up to issue an interrupt on error, EOP, or chain completion. Otherwise, no interrupt is registered and the application developer must check for and handle errors and completion.

alt_avalon_sgdma_do_sync_transfer()

Prototype: alt_u8 alt_avalon_sgdma_do_sync_transfer(alt_sgdma_dev *dev, alt_sgdma_descriptor *desc)

Thread-safe: No.

Available from ISR: Not recommended.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*dev`—a pointer to an SG-DMA device structure.

`*desc`—a pointer to a single, constructed descriptor. The descriptor must have its “next” descriptor field initialized either to a non-ready descriptor, or to the next descriptor in the chain.

Returns: Returns the contents of the `status` register.

Description: Sends a fully formed descriptor or list of descriptors to the SG-DMA controller for transfer. This function blocks both before transfer, if the SG-DMA controller is busy, and until the requested transfer has completed. If an error is detected during the transfer, it is abandoned and the controller’s `status` register contents are returned to the caller. Additional error information is available in the status bits of each descriptor that the SG-DMA processed. The user application searches through the descriptor or list of descriptors to gather specific error information.

alt_avalon_sgdma_construct_mem_to_mem_desc()

Prototype:	void alt_avalon_sgdma_construct_mem_to_mem_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *read_addr, alt_u32 *write_addr, alt_u16 length, int read_fixed, int write_fixed)
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>
Parameters:	`*desc`—a pointer to the descriptor being constructed. `*next`—a pointer to the "next" descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated. `*read_addr`—the first read address for the SG-DMA transfer. `*write_addr`—the first write address for the SG-DMA transfer. `length`—the number of bytes for the transfer. `read_fixed`—if non-zero, the SG-DMA reads from a fixed address. `write_fixed`—if non-zero, the SG-DMA writes to a fixed address.
Returns:	void
Description:	This function constructs a single SG-DMA descriptor in the memory specified in `alt_avalon_sgdma_descriptor *desc` for an Avalon-MM to Avalon-MM transfer. The function sets the `OWNED_BY_HW` bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the `RUN` bit is 1. The next field of the descriptor being constructed is set to the address in `*next`. The `OWNED_BY_HW` bit of the descriptor at `*next` is explicitly cleared. Once the SG-DMA completes processing of the `*desc`, it does not process the descriptor at `*next` until its `OWNED_BY_HW` bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's `*next` pointer in the *desc parameter. You must properly allocate memory for the creation of both the descriptor under construction as well as the next descriptor in the chain. Descriptors must be in a memory device mastered by the SG-DMA controller's chain read and chain write Avalon master ports. Care must be taken to ensure that both `*desc` and `*next` point to areas of memory mastered by the controller.

alt_avalon_sgdma_construct_stream_to_mem_desc()

Prototype:	void alt_avalon_sgdma_construct_stream_to_mem_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *write_addr, alt_u16 length_or_eop, int write_fixed)
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>
Parameters:	`*desc`—a pointer to the descriptor being constructed. `*next`—a pointer to the “next” descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated. `*write_addr`—the first write address for the SG-DMA transfer. `length_or_eop`—the number of bytes for the transfer. If set to zero (0x0), the transfer continues until an EOP signal is received from the Avalon-ST interface. `write_fixed`—if non-zero, the SG-DMA will write to a fixed address.
Returns:	void
Description:	This function constructs a single SG-DMA descriptor in the memory specified in `alt_avalon_sgdma_descriptor *desc` for an Avalon-ST to Avalon-MM transfer. The source (read) data for the transfer comes from the Avalon-ST interface connected to the SG-DMA controller's streaming read port. The function sets the `OWNED_BY_HW` bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the `RUN` bit is 1. The next field of the descriptor being constructed is set to the address in `*next`. The `OWNED_BY_HW` bit of the descriptor at `*next` is explicitly cleared. Once the SG-DMA completes processing of the `*desc`, it does not process the descriptor at `*next` until its `OWNED_BY_HW` bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's `*next` pointer in the *desc parameter. You must properly allocate memory for the creation of both the descriptor under construction as well as the next descriptor in the chain. Descriptors must be in a memory device mastered by the SG-DMA controller’s chain read and chain write Avalon master ports. Care must be taken to ensure that both `*desc` and `*next` point to areas of memory mastered by the controller.

alt_avalon_sgdma_construct_mem_to_stream_desc()

Prototype: void alt_avalon_sgdma_construct_mem_to_stream_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *read_addr, alt_u16 length, int read_fixed, int generate_sop, int generate_eop, alt_u8 atlantic_channel)

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*desc`—a pointer to the descriptor being constructed.

`*next`—a pointer to the “next” descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated.

`*read_addr`—the first read address for the SG-DMA transfer.

`length`—the number of bytes for the transfer.

`read_fixed`—if non-zero, the SG-DMA reads from a fixed address.

`generate_sop`—if non-zero, the SG-DMA generates a SOP on the Avalon-ST interface when commencing the transfer.

`generate_eop`—if non-zero, the SG-DMA generates an EOP on the Avalon-ST interface when completing the transfer.

`atlantic_channel`—an 8-bit Avalon-ST channel number. Channels are currently not supported. Set this parameter to 0.

Returns: void

Description: This function constructs a single SG-DMA descriptor in the memory specified in `alt_avalon_sgdma-descriptor *desc` for an Avalon-MM to Avalon-ST transfer. The destination (write) data for the transfer goes to the Avalon-ST interface connected to the SG-DMA controller's streaming write port. The function sets the `OWNED_BY_HW` bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the `RUN` bit is 1.

The next field of the descriptor being constructed is set to the address in `*next`. The `OWNED_BY_HW` bit of the descriptor at `*next` is explicitly cleared. Once the SG-DMA completes processing of the `*desc`, it does not process the descriptor at `*next` until its `OWNED_BY_HW` bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's `*next` pointer in the *desc parameter.

You are responsible for properly allocating memory for the creation of both the descriptor under construction as well as the next descriptor in the chain. Descriptors must be in a memory device mastered by the SG-DMA controller’s chain read and chain write Avalon master ports. Care must be taken to ensure that both `*desc` and `*next` point to areas of memory mastered by the controller.

alt_avalon_sgdma_check_descriptor_status()

Prototype: int alt_avalon_sgdma_check_descriptor_status(alt_sgdma_descriptor *desc)

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*desc`—a pointer to the constructed descriptor to examine.

Returns: Returns 0 if the descriptor is error-free, not owned by hardware, or a previously requested transfer completed normally. Other return codes are defined in **errno.h**.

Description: Checks a descriptor previously owned by hardware for any errors reported in a previous transfer. The routine reports: errors reported by the SG-DMA controller, the buffer in use.

alt_avalon_sgdma_register_callback()

Prototype: void alt_avalon_sgdma_register_callback(alt_sgdma_dev *dev, alt_avalon_sgdma_callback callback, alt_u16 chain_control, void *context)

Thread-safe: Yes.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*dev`—a pointer to the SG-DMA device structure.

`callback`—a pointer to the callback routine to execute at interrupt level.

`chain_control`—the SG-DMA control register contents.

`*context`—a pointer used to pass context-specific information to the ISR. `context` can point to any ISR-specific information.

Returns: void

Description: Associates a user-specific routine with the SG-DMA interrupt handler. If a callback is registered, all non-blocking transfers enables interrupts that causes the callback to be executed. The callback runs as part of the interrupt service routine, and care must be taken to follow the guidelines for acceptable interrupt service routine behavior as described in the *Nios II Software Developer's Handbook*.

To disable callbacks after registering one, call this routine with 0x0 as the callback argument.

alt_avalon_sgdma_start()

Prototype: void alt_avalon_sgdma_start(alt_sgdma_dev *dev)

Thread-safe: No.

Available from ISR: Yes.

Include: <**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>

Parameters: `*dev`—a pointer to the SG-DMA device structure.

Returns: void

Description: Starts the DMA engine and processes the descriptor pointed to in the controller's next descriptor pointer and all subsequent descriptors in the chain. It is not necessary to call this function when `do_sync` or `do_async` is used.

alt_avalon_sgdma_stop()

Prototype:	void alt_avalon_sgdma_stop(alt_sgdma_dev *dev)
Thread-safe:	No.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>
Parameters:	`*dev`—a pointer to the SG-DMA device structure.
Returns:	void
Description:	Stops the DMA engine following completion of the current buffer descriptor. It is not necessary to call this function when `do_sync` or `do_async` is used.

alt_avalon_sgdma_open()

Prototype:	alt_sgdma_dev* alt_avalon_sgdma_open(const char* name)
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs.h**>
Parameters:	`name`—the name of the SG-DMA device to open.
Returns:	A pointer to the SG-DMA device structure associated with the supplied name, or NULL if no corresponding SG-DMA device structure was found.
Description:	Retrieves a pointer to a hardware SG-DMA device structure.

Referenced Documents

This chapter references the *Nios II Software Developer's Handbook*.

Document Revision History

Table 21–14 shows the revision history for this chapter.

Table 21–14. Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	Added description of **Enable bursting on descriptor read master**.	—
November 2008 v8.1.0	■ Changed to 8-1/2 x 11 page size. ■ Added section DMA Descriptors in Functional Specifications ■ Revised descriptions of register fields and bits. ■ Reorganized sections Software Programming Model and Programming with SG-DMA Controller Core.	—
May 2008 v8.0.0	■ Added sections on burst transfers.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

22. DMA Controller Core

NII51006-9.0.0

Core Overview

The direct memory access (DMA) controller core with Avalon® interface performs bulk data transfers, reading data from a source address range and writing the data to a different address range. An Avalon Memor-Mapped (Avalon-MM) master peripheral, such as a CPU, can offload memory transfer tasks to the DMA controller. While the DMA controller performs memory transfers, the master is free to perform other tasks in parallel.

The DMA controller transfers data as efficiently as possible, reading and writing data at the maximum pace allowed by the source or destination. The DMA controller is capable of performing Avalon transfers with flow control, enabling it to automatically transfer data to or from a slow peripheral with flow control (for example, UART), at the maximum pace allowed by the peripheral.

The DMA controller is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. For the Nios® II processor, device drivers are provided in the HAL system library. See “Software Programming Model” on page 22–5 for details of HAL support.

This chapter contains the following sections:

- “Functional Description”
- “Instantiating the Core in SOPC Builder” on page 22–4
- “Device Support” on page 22–5
- “Software Programming Model” on page 22–5

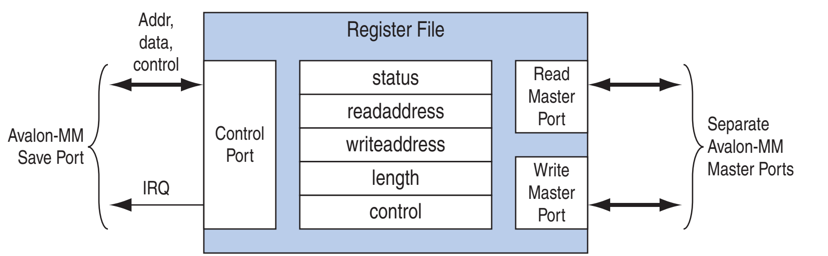
Functional Description

You can use the DMA controller to perform data transfers from a source address-space to a destination address-space. The controller has no concept of endianness and does not interpret the payload data. The concept of endianness only applies to a master that interprets payload data.

The source and destination may be either an Avalon-MM slave peripheral (for example, a constant address) or an address range in memory. The DMA controller can be used in conjunction with peripherals with flow control, which allows data transactions of fixed or variable length. The DMA controller can signal an interrupt request (IRQ) when a DMA transaction completes. A transaction is a sequence of one or more Avalon transfers initiated by the DMA controller core.

The DMA controller has two Avalon-MM master ports—a master read port and a master write port—and one Avalon-MM slave port for controlling the DMA as shown in Figure 22–1.

Figure 22–1. DMA Controller Block Diagram



A typical DMA transaction proceeds as follows:

1. A CPU prepares the DMA controller for a transaction by writing to the control port.
2. The CPU enables the DMA controller. The DMA controller then begins transferring data without additional intervention from the CPU. The DMA's master read port reads data from the read address, which may be a memory or a peripheral. The master write port writes the data to the destination address, which can also be a memory or peripheral. A shallow FIFO buffers data between the read and write ports.
3. The DMA transaction ends when a specified number of bytes are transferred (a fixed-length transaction) or an end-of-packet signal is asserted by either the sender or receiver (a variable-length transaction). At the end of the transaction, the DMA controller generates an interrupt request (IRQ) if it was configured by the CPU to do so.
4. During or after the transaction, the CPU can determine if a transaction is in progress, or if the transaction ended (and how) by examining the DMA controller's `status` register.

Setting Up DMA Transactions

An Avalon-MM master peripheral sets up and initiates DMA transactions by writing to registers via the control port. The Avalon-MM master programs the DMA engine using byte addresses which are byte aligned. The master peripheral configures the following options:

- Read (source) address location
- Write (destination) address location
- Size of the individual transfers: Byte (8-bit), halfword (16-bit), word (32-bit), doubleword (64-bit) or quadword (128-bit)
- Enable interrupt upon end of transaction
- Enable source or destination to end the DMA transaction with end-of-packet signal
- Specify whether source and destination are memory or peripheral

The master peripheral then sets a bit in the `control` register to initiate the DMA transaction.

The Master Read and Write Ports

The DMA controller reads data from the source address through the master read port, and then writes to the destination address through the master write port. You program the DMA controller using byte addresses. Read and write start addresses should be aligned to the transfer size. For example, to transfer data words, if the start address is 0, the address will increment to 4, 8, and 12. For heterogeneous systems where a number of different slave devices are of different widths, the data width for read and write masters matches the width of the widest data-width slave addressed by either the read or the write master. For bursting transfers, the burst length is set to the DMA transaction length with the appropriate unit conversion. For example, if a 32-bit data width DMA is programmed for a word transfer of 64 bytes, the length registered is programmed with 64 and the burst count port will be 16. If a 64-bit data width DMA is programmed for a doubleword transfer of 8 bytes, the length register is programmed with 8 and the burst count port will be 1.

There is a shallow FIFO buffer between the master read and write ports. The default depth is 2, which makes the write action depend on the data-available status of the FIFO, rather than on the status of the master read port.

Both the read and write master ports can perform Avalon transfers with flow control, which allows the slave peripheral to control the flow of data and terminate the DMA transaction.

For details about flow control in Avalon-MM data transfers and Avalon-MM peripherals, refer to *Avalon Interface Specifications*.

Addressing and Address Incrementing

When accessing memory, the read (or write) address increments by 1, 2, 4, 8, or 16 after each access, depending on the width of the data. On the other hand, a typical peripheral device (such as UART) has fixed register locations. In this case, the read/write address is held constant throughout the DMA transaction.

The rules for address incrementing are, in order of priority:

- If the `control` register's `RCON` (or `WCON`) bit is set, the read (or write) increment value is 0.
- Otherwise, the read and write increment values are set according to the transfer size specified in the control register, as shown in Table 22–1.

Table 22–1. Address Increment Values

Transfer Width	Increment
byte	1
halfword	2
word	4
doubleword	8
quadword	16

In systems with heterogeneous data widths, care must be taken to present the correct address or offset when configuring the DMA to access native-aligned slaves. For example, in a system using a 32-bit Nios II processor and a 16-bit DMA, the base address for the UART `txdata` register must be divided by the dma_data_width/cpu_data_width—2 in this example.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the DMA controller in SOPC Builder to specify the core's configuration. Instantiating the DMA controller in SOPC Builder creates one slave port and two master ports. You must specify which slave peripherals can be accessed by the read and write master ports. Likewise, you must specify which other master peripheral(s) can access the DMA control port and initiate DMA transactions. The DMA controller does not export any signals to the top level of the system module.

DMA Parameters (Basic)

This section describes the parameters you can configure on the **DMA Parameters** page.

Transfer Size

The parameter **Width of the DMA Length Register** specifies the minimum width of the DMA's transaction length register, which can be between 1 and 32. The `length` register determines the maximum number of transfers possible in a single DMA transaction.

By default, the length register is wide enough to span any of the slave peripherals mastered by the read or write ports. Overriding the length register may be necessary if the DMA master port (read or write) masters only data peripherals, such as a UART. In this case, the address span of each slave is small, but a larger number of transfers may be desired per DMA transaction.

Burst Transactions

When **Enable Burst Transfers** is turned on, the DMA controller performs burst transactions on its master read and write ports. The parameter **Maximum Burst Size** determines the maximum burst size allowed in a transaction.

In burst mode, the length of a transaction must not be longer than the configured maximum burst size. Otherwise, the transaction must be performed as multiple transactions.

FIFO Implementation

This option determines the implementation of the FIFO buffer between the master read and write ports. Select **Construct FIFO from Registers** to implement the FIFO using one register per storage bit. This option has a strong impact on logic utilization when the DMA controller's data width is large. See "Advanced Options" on page 22–5.

To implement the FIFO using embedded memory blocks available in the FPGA, select **Construct FIFO from Memory Blocks**.

Advanced Options

This section describes the parameters you can configure on the **Advanced Options** page.

Allowed Transactions

You can choose the transfer datawidth(s) supported by the DMA controller hardware. The following datawidth options can be enabled or disabled:

- Byte
- Halfword (two bytes)
- Word (four bytes)
- Doubleword (eight bytes)
- Quadword (sixteen bytes)

Disabling unnecessary transfer widths reduces the number of on-chip logic resources consumed by the DMA controller core. For example, if a system has both 16-bit and 32-bit memories, but the DMA controller transfers data to the 16-bit memory, 32-bit transfers could be disabled to conserve logic resources.

Device Support

The DMA Controller Core with Avalon Interface supports all Altera device families.

Software Programming Model

This section describes the programming model for the DMA controller, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides HAL system library drivers that enable you to access the DMA controller core using the HAL API for DMA devices.

HAL System Library Support

The Altera-provided driver implements a HAL DMA device driver that integrates into the HAL system library for Nios II systems. HAL users should access the DMA controller via the familiar HAL API, rather than accessing the registers directly.

If your program uses the HAL device driver to access the DMA controller, accessing the device registers directly interferes with the correct behavior of the driver.

The HAL DMA driver provides both ends of the DMA process; the driver registers itself as both a receive channel (`alt_dma_rxchan`) and a transmit channel (`alt_dma_txchan`). The *Nios II Software Developer's Handbook* provides complete details of the HAL system library and the usage of DMA devices.

ioctl() Operations

`ioctl()` operation requests are defined for both the receive and transmit channels, which allows you to control the hardware-dependent aspects of the DMA controller. Two `ioctl()` functions are defined for the receiver driver and the transmitter driver: `alt_dma_rxchan_ioctl()` and `alt_dma_txchan_ioctl()`. Table 22–2 lists the available operations. These are valid for both the transmit and receive channels.

Table 22–2. Operations for alt_dma_rxchan_ioctl() and alt_dma_txchan_ioctl()

Request	Meaning
`ALT_DMA_SET_MODE_8`	Transfers data in units of 8 bits. The parameter `arg` is ignored.
`ALT_DMA_SET_MODE_16`	Transfers data in units of 16 bits. The parameter `arg` is ignored.
`ALT_DMA_SET_MODE_32`	Transfers data in units of 32 bits. The parameter `arg` is ignored.
`ALT_DMA_SET_MODE_64`	Transfers data in units of 64 bits. The parameter `arg` is ignored.
`ALT_DMA_SET_MODE_128`	Transfers data in units of 128 bits. The parameter `arg` is ignored.
`ALT_DMA_RX_ONLY_ON` *(1)*	Sets a DMA receiver into streaming mode. In this case, data is read continuously from a single location. The parameter `arg` specifies the address to read from.
`ALT_DMA_RX_ONLY_OFF` *(1)*	Turns off streaming mode for a receive channel. The parameter `arg` is ignored.
`ALT_DMA_TX_ONLY_ON` *(1)*	Sets a DMA transmitter into streaming mode. In this case, data is written continuously to a single location. The parameter `arg` specifies the address to write to.
`ALT_DMA_TX_ONLY_OFF` *(1)*	Turns off streaming mode for a transmit channel. The parameter `arg` is ignored.

Note to Table 22–2:

(1) These macro names changed in version 1.1 of the Nios II Embedded Design Suite (EDS). The old names (`ALT_DMA_TX_STREAM_ON`, `ALT_DMA_TX_STREAM_OFF`, `ALT_DMA_RX_STREAM_ON`, and `ALT_DMA_RX_STREAM_OFF`) are still valid, but new designs should use the new names.

Limitations

Currently the Altera-provided drivers do not support 64-bit and 128-bit DMA transactions.

This function is not thread safe. If you want to access the DMA controller from more than one thread then you should use a semaphore or mutex to ensure that only one thread is executing within this function at any time.

Software Files

The DMA controller is accompanied by the following software files. These files define the low-level interface to the hardware. Application developers should not modify these files.

- **altera_avalon_dma_regs.h**—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used only by device driver functions.
- **altera_avalon_dma.h**, **altera_avalon_dma.c**—These files implement the DMA controller's device driver for the HAL system library.

Register Map

Programmers using the HAL API never access the DMA controller hardware directly via its registers. In general, the register map is only useful to programmers writing a device driver.

The Altera-provided HAL device driver accesses the device registers directly. If you are writing a device driver, and the HAL driver is active for the same device, your driver will conflict and fail to operate.

Table 22–3 shows the register map for the DMA controller. Device drivers control and communicate with the hardware through five memory-mapped 32-bit registers.

Table 22–3. DMA Controller Register Map

Offset	Register Name	Read/Write	31 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	`status` *(1)*	RW	*(2)*									LEN	WEOP	REOP	BUSY	DONE
1	`readaddress`	RW	Read master start address													
2	`writeaddress`	RW	Write master start address													
3	`length`	RW	DMA transaction length (in bytes)													
4	—	—	Reserved *(3)*													
5	—	—	Reserved *(3)*													
6	`control`	RW	*(2)*	SOFTWARERESET	QUADWORD	DOUBLEWORD	WCON	RCON	LEEN	WEEN	REEN	I_EN	GO	WORD	HW	BYTE
7	—	—	Reserved *(3)*													

Notes to Table 22–3:

(1) Writing zero to the `status` register clears the `LEN`, `WEOP`, `REOP`, and `DONE` bits.

(2) These bits are reserved. Read values are undefined. Write zero.

(3) This register is reserved. Read values are undefined. The result of a write is undefined.

status Register

The `status` register consists of individual bits that indicate conditions inside the DMA controller. The `status` register can be read at any time. Reading the `status` register does not change its value.

The `status` register bits are shown in Table 22–4.

Table 22–4. status Register Bits (Part 1 of 2)

Bit Number	Bit Name	Read/Write/Clear	Description
0	`DONE`	R/C	A DMA transaction is complete. The `DONE` bit is set to 1 when an end of packet condition is detected or the specified transaction length is completed. Write zero to the `status` register to clear the `DONE` bit.
1	`BUSY`	R	The `BUSY` bit is 1 when a DMA transaction is in progress.

Table 22–4. status Register Bits (Part 2 of 2)

Bit Number	Bit Name	Read/Write/Clear	Description
2	REOP	R	The REOP bit is 1 when a transaction is completed due to an end-of-packet event on the read side.
3	WEOP	R	The WEOP bit is 1 when a transaction is completed due to an end of packet event on the write side.
4	LEN	R	The LEN bit is set to 1 when the length register decrements to zero.

readaddress Register

The readaddress register specifies the first location to be read in a DMA transaction. The readaddress register width is determined at system generation time. It is wide enough to address the full range of all slave ports mastered by the read port.

writeaddress Register

The writeaddress register specifies the first location to be written in a DMA transaction. The writeaddress register width is determined at system generation time. It is wide enough to address the full range of all slave ports mastered by the write port.

length Register

The length register specifies the number of bytes to be transferred from the read port to the write port. The length register is specified in bytes. For example, the value must be a multiple of 4 for word transfers, and a multiple of 2 for halfword transfers.

The length register is decremented as each data value is written by the write master port. When length reaches 0 the LEN bit is set. The length register does not decrement below 0.

The length register width is determined at system generation time. It is at least wide enough to span any of the slave ports mastered by the read or write master ports, and it can be made wider if necessary.

control Register

The control register is composed of individual bits that control the DMA's internal operation. The control register's value can be read at any time. The control register bits determine which, if any, conditions of the DMA transaction result in the end of a transaction and an interrupt request.

The control register bits are shown in Table 22–5.

Table 22–5. Control Register Bits (Part 1 of 2)

Bit Number	Bit Name	Read/ Write/ Clear	Description
0	BYTE	RW	Specifies byte transfers.
1	HW	RW	Specifies halfword (16-bit) transfers.
2	WORD	RW	Specifies word (32-bit) transfers.

 © March 2009 Altera Corporation

Table 22–5. Control Register Bits (Part 2 of 2)

Bit Number	Bit Name	Read/ Write/ Clear	Description
3	GO	RW	Enables DMA transaction. When the GO bit is set to 0, the DMA is prevented from executing transfers. When the GO bit is set to 1 and the length register is non-zero, transfers occur.
4	I_EN	RW	Enables interrupt requests (IRQ). When the I_EN bit is 1, the DMA controller generates an IRQ when the status register's DONE bit is set to 1. IRQs are disabled when the I_EN bit is 0.
5	REEN	RW	Ends transaction on read-side end-of-packet. When the REEN bit is set to 1, a slave port with flow control on the read side may end the DMA transaction by asserting its end-of-packet signal.
6	WEEN	RW	Ends transaction on write-side end-of-packet. When the WEEN bit is set to 1, a slave port with flow control on the write side may end the DMA transaction by asserting its end-of-packet signal.
7	LEEN	RW	Ends transaction when the length register reaches zero. When the LEEN bit is 1, the DMA transaction ends when the length register reaches 0. When this bit is 0, length reaching 0 does not cause a transaction to end. In this case, the DMA transaction must be terminated by an end-of-packet signal from either the read or write master port.
8	RCON	RW	Reads from a constant address. When RCON is 0, the read address increments after every data transfer. This is the mechanism for the DMA controller to read a range of memory addresses. When RCON is 1, the read address does not increment. This is the mechanism for the DMA controller to read from a peripheral at a constant memory address. For details, see "Addressing and Address Incrementing" on page 22–3.
9	WCON	RW	Writes to a constant address. Similar to the RCON bit, when WCON is 0 the write address increments after every data transfer; when WCON is 1 the write address does not increment. For details, see "Addressing and Address Incrementing" on page 22–3.
10	DOUBLEWORD	RW	Specifies doubleword transfers.
11	QUADWORD	RW	Specifies quadword transfers.
12	SOFTWARERESET	RW	Software can reset the DMA engine by writing this bit to 1 twice. Upon the second write of 1 to the SOFTWARERESET bit, the DMA control is reset identically to a system reset. The logic which sequences the software reset process then resets itself automatically.

The data width of DMA transactions is specified by the BYTE, HW, WORD, DOUBLEWORD, and QUADWORD bits. Only one of these bits can be set at a time. If more than one of the bits is set, the DMA controller behavior is undefined. The width of the transfer is determined by the narrower of the two slaves read and written. For example, a DMA transaction that reads from a 16-bit flash memory and writes to a 32-bit on-chip memory requires a halfword transfer. In this case, HW must be set to 1, and BYTE, WORD, DOUBLEWORD, and QUADWORD must be set to 0.

To successfully perform transactions of a specific width, that width must be enabled in hardware using the **Allowed Transaction** hardware option. For example, the DMA controller behavior is undefined if quadword transfers are disabled in hardware, but the QUADWORD bit is set during a DMA transaction.

Executing a DMA software reset when a DMA transfer is active may result in permanent bus lockup (until the next system reset). The SOFTWARERESET bit should therefore not be written except as a last resort.

Interrupt Behavior

The DMA controller has a single IRQ output that is asserted when the status register's DONE bit equals 1 and the control register's I_EN bit equals 1.

Writing the status register clears the DONE bit and acknowledges the IRQ. A master peripheral can read the status register and determine how the DMA transaction finished by checking the LEN, REOP, and WEOP bits.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 22–6 shows the revision history for this chapter.

Table 22–6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Updated the Functional Description of the core.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

23. Video Sync Generator and Pixel Converter Cores

QII55006-9.0.0

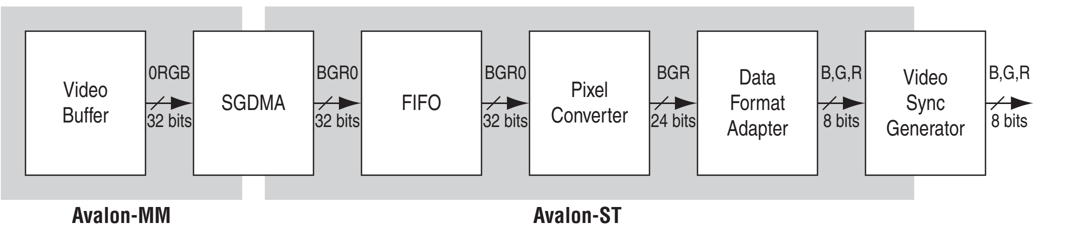
Core Overview

The video sync generator core accepts a continuous stream of pixel data in RGB format, and outputs the data to an off-chip display controller with proper timing. You can configure the video sync generator core to support different display resolutions and synchronization timings.

The pixel converter core transforms the pixel data to the format required by the video sync generator. Figure 23–1 shows a typical placement of the video sync generator and pixel converter cores in a system.

In this example, the video buffer stores the pixel data in 32-bit unpacked format. The extra byte in the pixel data is discarded by the pixel converter core before the data is serialized and sent to the video sync generator core.

Figure 23–1. Typical Placement in a System



The video sync generator and pixel converter cores are SOPC Builder-ready and integrate easily into any SOPC Builder-generated system.

These cores are deployed in the Nios II Embedded Software Evaluation Kit (NEEK), which includes an LCD display daughtercard assembly attached via an HSMC connector.

This chapter contains the following sections:

- "Video Sync Generator" on page 23–2
- "Pixel Converter" on page 23–5
- "Device Support" on page 23–6
- "Hardware Simulation Considerations" on page 23–6

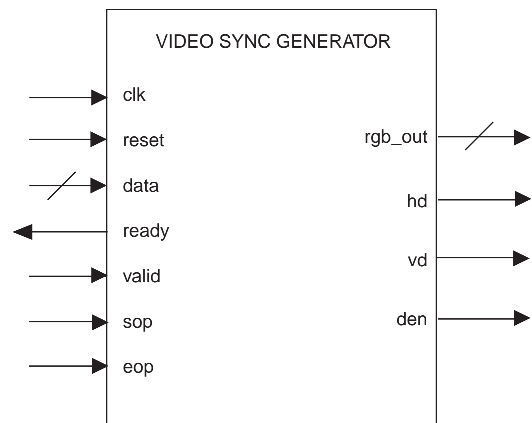
Video Sync Generator

This section describes the hardware structure and functionality of the video sync generator core.

Functional Description

The video sync generator core adds horizontal and vertical synchronization signals to the pixel data that comes through its Avalon® (Avalon-ST) input interface and outputs the data to an off-chip display controller. No processing or validation is performed on the pixel data. Figure 23–2 shows a block diagram of the video sync generator.

Figure 23–2. Video Sync Generator Block Diagram



You can configure various aspects of the core and its Avalon-ST interface to suit your requirements. You can specify the data width, number of beats required to transfer each pixel and synchronization signals. See "Instantiating the Core in SOPC Builder" on page 23–3 for more information on the available options.

To ensure incoming pixel data is sent to the display controller with correct timing, the video sync generator core must synchronize itself to the first pixel in a frame. The first active pixel is indicated by an `sop` pulse.

The video sync generator core expects continuous streams of pixel data at its input interface and assumes that each incoming packet contains the correct number of pixels (Number of rows * Number of columns). Data starvation disrupts synchronization and results in unexpected output on the display.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the video sync generator core in SOPC Builder to configure the core. Table 23–1 lists the available parameters in the MegaWizard interface.

Table 23–1. Video Sync Generator Parameters

Parameter Name	Description
Horizontal Sync Pulse Pixels	The width of the h-sync pulse in number of pixels.
Total Vertical Scan Lines	The total number of lines in one video frame. The value is the sum of the following parameters: **Number of Rows**, **Vertical Blank Lines**, and **Vertical Front Porch Lines**.
Number of Rows	The number of active scan lines in each video frame.
Horizontal Sync Pulse Polarity	The polarity of the h-sync pulse; 0 = active low and 1 = active high.
Horizontal Front Porch Pixels	The number of blanking pixels that follow the active pixels. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Vertical Sync Pulse Polarity	The polarity of the v-sync pulse; 0 = active low and 1 = active high.
Vertical Sync Pulse Lines	The width of the v-sync pulse in number of lines.
Vertical Front Porch Lines	The number of blanking lines that follow the active lines. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Number of Columns	The number of active pixels in each line.
Horizontal Blank Pixels	The number of blanking pixels that precede the active pixels. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Total Horizontal Scan Pixels	The total number of pixels in one line. The value is the sum of the following parameters: **Number of Columns**, **Horizontal Blank Pixel**, and **Horizontal Front Porch Pixels**.
Beats Per Pixel	The number of beats required to transfer one pixel. Valid values are 1 and 3. This parameter, when multiplied by **Data Stream Bit Width** must be equal to the total number of bits in one pixel. This parameter affects the operating clock frequency, as shown in the following equation: Operating clock frequency = (**Beats per pixel**) * (Pixel_rate), where Pixel_rate (in MHz) = ((**Total Horizontal Scan Pixels**) * (**Total Vertical Scan Lines**) * (Display refresh rate in Hz))/1000000.
Vertical Blank Lines	The number of blanking lines that proceed the active lines. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Data Stream Bit Width	The width of the inbound and outbound data.

Signals

Table 23–2 lists the input and output signals for the video sync generator core.

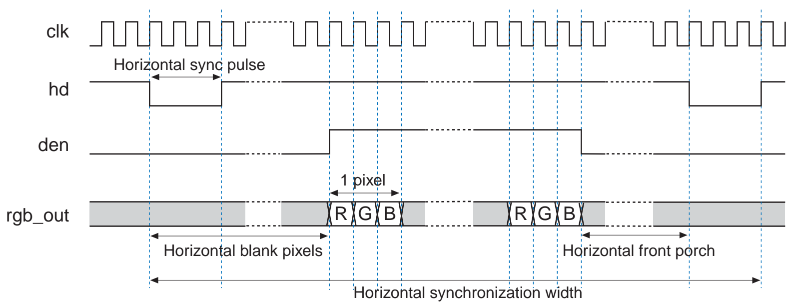
Table 23–2. Video Sync Generator Core Signals

Signal Name	Width (Bits)	Direction	Description
Global Signals			
clk	1	Input	System clock.
reset	1	Input	System reset.
Avalon-ST Signals			
data	Variable-width	Input	Incoming pixel data. The datawidth is determined by the parameter **Data Stream Bit Width**.
ready	1	Output	This signal is asserted when the video sync generator is ready to receive the pixel data.
valid	1	Input	This signal is not used by the video sync generator core because the core always expects valid pixel data on the next clock cycle after the ready signal is asserted.
sop	1	Input	Start-of-packet. This signal is asserted when the first pixel is received.
eop	1	Input	End-of-packet. This signal is asserted when the last pixel is received.
LCD Output Signals			
rgb_out	Variable-width	Output	Display data. The datawidth is determined by the parameter **Data Stream Bit Width**.
hd	1	Output	Horizontal synchronization pulse for display.
vd	1	Output	Vertical synchronization pulse for display.
den	1	Output	This signal is asserted when the video sync generator core outputs valid data for display.

Timing Diagrams

The horizontal and vertical synchronization timings are determined by the parameters setting. Figure 23–3 shows the horizontal synchronization timing when the parameters **Data Stream Bit Width** and **Beats Per Pixel** are set to 8 and 3, respectively.

Figure 23–3. Horizontal Synchronization Timing—8 Bits DataWidth and 3 Beats Per Pixel



 © March 2009 Altera Corporation

Figure 23–4 sho.ws the horizontal synchronization timing when the parameters **Data Stream Bit Width** and **Beats Per Pixel** are set to 24 and 1, respectively.

Figure 23–4. Horizontal Synchronization Timing—24 Bits DataWidth and 1 Beat Per Pixel

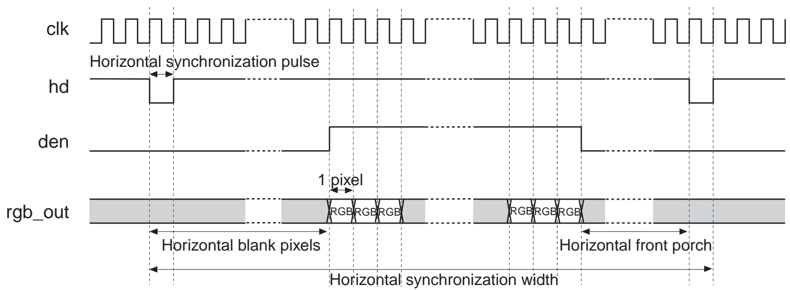
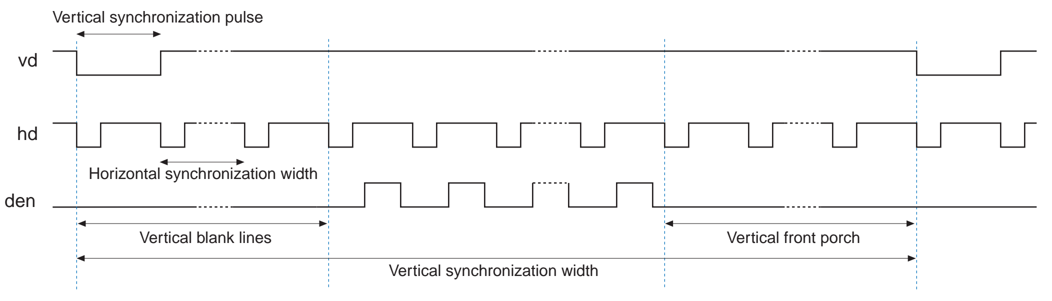


Figure 23–5 shows the vertical synchronization timing.

Figure 23–5. Vertical Synchronization Timing—8 Bits DataWidth and 3 Beats Per Pixel / 24 Bits DataWidth and 1 Beat Per Pixel



Pixel Converter

This section describes the hardware structure and functionality of the pixel converter core.

Functional Description

The pixel converter core receives pixel data on its Avalon-ST input interface and transforms the pixel data to the format required by the video sync generator. The least significant byte of the 32-bit wide pixel data is removed and the remaining 24 bits are wired directly to the core's Avalon-ST output interface.

Instantiating the Core in SOPC Builder

Use the MegaWizard interface for the pixel converter core in SOPC Builder to add the core to a system. You can configure the following parameter:

Source symbols per beat—The number of symbols per beat on the Avalon-ST source interface.

Signals

Table 23–3 lists the input and output signals for the pixel converter core.

Table 23–3. Pixel Converter Input Interface Signals

Signal Name	Width (Bits)	Direction	Description
Global Signals			
clk	1	Input	Not in use.
reset_n	1	Input	
Avalon-ST Signals			
data_in	32	Input	Incoming pixel data. Contains four 8-bit symbols that are transferred in 1 beat.
data_out	24	Output	Output data. Contains three 8-bit symbols that are transferred in 1 beat.
sop_in	1	Input	Wired directly to the corresponding output signals.
eop_in	1	Input	
ready_in	1	Input	
valid_in	1	Input	
empty_in	1	Input	
sop_out	1	Output	Wired directly from the input signals.
eop_out	1	Output	
ready_out	1	Output	
valid_out	1	Output	
empty_out	1	Output	

Device Support

The video sync generator and pixel converter cores support all Altera device families.

Hardware Simulation Considerations

For a typical 60 Hz refresh rate, set the simulation length for the video sync generator core to at least 16.7 µs to get a full video frame. Depending on the size of the video frame, simulation may take a very long time to complete.

Referenced Documents

This chapter references *Avalon Interface Specifications*.

Document Revision History

Table 23–4 shows the revision history for this chapter.

Table 23–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Added new parameters for both cores.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

24. Interval Timer Core

NII51008-9.0.0

Core Overview

The Interval Timer core with Avalon® interface is an interval timer for Avalon-based processor systems, such as a Nios® II processor system. The core provides the following features:

- 32-bit and 64-bit counters.
- Controls to start, stop, and reset the timer.
- Two count modes: count down once and continuous count-down.
- Count-down period register.
- Option to enable or disable the interrupt request (IRQ) when timer reaches zero.
- Optional watchdog timer feature that resets the system if timer ever reaches zero.
- Optional periodic pulse generator feature that outputs a pulse when timer reaches zero.
- Compatible with 32-bit and 16-bit processors.

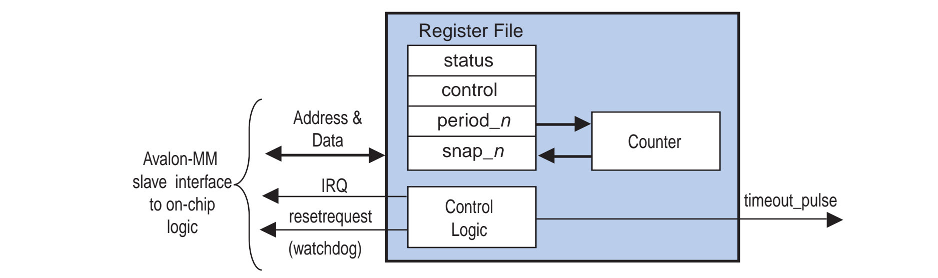
Device drivers are provided in the HAL system library for the Nios II processor. The interval timer core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 24–2
- "Instantiating the Core in SOPC Builder" on page 24–3
- "Software Programming Model" on page 24–5

Functional Description

Figure 24–1 shows a block diagram of the interval timer core.

Figure 24–1. Interval Timer Core Block Diagram



The intervanl timer core has two user-visible features:

- The Avalon Memory-Mapped (Avalon-MM) interface that provides access to six 16-bit registers
- An optional pulse output that can be used as a periodic pulse generator

All registers are 16-bits wide, making the core compatible with both 16-bit and 32-bit processors. Certain registers only exist in hardware for a given configuration. For example, if the core is configured with a fixed period, the period registers do not exist in hardware.

The following sequence describes the basic behavior of the interval timer core:

- An Avalon-MM master peripheral, such as a Nios II processor, writes the core's `control` register to perform the following tasks:
  - Start and stop the timer
  - Enable/disable the IRQ
  - Specify count-down once or continuous count-down mode
- A processor reads the `status` register for information about current timer activity.
- A processor can specify the timer period by writing a value to the period registers.
- An internal counter counts down to zero, and whenever it reaches zero, it is immediately reloaded from the period registers.
- A processor can read the current counter value by first writing to one of the snap registers to request a coherent snapshot of the counter, and then reading the snap registers for the full value.
- When the count reaches zero, one or more of the following events are triggered:
  - If IRQs are enabled, an IRQ is generated.
  - The optional pulse-generator output is asserted for one clock period.
  - The optional watchdog output resets the system.

Avalon-MM Slave Interface

The interval timer core implements a simple Avalon-MM slave interface to provide access to the register file. The Avalon-MM slave port uses the `resetrequest` signal to implement watchdog timer behavior. This signal is a non-maskable reset signal, and it drives the reset input of all Avalon-MM peripherals in the SOPC Builder system. When the `resetrequest` signal is asserted, it forces any processor connected to the system to reboot. For more information, refer to "Configuring the Timer as a Watchdog Timer" on page 24–4.

Device Support

The interval timer core supports all Altera® device families.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the interval timer core in SOPC Builder to specify the hardware features. This section describes the options available in the MegaWizard Interace.

Timeout Period

The **Timeout Period** setting determines the initial value of the period registers. When the **Writeable period** option is on, a processor can change the value of the period by writing to the period registers. When the **Writeable period** option is off, the period is fixed and cannot be updated at runtime. See "Hardware Options" on page 24–3 for information on register options.

The **Timeout Period** is an integer multiple of the **Timer Frequency**. The **Timer Frequency** is fixed at the frequency setting of the system clock associated with the timer. The **Timeout Period** setting can be specified in units of **µs** (microseconds), **ms** (milliseconds), **seconds** , or **clocks** (number of cycles of the system clock associated with the timer). The actual period depends on the frequency of the system clock associated with the timer. If the period is specified in **µs**, **ms**, or **seconds**, the true period will be the smallest number of clock cycles that is greater or equal to the specified **Timeout Period** value. For example, if the associated system clock has a frequency of 30 **ns**, and the specified **Timeout Period** value is 1 **µs**, the true timeout period will be 1.020 microseconds.

Counter Size

The **Counter Size** setting determines the timer's width, which can be set to either 32 or 64 bits. A 32-bit timer has two 16-bit period registers, whereas a 64-bit timer has four 16-bit period registers. This option applies to the snap registers as well.

Hardware Options

The following options affect the hardware structure of the interval timer core. As a convenience, the **Preset Configurations** list offers several pre-defined hardware configurations, such as:

- **Simple periodic interrupt**—This configuration is useful for systems that require only a periodic IRQ generator. The period is fixed and the timer cannot be stopped, but the IRQ can be disabled.
- **Full-featured**—This configuration is useful for embedded processor systems that require a timer with variable period that can be started and stopped under processor control.
- **Watchdog—**This configuration is useful for systems that require watchdog timer to reset the system in the event that the system has stopped responding. Refer to "Configuring the Timer as a Watchdog Timer" on page 24–4.

Register Options

Table 24–1 shows the settings that affect the interval timer core's registers.

Table 24–1. Register Options

Option	Description
Writeable period	When this option is enabled, a master peripheral can change the count-down period by writing to the period registers. When disabled, the count-down period is fixed at the specified **Timeout Period**, and the period registers do not exist in hardware.
Readable snapshot	When this option is enabled, a master peripheral can read a snapshot of the current count-down. When disabled, the status of the counter is detectable only via other indicators, such as the `status` register or the IRQ signal. In this case, the snap registers do not exist in hardware, and reading these registers produces an undefined value.
Start/Stop control bits	When this option is enabled, a master peripheral can start and stop the timer by writing the START and STOP bits in the `control` register. When disabled, the timer runs continuously. When the **System reset on timeout (watchdog)** option is enabled, the START bit is also present, regardless of the **Start/Stop control bits** option.

Output Signal Options

Table 24–2 shows the settings that affect the interval timer core's output signals.

Table 24–2. Output Signal Options

Option	Description
Timeout pulse (1 clock wide)	When this option is on, the core outputs a signal `timeout_pulse`. This signal pulses high for one clock cycle whenever the timer reaches zero. When this option is off, the `timeout_pulse` signal does not exist.
System reset on timeout (watchdog)	When this option is on, the core's Avalon-MM slave port includes the `resetrequest` signal. This signal pulses high for one clock cycle whenever the timer reaches zero resulting in a system-wide reset. The internal timer is stopped at reset. Explicitly writing the START bit of the `control` register starts the timer. When this option is off, the `resetrequest` signal does not exist. Refer to "Configuring the Timer as a Watchdog Timer".

Configuring the Timer as a Watchdog Timer

To configure the core for use as a watchdog, in the MegaWizard Interface select **Watchdog** in the **Preset Configurations** list, or choose the following settings:

- Set the **Timeout Period** to the desired "watchdog" period.
- Turn off **Writeable period**.
- Turn off **Readable snapshot**.
- Turn off **Start/Stop control bits**.
- Turn off **Timeout pulse**.
- Turn on **System reset on timeout (watchdog)**.

© March 2009 Altera Corporation

A watchdog timer wakes up (comes out of reset) stopped. A processor later starts the timer by writing a 1 to the `control` register's `START` bit. Once started, the timer can never be stopped. If the internal counter ever reaches zero, the watchdog timer resets the system by generating a pulse on its `resetrequest` output. To prevent the system from resetting, the processor must periodically reset the timer's count-down value by writing one of the period registers (the written value is ignored). If the processor fails to access the timer because, for example, software stopped executing normally, the watchdog timer resets the system and returns the system to a defined state.

Software Programming Model

The following sections describe the software programming model for the interval timer core, including the register map and software declarations to access the hardware. For Nios II processor users, Altera provides hardware abstraction layer (HAL) system library drivers that enable you to access the interval timer core using the HAL application programming interface (API) functions.

HAL System Library Support

The Altera-provided drivers integrate into the HAL system library for Nios II systems. When possible, HAL users should access the core via the HAL API, rather than accessing the core's registers directly.

Altera provides a driver for both the HAL timer device models: system clock timer, and timestamp timer.

System Clock Driver

When configured as the system clock, the interval timer core runs continuously in periodic mode, using the default period set in SOPC builder. The system clock services are then run as a part of the interrupt service routine for this timer. The driver is interrupt-driven, and therefore must have its interrupt signal connected in the system hardware.

The Nios II integrated development environment (IDE) allows you to specify system library properties that determine which timer device will be used as the system clock timer.

Timestamp Driver

The interval timer core may be used as a timestamp device if it meets the following conditions:

- The timer has a writeable `period` register, as configured in SOPC Builder.
- The timer is not selected as the system clock.

The Nios II IDE allows you to specify system library properties that determine which timer device will be used as the timestamp timer.

If the timer hardware is not configured with writeable `period` registers, calls to the `alt_timestamp_start()` API function will not reset the timestamp counter. All other HAL API calls will perform as expected.

For more information about using the system clock and timestamp features that use these drivers, refer to the *Nios II Software Developer's Handbook*. The Nios II Embedded Design Suite (EDS) also provides several example designs that use the interval timer core.

Limitations

The HAL driver for the interval timer core does not support the watchdog reset feature of the core.

Software Files

The interval timer core is accompanied by the following software files. These files define the low-level interface to the hardware, and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_timer_regs.h—**This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_timer.h**, **altera_avalon_timer_sc.c, altera_avalon_timer_ts.c, altera_avalon_timer_vars.c—**These files implement the timer device drivers for the HAL system library.

Register Map

You do not need to access the interval timer core directly via its registers if using the standard features provided in the HAL system library for the Nios II processor. In general, the register map is only useful to programmers writing a device driver.

The Altera-provided HAL device driver accesses the device registers directly. If you are writing a device driver, and the HAL driver is active for the same device, your driver will conflict and fail to operate correctly.

Table 24–3 shows the register map for the 32-bit timer.

Table 24–3. Register Map—32-bit Timer

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	`status`	RW	*(1)*					`RUN`	`TO`
1	`control`	RW	*(1)*			`STOP`	`START`	`CONT`	`ITO`
2	`periodl`	RW	Timeout Period – 1 (bits [15:0])						
3	`periodh`	RW	Timeout Period – 1 (bits [31:16])						
4	`snapl`	RW	Counter Snapshot (bits [15:0])						
5	`snaph`	RW	Counter Snapshot (bits [31:16])						

Note to Table 24–3:

(1) Reserved. Read values are undefined. Write zero.

Table 24–4 shows the register map for the 64-bit timer.

Table 24–4. Register Map—64-bit Timer

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	status	RW	*(1)*					RUN	TO
1	control	RW	*(1)*			STOP	START	CONT	ITO
2	period_0	RW	Timeout Period – 1 (bits [15:0])						
3	period_1	RW	Timeout Period – 1 (bits [31:16])						
4	period_2	RW	Timeout Period – 1 (bits [47:32])						
5	period_3	RW	Timeout Period – 1 (bits [63:48])						
6	snap_0	RW	Counter Snapshot (bits [15:0])						
7	snap_1	RW	Counter Snapshot (bits [31:16])						
8	snap_2	RW	Counter Snapshot (bits [47:32])						
9	snap_3	RW	Counter Snapshot (bits [63:48])						

Note to Table 24–4:

(1) Reserved. Read values are undefined. Write zero.

status Register

The status register has two defined bits, as shown in Table 24–5.

Table 24–5. status Register Bits

Bit	Name	R/W/C	Description
0	TO	RC	The TO (timeout) bit is set to 1 when the internal counter reaches zero. Once set by a timeout event, the TO bit stays set until explicitly cleared by a master peripheral. Write zero to the status register to clear the TO bit.
1	RUN	R	The RUN bit reads as 1 when the internal counter is running; otherwise this bit reads as 0. The RUN bit is not changed by a write operation to the status register.

control Register

The control register has four defined bits, as shown in Table 24–6.

Table 24–6. control Register Bits (Part 1 of 2)

Bit	Name	R/W/C	Description
0	ITO	RW	If the ITO bit is 1, the interval timer core generates an IRQ when the status register's TO bit is 1. When the ITO bit is 0, the timer does not generate IRQs.
1	CONT	RW	The CONT (continuous) bit determines how the internal counter behaves when it reaches zero. If the CONT bit is 1, the counter runs continuously until it is stopped by the STOP bit. If CONT is 0, the counter stops after it reaches zero. When the counter reaches zero, it reloads with the value stored in the period registers, regardless of the CONT bit.
2	START *(1)*	W	Writing a 1 to the START bit starts the internal counter running (counting down). The START bit is an event bit that enables the counter when a write operation is performed. If the timer is stopped, writing a 1 to the START bit causes the timer to restart counting from the number currently stored in its counter. If the timer is already running, writing a 1 to START has no effect. Writing 0 to the START bit has no effect.

Table 24–6. control Register Bits (Part 2 of 2)

Bit	Name	R/W/C	Description
3	STOP *(1)*	W	Writing a 1 to the `STOP` bit stops the internal counter. The `STOP` bit is an event bit that causes the counter to stop when a write operation is performed. If the timer is already stopped, writing a 1 to `STOP` has no effect. Writing a 0 to the stop bit has no effect. If the timer hardware is configured with **Start/Stop control bits** off, writing the `STOP` bit has no effect.

Note to Table 24–6:

(1) Writing 1 to both `START` and `STOP` bits simultaneously produces an undefined result.

period_n Registers

The `period_`*n* registers together store the timeout period value. The internal counter is loaded with the value stored in these registers whenever one of the following occurs:

- A write operation to one of the `period_`*n* register
- The internal counter reaches 0

The timer's actual period is one cycle greater than the value stored in the `period_`*n* registers because the counter assumes the value zero for one clock cycle.

Writing to one of the `period_`*n* registers stops the internal counter, except when the hardware is configured with **Start/Stop control bits** off. If **Start/Stop control bits** is off, writing either register does not stop the counter. When the hardware is configured with **Writeable period** disabled, writing to one of the `period_`*n* registers causes the counter to reset to the fixed **Timeout Period** specified at system generation time.

snap_n Registers

A master peripheral may request a coherent snapshot of the current internal counter by performing a write operation (write-data ignored) to one of the `snap_`*n* registers. When a write occurs, the value of the counter is copied to `snap_`*n* registers. The snapshot occurs whether or not the counter is running. Requesting a snapshot does not change the internal counter's operation.

Interrupt Behavior

The interval timer core generates an IRQ whenever the internal counter reaches zero and the `ITO` bit of the `control` register is set to 1. Acknowledge the IRQ in one of two ways:

- Clear the `TO` bit of the `status` register
- Disable interrupts by clearing the `ITO` bit of the `control` register

Failure to acknowledge the IRQ produces an undefined result.

Referenced Documents

This chapter references the *Nios II Software Developer's Handbook*.

Document Revision History

Table 24–7 shows the revision history for this chapter.

Table 24–7. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. Updated the core's name to reflect the name used in SOPC Builder.	—
May 2008 v8.0.0	Added a new parameter and register map for the 64-bit timer.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

25. System ID Core

NII51014-9.0.0

Core Overview

The system ID core with Avalon® interface is a simple read-only device that provides SOPC Builder systems with a unique identifier. Nios® II processor systems use the system ID core to verify that an executable program was compiled targeting the actual hardware image configured in the target FPGA. If the expected ID in the executable does not match the system ID core in the FPGA, it is possible that the software will not execute correctly.

This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 25–2
- "Instantiating the Core in SOPC Builder" on page 25–2
- "Software Programming Model" on page 25–2

Functional Description

The system ID core provides a read-only Avalon Memory-Mapped (Avalon-MM) slave interface. This interface has two 32-bit registers, as shown in Table 25–1. The value of each register is determined at system generation time, and always returns a constant value.

Table 25–1. System ID Core Register Map

Offset	Register Name	R/W	Description
0	`id`	R	A unique 32-bit value that is based on the contents of the SOPC Builder system. The id is similar to a check-sum value; SOPC Builder systems with different components, different configuration options, or both, produce different id values.
1	`timestamp`	R	A unique 32-bit value that is based on the system generation time. The value is equivalent to the number of seconds after Jan. 1, 1970.

There are two basic ways to use the system ID core:

- Verify the system ID before downloading new software to a system. This method is used by software development tools, such as the Nios II integrated development environment (IDE). There is little point in downloading a program to a target hardware system, if the program is compiled for different hardware. Therefore, the Nios II IDE checks that the system ID core in hardware matches the expected system ID of the software before downloading a program to run or debug.
- Check system ID after reset. If a program is running on hardware other than the expected SOPC Builder system, the program may fail to function altogether. If the program does not crash, it can behave erroneously in subtle ways that are difficult to debug. To protect against this case, a program can compare the expected system ID against the system ID core, and report an error if they do not match.

Device Support

The system ID core supports all Altera® device families.

Instantiating the Core in SOPC Builder

The System ID core has no user-configurable features. The `id` and `timestamp` register values are determined at system generation time based on the configuration of the SOPC Builder system and the current time. You can add only one system ID core to an SOPC Builder system, and its name is always `sysid`.

After system generation, you can examine the values stored in the `id` and `timestamp` registers by opening the MegaWizard™ Interface for the System ID core. Hovering the mouse over the component in SOPC Builder also displays a tool-tip showing the values.

Software Programming Model

This section describes the software programming model for the system ID core. For Nios II processor users, Altera provides the HAL system library header file that defines the system ID core registers.

The System ID core comes with the following software files. These files provide low-level access to the hardware. Application developers should not modify these files.

- **alt_avalon_sysid_regs.h**—Defines the interface to the hardware registers.
- **alt_avalon_sysid.c, alt_avalon_sysid.h**—Header and source files defining the hardware access functions.

Altera provides one access routine, `alt_avalon_sysid_test()`, that returns a value indicating whether the system ID expected by software matches the system ID core.

alt_avalon_sysid_test()

Prototype:	`alt_32 alt_avalon_sysid_test(void)`
Thread-safe:	No.
Available from ISR:	Yes.
Include:	**<altera_avalon_sysid.h>**
Description:	Returns 0 if the values stored in the hardware registers match the values expected by software. Returns 1 if the hardware timestamp is greater than the software timestamp. Returns -1 if the software timestamp is greater than the hardware timestamp.

Document Revision History

Table 25–2 shows the revision history for this chapter.

Table 25–2. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

26. Mutex Core

NII51020-9.0.0

Core Overview

Multiprocessor environments can use the mutex core with Avalon® interface to coordinate accesses to a shared resource. The mutex core provides a protocol to ensure mutually exclusive ownership of a shared resource.

The mutex core provides a hardware-based atomic test-and-set operation, allowing software in a multiprocessor environment to determine which processor owns the mutex. The mutex core can be used in conjunction with shared memory to implement additional interprocessor coordination features, such as mailboxes and software mutexes.

The mutex core is designed for use in Avalon-based processor systems, such as a Nios® II processor system. Altera provides device drivers for the Nios II processor to enable use of the hardware mutex.

The mutex core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 26–2
- "Instantiating the Core in SOPC Builder" on page 26–2
- "Software Programming Model" on page 26–2
- "Mutex API" on page 26–4

Functional Description

The mutex core has a simple Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to two memory-mapped, 32-bit registers. Table 26–1 shows the registers.

Table 26–1. Mutex Core Register Map

Offset	Register Name	R/W	Bit Description		
			31 … 16	15 … 1	0
0	`mutex`	RW	`OWNER`	`VALUE`	
1	`reset`	RW	Reserved		`RESET`

The mutex core has the following basic behavior. This description assumes there are multiple processors accessing a single mutex core, and each processor has a unique identifier (ID).

- When the `VALUE` field is `0x0000`, the mutex is unlocked and available. Otherwise, the mutex is locked and unavailable.
- The `mutex` register is always readable. Avalon-MM master peripherals, such as a processor, can read the `mutex` register to determine its current state.

- The mutex register is writable only under specific conditions. A write operation changes the mutex register only if one or both of the following conditions are true:
  - The VALUE field of the mutex register is zero.
  - The OWNER field of the mutex register matches the OWNER field in the data to be written.
- A processor attempts to acquire the mutex by writing its ID to the OWNER field, and writing a non-zero value to the VALUE field. The processor then checks if the acquisition succeeded by verifying the OWNER field.
- After system reset, the RESET bit in the reset register is high. Writing a one to this bit clears it.

Device Support

The mutex core supports all Altera device families.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ Interface for the mutex core in SOPC Builder to specify the core's hardware features. The MegaWizard Interface provides the following options:

- **Initial Value**—the initial contents of the VALUE field after reset. If the **Initial Value** setting is non-zero, you must also specify **Initial Owner**.
- **Initial Owner**—the initial contents of the OWNER field after reset. When **Initial Owner** is specified, this owner must release the mutex before it can be acquired by another owner.

Software Programming Model

The following sections describe the software programming model for the mutex core. For Nios II processor users, Altera provides routines to access the mutex core hardware. These functions are specific to the mutex core and directly manipulate low-level hardware. The mutex core cannot be accessed via the HAL API or the ANSI C standard library. In Nios II processor systems, a processor locks the mutex by writing the value of its cpuid control register to the OWNER field of the mutex register.

Software Files

Altera provides the following software files accompanying the mutex core:

- **altera_avalon_mutex_regs.h**—Defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_mutex.h**—Defines data structures and functions to access the mutex core hardware.
- **altera_avalon_mutex.c**—Contains the implementations of the functions to access the mutex core

Hardware Access Routines

This section describes the low-level software constructs for manipulating the mutex core. The file **altera_avalon_mutex.h** declares a structure `alt_mutex_dev` that represents an instance of a mutex device. It also declares routines for accessing the mutex hardware structure, listed in Table 26–2.

Table 26–2. Hardware Access Routines

Function Name	Description
`altera_avalon_mutex_open()`	Claims a handle to a mutex, enabling all the other functions to access the mutex core.
`altera_avalon_mutex_trylock()`	Tries to lock the mutex. Returns immediately if it fails to lock the mutex.
`altera_avalon_mutex_lock()`	Locks the mutex. Will not return until it has successfully claimed the mutex.
`altera_avalon_mutex_unlock()`	Unlocks the mutex.
`altera_avalon_mutex_is_mine()`	Determines if this CPU owns the mutex.
`altera_avalon_mutex_first_lock()`	Tests whether the mutex has been released since reset.

These routines coordinate access to the software mutex structure using a hardware mutex core. For a complete description of each function, see section "Mutex API" on page 26–4.

The code shown in Example 26–1 demonstrates opening a mutex device handle and locking a mutex.

Example 26–1. Opening and Locking a mutex

```
#include <altera_avalon_mutex.h>
/* get the mutex device handle */
alt_mutex_dev* mutex = altera_avalon_mutex_open( "/dev/mutex" );
/* acquire the mutex, setting the value to one */
altera_avalon_mutex_lock( mutex, 1 );
/*
 * Access a shared resource here.
 */
/* release the lock */
altera_avalon_mutex_unlock( mutex );
```

Mutex API

This section describes the application programming interface (API) for the mutex core.

altera_avalon_mutex_is_mine()

Prototype:	`int altera_avalon_mutex_is_mine(alt_mutex_dev* dev)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to test.
Returns:	Returns non zero if the mutex is owned by this CPU.
Description:	`altera_avalon_mutex_is_mine()` determines if this CPU owns the mutex.

altera_avalon_mutex_first_lock()

Prototype:	`int altera_avalon_mutex_first_lock(alt_mutex_dev* dev)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to test.
Returns:	Returns 1 if this mutex has not been released since reset, otherwise returns 0.
Description:	`altera_avalon_mutex_first_lock()` determines whether this mutex has been released since reset.

altera_avalon_mutex_lock()

Prototype:	`void altera_avalon_mutex_lock(alt_mutex_dev* dev, alt_u32 value)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to acquire. `value`—the new value to write to the mutex.
Returns:	—
Description:	`altera_avalon_mutex_lock()` is a blocking routine that acquires a hardware mutex, and at the same time, loads the mutex with the `value` parameter.

altera_avalon_mutex_open()

Prototype:	`alt_mutex_dev* alt_hardware_mutex_open(const char* name)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`name`—the name of the mutex device to open.
Returns:	A pointer to the mutex device structure associated with the supplied name, or NULL if no corresponding mutex device structure was found.
Description:	`altera_avalon_mutex_open()` retrieves a pointer to a hardware mutex device structure.

altera_avalon_mutex_trylock()

Prototype:	`int altera_avalon_mutex_trylock(alt_mutex_dev* dev, alt_u32 value)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to lock. `value`—the new value to write to the mutex.
Returns:	`0` = The mutex was successfully locked. Others = The mutex was not locked.
Description:	`altera_avalon_mutex_trylock()` tries once to lock the hardware mutex, and returns immediately.

altera_avalon_mutex_unlock()

Prototype:	`void altera_avalon_mutex_unlock(alt_mutex_dev* dev)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to unlock.
Returns:	Null.
Description:	`altera_avalon_mutex_unlock()` releases a hardware mutex device. Upon release, the value stored in the mutex is set to zero. If the caller does not hold the mutex, the behavior of this function is undefined.

Document Revision History

Table 26–3 shows the revision history for this chapter.

Table 26–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

 © March 2009 Altera Corporation

27. Mailbox Core

NII53001-9.0.0

Core Overview

Multiprocessor environments can use the mailbox core with Avalon® interface to send messages between processors.

The mailbox core contains mutexes to ensure that only one processor modifies the mailbox contents at a time. The mailbox core must be used in conjunction with a separate shared memory that is used for storing the actual messages.

The mailbox core is designed for use in Avalon-based processor systems, such as a Nios® II processor system. Altera provides device drivers for the Nios II processor. The mailbox core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 27–2
- "Instantiating the Core in SOPC Builder" on page 27–2
- "Software Programming Model" on page 27–3
- "Mailbox API" on page 27–5

Functional Description

The mailbox core has a simple Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to four memory-mapped, 32-bit registers. Table 27–1 shows the registers.

Table 27–1. Mutex Core Register Map

Offset	Register Name	R/W	Bit Description		
			31 … 16	15 … 1	0
0	`mutex0`	RW	`OWNER`	`VALUE`	
1	`reset0`	RW	Reserved		`RESET`
2	`mutex1`	RW	`OWNER`	`VALUE`	
3	`reset1`	RW	Reserved		`RESET`

The mailbox component contains two mutexes: One to ensure unique write access to shared memory and one to ensure unique read access from shared memory. The mailbox core is used in conjunction with a separate memory in the system that is shared among multiple processors.

Mailbox functionality using the mutexes and memory is implemented entirely in the software. Refer to "Software Programming Model" on page 27–3 for details about how to use the mailbox core in software.

For a detailed description of the mutex hardware operation, refer to the *Mutex Core* chapter in volume 5 of the *Quartus II Handbook*.

Device Support

The mailbox core supports all Altera® device families.

Instantiating the Core in SOPC Builder

You can instantiate and configure the mailbox core in an SOPC Builder system using the following process:

1. Decide which processors share the mailbox.
2. On the SOPC Builder **System Contents** tab, instantiate a memory component to serve as the mailbox buffer. Any RAM can be used as the mailbox buffer. The mailbox buffer can share space in an existing memory, such as program memory; it does not require a dedicated memory.
3. On the SOPC Builder **System Contents** tab, instantiate the mailbox component. The mailbox MegaWizard™ Interface presents the following options:
   - **Memory module**—Specifies which memory to use for the mailbox buffer. If the **Memory module** list does not contain the desired shared memory, the memory is not connected in the system correctly. Refer to Step 4 on page 27–2.
   - **CPUs available with this memory**—Shows all the processors that can share the mailbox. This field is always read-only. Use it to verify that the processor connections are correct. If a processor that needs to share the mailbox is missing from the list, refer to Step 4 on page 27–2.
   - **Shared mailbox memory offset**—Specifies an offset into the memory. The mailbox message buffer starts at this offset.
   - **Mailbox size (bytes)**—Specifies the number of bytes to use for the mailbox message buffer. The Nios II driver software provided by Altera uses eight bytes of overhead to implement the mailbox functionality. For a mailbox capable of passing only one message at a time, **Mailbox size (bytes)** must be at least 12 bytes.
   - **Maximum available bytes**—Specifies the number of bytes in the selected memory available for use as the mailbox message buffer. This field is always read-only.
4. If not already connected, make component connections on the SOPC Builder **System Contents** tab.
   a. Connect each processor’s data bus master port to the mailbox slave port.
   b. Connect each processor’s data bus master port to the shared mailbox memory.

Software Programming Model

The following sections describe the software programming model for the mailbox core. For Nios II processor users, Altera provides routines to access the mailbox core hardware. These functions are specific to the mailbox core and directly manipulate low-level hardware.

The mailbox software programming model has the following characteristics and assumes there are multiple processors accessing a single mailbox core and a shared memory:

- Each mailbox message is one 32-bit word.
- There is a predefined address range in shared memory dedicated to storing messages. The size of this address range imposes a maximum limit on the number of messages pending.
- The mailbox software implements a message FIFO between processors. Only one processor can write to the mailbox at a time, and only one processor can read from the mailbox at a time, ensuring message integrity.
- The software on both the sending and receiving processors must agree on a protocol for interpreting mailbox messages. Typically, processors treat the message as a pointer to a structure in shared memory.
- The sending processor can post messages in succession, up to the limit imposed by the size of the message address range.
- When messages exist in the mailbox, the receiving processor can read messages. The receiving processor can block until a message appears, or it can poll the mailbox for new messages.
- Reading the message removes the message from the mailbox.

Software Files

Altera provides the following software files accompanying the mailbox core hardware:

- **altera_avalon_mailbox_regs.h**—Defines the core's register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_mailbox.h**—Defines data structures and functions to access the mailbox core hardware.
- **altera_avalon_mailbox.c**—Contains the implementations of the functions to access the mailbox core.

Programming with the Mailbox Core

This section describes the software constructs for manipulating the mailbox core hardware.

The file **altera_avalon_mailbox.h** declares a structure `alt_mailbox_dev` that represents an instance of a mailbox device. It also declares functions for accessing the mailbox hardware structure, listed in Table 27–2. For a complete description of each function, refer to "Mailbox API" on page 27–5.

Table 27–2. Mailbox API Functions

Function Name	Description
altera_avalon_mailbox_close()	Closes the handle to a mailbox.
altera_avalon_mailbox_get()	Returns a message if one is present, but does not block waiting for a message.
altera_avalon_mailbox_open()	Claims a handle to a mailbox, enabling all the other functions to access the mailbox core.
altera_avalon_mailbox_pend()	Blocks waiting for a message to be in the mailbox.
altera_avalon_mailbox_post()	Posts a message to the mailbox.

Example 27–1 demonstrates writing to and reading from a mailbox. For this example, assume that the hardware system has two processors communicating via mailboxes. The system includes two mailbox cores, which provide two-way communication between the processors.

Example 27–1. Writing to and Reading from a Mailbox

```
#include <stdio.h>
#include "altera_avalon_mailbox.h"

int main()
{
  alt_u32 message = 0;
  alt_mailbox_dev* send_dev, recv_dev;
  /* Open the two mailboxes between this processor and another */
  send_dev = altera_avalon_mailbox_open("/dev/mailbox_0");
  recv_dev = altera_avalon_mailbox_open("/dev/mailbox_1");

  while(1)
  {
    /* Send a message to the other processor */
    altera_avalon_mailbox_post(send_dev, message);

    /* Wait for the other processor to send a message back */
    message = altera_avalon_mailbox_pend(recv_dev);
  }
  return 0;
}
```

Mailbox API

This section describes the application programming interface (API) for the mailbox core.

altera_avalon_mailbox_close()

Prototype:	`void altera_avalon_mailbox_close (alt_mailbox_dev* dev);`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mailbox.h**>
Parameters:	`dev`—the mailbox to close.
Returns:	Null.
Description:	`altera_avalon_mailbox_close()` closes the mailbox.

altera_avalon_mailbox_get()

Prototype:	`alt_u32 altera_avalon_mailbox_get (alt_mailbox_dev* dev, int* err);`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mailbox.h**>
Parameters:	`dev`—the mailbox handle. `err`—pointer to an error code that is returned.
Returns:	Returns a message if one is available in the mailbox, otherwise returns 0. The value pointed to by `err` is 0 if the message was read correctly, or `EWOULDBLOCK` if there is no message to read.
Description:	`altera_avalon_mailbox_get()` returns a message if one is present, but does not block waiting for a message.

altera_avalon_mailbox_open()

Prototype:	`alt_mailbox_dev* altera_avalon_mailbox_open (const char* name);`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mailbox.h**>
Parameters:	`name`—the name of the mailbox device to open.
Returns:	Returns a handle to the mailbox, or NULL if this mailbox does not exist.
Description:	`altera_avalon_mailbox_open()` opens a mailbox.

altera_avalon_mailbox_pend()

Prototype:	`alt_u32 altera_avalon_mailbox_pend (alt_mailbox_dev* dev);`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mailbox.h**>
Parameters:	`dev`—the mailbox device to read a message from.
Returns:	Returns the message.
Description:	`altera_avalon_mailbox_pend()` is a blocking routine that waits for a message to appear in the mailbox and then reads it.

altera_avalon_mailbox_post()

Prototype:	`int altera_avalon_mailbox_post (alt_mailbox_dev* dev, alt_u32 msg);`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mailbox.h**>
Parameters:	`dev`—the mailbox device to post a message to. `msg`—the value to post.
Returns:	Returns 0 on success, or `EWOULDBLOCK` if the mailbox is full.
Description:	`altera_avalon_mailbox_post()` posts a message to the mailbox.

Document Revision History

Table 27–3 shows the revision history for this chapter.

Table 27–3. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

© March 2009 Altera Corporation

Section V. Test and Debug Peripherals

This section describes test and debug peripherals provided by Altera for SOPC Builder systems.

This section includes the following chapters:

- Chapter 28, Cyclone III Remote Update Controller Core
- Chapter 29, Performance Counter Core
- Chapter 30, Avalon Streaming Test Pattern Generator and Checker Cores

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

28. Cyclone III Remote Update Controller Core

QII55017-9.0.0

Core Overview

The Cyclone® III Remote Update Controller core provides a method to control the Cyclone III remote update block from SOPC Builder systems. The core allows you to access all features of the ALTREMOTE_UPDATE megafunction through a simple Avalon® Memory-Mapped (Avalon-MM) slave interface. The slave interface allows Avalon-MM master peripherals, such as a Nios® II processor, to communicate with the core simply by reading and writing the registers.

The Cyclone III Remote Update Controller core is a thin Avalon interface layer on top of the ALTREMOTE_UPDATE megafunction. Every function of the core maps directly to a function of the megafunction. Altera recommends that you familiarize yourself with the ALTREMOTE_UPDATE megafunction before using the core.

For more information about the ALTREMOTE_UPDATE megafunction, refer to the *altremote_update Megafunction User Guide*. For more information about remote system upgrade in Cyclone III devices, refer to the *Remote System Upgrade With Cyclone III Devices* chapter in volume 1 of the *Cyclone III Device Handbook*.

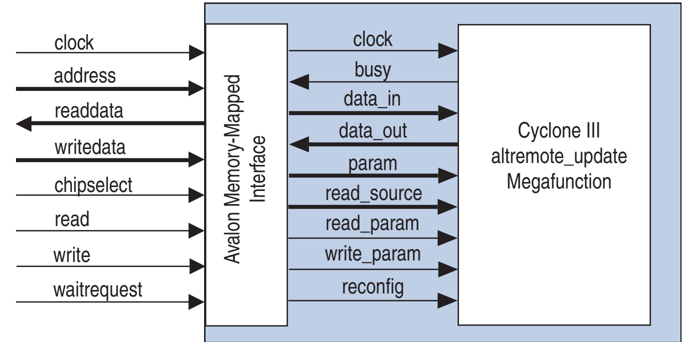
The Cyclone III Remote Update Controller core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 28–2
- "Instantiating the Core in SOPC Builder" on page 28–2

Functional Description

Figure 28–1 shows a block diagram of the Cyclone III Remote Update Controller core.

Figure 28–1. Cyclone III Remote Update Controller Core Block Diagram



Avalon-MM Slave Interface and Registers

The `address` bus on the core's Avalon-MM interface is 6 bits wide. The lower three bits of the `address` bus map directly to the `param` signal of the ALTREMOTE_UPDATE megafunction whereas the upper three bits map to the `read_source` signal.

Reading or writing to address offsets 0x00 – 0x1F of the Cyclone III Remote Update Controller core is equivalent to performing read or write operations to the ALTREMOTE_UPDATE megafunction using the `param` and `read_source` signals.

Table 28–1 shows the mapping of the 5 lowest order Remote Update Controller address bits to the ALTREMOTE_UPDATE megafunction signals.

Table 28–1. Avalon-MM Address Bits to Megafunction Signals Mapping

Address Bit	Megafunction Signal
address[0]	param[0]
address[1]	param[1]
address[2]	param[2]
address[3]	read_source[0]
address[4]	read_source[1]

The highest order address bit [5] is used to access a single `control/status` register. Reading or writing any address offset from 0x20 – 0x3F accesses the `control/status` register.

Table 28–2 shows the bit map of the `control/status` register.

Table 28–2. Bit Map of Control/Status Register

Bit(s)	Field	Access	Description
0	RECONFIG	RW	Set this bit to 1 to reset the FPGA and trigger reconfiguration.
1	RESET_TIMER	RW	Set this bit to 1 to reset the watchdog timer. Then, set this bit to 0 to allow the watchdog timer to continue.
2..31	Reserved		

Device Support

The Cyclone III Remote Update Controller core can only target Cyclone III device family. Both CFI flash and EPCS configuration devices are supported as non-volatile storage for configuration images.

Instantiating the Core in SOPC Builder

The Cyclone III Remote Update Controller core has no user-configurable parameters.

Software Programming Model

Software programs can operate the Cyclone III Remote Update Controller core by reading from and writing to the core's registers.

You can only reconfigure the FPGA to an application image from the factory image. Any attempt to reconfigure from an already reconfigured application image causes the FPGA to return to the factory image.

This section describes the most common types of operations using the Cyclone III Remote Update Controller core.

Setting the Configuration Offset

Before you reconfigure the FPGA, you must first specify the offset within the memory device from which you want to execute a reconfiguration. The offset is the relative address within the memory device where the configuration image is located. Write the offset value to address 0x04 of the core to set the configuration offset.

For example, if your system contains a CFI flash memory mapped at address 0x04000000, and the configuration image is located at address 0x100000 in the flash memory, the offset to set in the Cyclone III Remote Update Controller core is 0x100000.

Shifting the Configuration Offset Value

The ALTREMOTE_UPDATE megafunction requires that you provide only the 22 highest-order bits of a 24-bit address offset. To translate the address, right shift the offset by two bits. This results in a properly oriented 22-bit address offset.

If you are using a CFI flash device, you must also take into account the data width of the flash. If the data width of your flash device is 16 bits, you must provide a 16-bit address offset to the Cyclone III Remote Update Controller core. This requires an additional 1-bit right shift of the byte address offset. No translation is necessary if the data width of your flash is 8 bits.

If you are using an EPCS serial configuration device, consider the data width of the device to be 8 bits. Even though the EPCS device is a serial device, it uses byte addressing internally.

For example, an FPGA is set up to configure itself using active parallel mode from a 16-bit CFI flash memory mapped at address 0x04000000 in an SOPC Builder system, and the configuration image is located at byte offset 0x100000 within the flash memory. To derive the correct configuration offset, you must first right shift the byte offset 0x100000 by one bit to obtain the 16-bit address. Then, right shift by another two bits to obtain the highest 22 bits of the 24-bit offset. The result is a configuration offset of 0x20000 (0x100000 >> 3 = 0x20000), to be written to address 0x04 of the core.

Setting up the Watchdog Timer

You can set up the watchdog timer by writing the upper 12 bits of the 29-bit timeout value to address 0x02 of the core. To reset the watchdog timer, set the `RESET_TIMER` bit of the `control/status` register to 1 and immediately set the bit to 0.

Ensure that you don't accidentally set bit 0 of the `control/status` register to 1. Otherwise, you will trigger a reconfiguration of the FPGA.

For more information on watchdog timer, refer to the *ALTREMOTE_UPDATE Megafunction User Guide*.

If you do not use the watchdog timer feature of the ALTREMOTE_UPDATE megafunction, it must be disabled before a reconfiguration is performed. To disable the watchdog timer, write `0x00` to address 0x03 of the core.

Triggering a Reconfiguration

You can trigger a reconfiguration once you have set the reconfiguration offset in the Cyclone III Remote Update Controller core, and you have either setup or disabled the watchdog timer. To trigger a reconfiguration, set the `RECONFIG` bit in the `control/status` register to 1. Consequently, the FPGA performs a reset and reconfigures itself from the configuration image specified.

Code Example

Example 28–1 shows a C function that can be used to operate the Cyclone III Remote Update Controller core from a processor such as Nios II.

Example 28–1. FPGA Reconfiguration Function

```
/*******************************************************************************
 * Function: CycloneIII_Reconfig
 * Purpose: Uses the ALT_REMOTE_UPDATE megafunction to reconfigure a Cyclone III FPGA.
 * Parameters:
 *    remote_update_base -  base address of the remote update controller
 *    flash_base         -  base address of flash device
 *    reconfig_offset    -  offset in flash from which to reconfigure
 *    watchdog_timeout   -  29-bit watchdog timeout value
 *    width_of_flash     -  data-width of flash device
 * Returns: 0 ( but never exits since it reconfigures the FPGA )
 ****************************************************************************/
int CycloneIII_Reconfig( int remote_update_base,
                         int flash_base,
                         int reconfig_offset,
                         int watchdog_timeout,
                         int width_of_flash )
{int offset_shift;

  // Obtain upper 12 bits of 29-bit watchdog timeout value
  watchdog_timeout = watchdog_timeout >> 17;

  // Only enable the watchdog timer if its timeout value is greater than 0.
  if( watchdog_timeout > 0 )
  {
    // Set the watchdog timeout value
    IOWR( remote_update_base, 0x2, watchdog_timeout );
  }
  else
  {
   // Disable the watchdog timer
   IOWR( remote_update_base, 0x3, 0 );
  }

  // Calculate how much to shift the reconfig offset location:
  //  width_of_flash == 8->offset_shift = 2.
  //  width_of_flash == 16->offset_shift = 3
  offset_shift = (( width_of_flash / 8 ) + 1 );

  // Write the offset of the desired reconfiguration image in flash
  IOWR( remote_update_base, 0x4, reconfig_offset >> offset_shift );

  // Perform the reconfiguration by setting bit 0 in the
  // control/status register
  IOWR( remote_update_base, 0x20, 0x1 );

  return( 0 );
}
```

Related Documentation

This chapter references the following documents:

- *altremote_update Megafunction User Guide*
- *Remote System Upgrade With Cyclone III Devices* in volume 1 of the *Cyclone III Device Handbook*.

Document Revision History

Table 28–3 shows the revision history for this chapter.

Table 28–3. Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Initial release.	—

29. Performance Counter Core

QII55001-9.0.0

Core Overview

The performance counter core with Avalon® interface enables relatively unobtrusive, real-time profiling of software programs. With the performance counter, you can accurately measure execution time taken by multiple sections of code. You need only add a single instruction at the beginning and end of each section to be measured.

The main benefit of using the performance counter core is the accuracy of the profiling results. Alternatives include the following approaches:

- GNU profiler, `gprof`—`gprof` provides broad low-precision timing information about the entire software system. It uses a substantial amount of RAM, and degrades the real-time performance. For many embedded applications, `gprof` distorts real-time behavior too much to be useful.
- Interval timer peripheral—The interval timer is less intrusive than `gprof`. It can provide good results for narrowly targeted sections of code.

The performance counter core is unobtrusive, requiring only a single instruction to start and stop profiling, and no RAM. It is appropriate for high-precision measurements of narrowly targeted sections of code.

For further discussion of all three profiling methods, refer to *AN 391: Profiling Nios II Systems*.

The performance counter core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. The core is designed for use in Avalon-based processor systems, such as a Nios® II processor system. Altera® device drivers enable the Nios II processor to use the performance counters.

This chapter contains the following sections:

- "Functional Description" on page 29–2
- "Device and Tools Support" on page 29–3
- "Instantiating the Core in SOPC Builder" on page 29–3
- "Hardware Simulation Considerations" on page 29–4
- "Software Programming Model" on page 29–4
- "Performance Counter API" on page 29–6

Functional Description

The performance counter core is a set of counters which track clock cycles, timing multiple sections of your software. You can start and stop these counters in your software, individually or as a group. You can read cycle counts from hardware registers.

The core contains two counters for every section:

- Time: A 64-bit clock cycle counter.
- Events: A 32-bit event counter.

Section Counters

Each 64-bit time counter records the aggregate number of clock cycles spent in a section of code. The 32-bit event counter records the number of times the section executes.

The performance counter core can have up to seven section counters.

Global Counter

The global counter controls all section counters. The section counters are enabled only when the global counter is running.

The 64-bit global clock cycle counter tracks the aggregate time for which the counters were enabled. The 32-bit global event counter tracks the number of global events, that is, the number of times the performance counter core has been enabled.

Register Map

The performance counter core has a simple Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to memory-mapped registers. Reading from the registers retrieves the current times and event counts. Writing to the registers starts, stops and resets the counters. Table 29–1 shows the registers in detail.

Table 29–1. Performance Counter Core Register Map (Part 1 of 2)

Offset	Register Name	Bit Description		
		Read	Write	
		31 ... 0	31 ... 1	0
0	$T[0]_{lo}$	global clock cycle counter [31: 0]	*(1)*	0 = STOP 1 = RESET
1	$T[0]_{hi}$	global clock cycle counter [63:32]	*(1)*	0 = START
2	Ev[0]	global event counter	*(1)*	*(1)*
3	—	*(1)*	*(1)*	*(1)*
4	$T[1]_{lo}$	section 1 clock cycle counter [31:0]	*(1)*	0 = STOP
5	$T[1]_{hi}$	section 1 clock cycle counter [63:32]	*(1)*	0 = START
6	Ev[1]	section 1 event counter	*(1)*	*(1)*
7	—	*(1)*	*(1)*	*(1)*
8	$T[2]_{lo}$	section 2 clock cycle counter [31:0]	*(1)*	0 = STOP

Table 29–1. Performance Counter Core Register Map (Part 2 of 2)

Offset	Register Name	Bit Description		
		Read	Write	
		31 ... 0	31 ... 1	0
9	$T[2]_{hi}$	section 2 clock cycle counter [63:32]	*(1)*	0 = START
10	`Ev[2]`	section 2 event counter	*(1)*	*(1)*
11	—	*(1)*	*(1)*	*(1)*
. . .	. . .	. . .	. . .	. . .
4*n* + 0	$T[n]_{lo}$	section *n* clock cycle counter [31:0]	*(1)*	0 = STOP
4*n* + 1	$T[n]_{hi}$	section *n* clock cycle counter [63:32]	*(1)*	0 = START
4*n* + 2	`Ev[n]`	section *n* event counter	*(1)*	*(1)*
4*n* + 3	—	*(1)*	*(1)*	*(1)*

Note to Table 29–1:

(1) Reserved. Read values are undefined. When writing, set reserved bits to zero.

System Reset Considerations

After system reset, the performance counter core is stopped and disabled, and all counters contain zero.

Device and Tools Support

The performance counter core supports all Altera device families supported by SOPC Builder, and provides device drivers for the Nios II hardware abstraction layer (HAL) system library.

Instantiating the Core in SOPC Builder

Use the MegaWizard™ interface for the performance counter core in SOPC Builder to specify the core's hardware features.

Define Counters

Choose the number of section counters you want to generate by selecting from the **Number of simultaneously-measured sections** list. The performance counter core may have up to seven sections. If you require more that seven sections, you can instantiate multiple performance counter cores.

Multiple Clock Domain Considerations

If your SOPC Builder system uses multiple clocks, place the performance counter core in the same clock domain as the CPU. Otherwise, it is not possible to convert cycle counts to seconds correctly.

Hardware Simulation Considerations

You can use this core in simulation with no special considerations.

Software Programming Model

The following sections describe the software programming model for the performance counter core.

Software Files

Altera provides the following software files for Nios II systems. These files define the low-level access to the hardware and provide control and reporting functions. Do not modify these files.

- **altera_avalon_performance_counter.h, altera_avalon_performance_counter.c**—The header and source code for the functions and macros needed to control the performance counter core and retrieve raw results.
- **perf_print_formatted_report.c**—The source code for simple profile reporting.

Using the Performance Counter

In a Nios II system, you can control the performance counter core with a set of highly efficient C macros, and extract the results with C functions.

API Summary

The Nios II application program interface (API) for the performance counter core consists of functions, macros and constants.

Functions and macros

Table 29–2 lists macros and functions for accessing the performance counter hardware structure.

Table 29–2. Performance Counter Macros and Functions

Name	Summary
`PERF_RESET()`	Stops and disables all counters, resetting them to 0.
`PERF_START_MEASURING()`	Starts the global counter and enables section counters.
`PERF_STOP_MEASURING()`	Stops the global counter and disables section counters.
`PERF_BEGIN()`	Starts timing a code section.
`PERF_END()`	Stops timing a code section.
`perf_print_formatted_report()`	Sends a formatted summary of the profiling results to `stdout`.
`perf_get_total_time()`	Returns the aggregate global profiling time in clock cycles.
`perf_get_section_time()`	Returns the aggregate time for one section in clock cycles.
`perf_get_num_starts()`	Returns the number of counter events.
`alt_get_cpu_freq()`	Returns the CPU frequency in Hz.

For a complete description of each macro and function, see "Performance Counter API" on page 29–6.

 © March 2009 Altera Corporation

Hardware Constants

You can get the performance counter hardware parameters from constants defined in **system.h**. The constant names are based on the performance counter instance name, specified on the **System Contents** tab in SOPC Builder. Table 29–3 lists the hardware constants.

Table 29–3. Performance Counter Constants

Name *(1)*	Meaning
PERFORMANCE_COUNTER_BASE	Base address of core
PERFORMANCE_COUNTER_SPAN	Number of hardware registers
PERFORMANCE_COUNTER_HOW_MANY_SECTIONS	Number of section counters

Note to Table 29–3:

(1) Example based on instance name `performance_counter`.

Startup

Before using the performance counter core, invoke `PERF_RESET` to stop, disable and zero all counters.

Global Counter Usage

Use the global counter to enable and disable the entire performance counter core. For example, you might choose to leave profiling disabled until your software has completed its initialization.

Section Counter Usage

To measure a section in your code, surround it with the macros `PERF_BEGIN()` and `PERF_END()`. These macros consist of a single write to the performance counter core.

You can simultaneously measure as many code sections as you like, up to the number specified in SOPC Builder. See “Define Counters” on page 29–3 for details. You can start and stop counters individually, or as a group.

Typically, you assign one counter to each section of code you intend to profile. However, in some situations you may wish to group several sections of code in a single section counter. As an example, to measure general interrupt overhead, you can measure all interrupt service routines (ISRs) with one counter.

To avoid confusion, assign a mnemonic symbol for each section number.

Viewing Counter Values

Library routines allow you to retrieve and analyze the results. Use `perf_print_formatted_report()` to list the results to `stdout`, as shown in Example 29–1.

Example 29–1.

```
perf_print_formatted_report(
     (void *)PERFORMANCE_COUNTER_BASE, // Peripheral's HW base address
     alt_get_cpu_freq(),               // defined in "system.h"
     3,                                // How many sections to print
     "1st checksum_test",              // Display-names of sections
     "pc_overhead",
     "ts_overhead");
```

Example 29–2 creates a table similar to this result.

Example 29–2.

```
--Performance Counter Report--
Total Time: 2.07711 seconds (103855534 clock-cycles)
+-----------------+--------+-----------+--------------+-----------+
| Section         |    %   | Time (sec)| Time (clocks) |Occurrences|
+-----------------+--------+-----------+--------------+-----------+
|1st checksum_test|     50 |   1.03800 |      51899750 |         1 |
+-----------------+--------+-----------+--------------+-----------+
| pc_overhead     |1.73e-05|   0.00000 |            18 |         1 |
+-----------------+--------+-----------+--------------+-----------+
| ts_overhead     |4.24e-05|   0.00000 |            44 |         1 |
+-----------------+--------+-----------+--------------+-----------+
```

For full documentation of `perf_print_formatted_report()`, see "Performance Counter API" on page 29–6.

Interrupt Behavior

The performance counter core does not generate interrupts.

You can start and stop performance counters, and read raw performance results, in an interrupt service routine (ISR). Do not call the `perf_print_formatted_report()` function from an ISR.

If an interrupt occurs during the measurement of a section of code, the time taken by the CPU to process the interrupt and return to the section is added to the measurement time. The same applies to context switches in a multithreaded environment. Your software must take appropriate measures to avoid or handle these situations.

Performance Counter API

This section describes the application programming interface (API) for the performance counter core.

For Nios II processor users, Altera provides routines to access the performance counter core hardware. These functions are specific to the performance counter core and directly manipulate low level hardware. The performance counter core cannot be accessed via the HAL API or the ANSI C standard library.

PERF_RESET()

Prototype:	`PERF_RESET(p)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`p`—performance counter core base address.
Returns:	—
Description:	Macro `PERF_RESET()` stops and disables all counters, resetting them to 0.

PERF_START_MEASURING()

Prototype:	`PERF_START_MEASURING(p)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`p`—performance counter core base address.
Returns:	—
Description:	Macro `PERF_START_MEASURING()` starts the global counter, enabling the performance counter core. The behavior of individual section counters is controlled by `PERF_BEGIN()` and `PERF_END()`. `PERF_START_MEASURING()` defines the start of a global event, and increments the global event counter. This macro is a single write to the performance counter core.

PERF_STOP_MEASURING()

Prototype:	`PERF_STOP_MEASURING(p)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`p`—performance counter core base address.
Returns:	—
Description:	Macro `PERF_STOP_MEASURING()` stops the global counter, disabling the performance counter core. This macro is a single write to the performance counter core.

PERF_BEGIN()

Prototype:	`PERF_BEGIN(p,n)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`p`—performance counter core base address. `n`—counter section number. Section counter numbers start at 1. Do not refer to counter 0 in this macro.

Returns: —

Description: Macro PERF_BEGIN() starts the timer for a code section, defining the beginning of a section event, and incrementing the section event counter. If you subsequently use PERF_STOP_MEASURING() and PERF_START_MEASURING() to disable and re-enable the core, the section counter will resume. This macro is a single write to the performance counter core.

PERF_END()

Prototype: PERF_END(p,n)

Thread-safe: Yes.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: p—performance counter core base address.

n—counter section number. Section counter numbers start at 1. Do not refer to counter 0 in this macro.

Returns: —

Description: Macro PERF_END() stops timing a code section. The section counter does not run, regardless whether the core is enabled or not. This macro is a single write to the performance counter core.

perf_print_formatted_report()

Prototype:
```
int perf_print_formatted_report (
    void* perf_base,
    alt_u32 clock_freq_hertz,
    int num_sections,
    char* section_name_1, ...
    char* section_name_n)
```

Thread-safe: No.

Available from ISR: No.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `perf_base`—Performance counter core base address.

`clock_freq_hertz`—Clock frequency.

`num_sections`—The number of section counters to display. This must not exceed *<instance_name>*_HOW_MANY_SECTIONS.

`section_name_1 ... section_name_n`—The section names to display. The number of section names varies depending on the number of sections to display.

Returns: 0

Description: Function `perf_print_formatted_report()` reads the profiling results from the performance counter core, and prints a formatted summary table.

This function disables all counters. However, for predictable results in a multi-threaded or interrupt environment, invoke `PERF_STOP_MEASURING()` when you reach the end of the code to be measured, rather than relying on `perf_print_formatted_report()`.

This function requires the C standard library. Do not use the small C library with this function.

perf_get_total_time()

Prototype: `alt_u64 perf_get_total_time(void* hw_base_address)`

Thread-safe: No.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `hw_base_address`—base address of performance counter core.

Returns: Aggregate global time in clock cycles.

Description: Function `perf_get_total_time()` reads the raw global time. This is the aggregate time, in clock cycles, that the performance counter core has been enabled. This function has the side effect of stopping the counters.

perf_get_section_time()

Prototype: `alt_u64 perf_get_section_time (void* hw_base_address, int which_section)`

Thread-safe: No.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `hw_base_address`—performance counter core base address.
`which_section`—counter section number.

Returns: Aggregate section time in clock cycles.

Description: Function `perf_get_section_time()` reads the raw time for a given section. This is the time, in clock cycles, that the section has been running. This function has the side effect of stopping the counters.

perf_get_num_starts()

Prototype: `alt_u32 perf_get_num_starts (void* hw_base_address, int which_section)`

Thread-safe: Yes.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters: `hw_base_address`—performance counter core base address.
`which_section`—counter section number.

Returns: Number of counter events.

Description: Function `perf_get_num_starts()` retrieves the number of counter events (or times a counter has been started). If `which_section` = 0, it retrieves the number of global events (times the performance counter core has been enabled). This function does not stop the counters.

alt_get_cpu_freq()

Prototype: `alt_u32 alt_get_cpu_freq()`

Thread-safe: Yes.

Available from ISR: Yes.

Include: **<altera_avalon_performance_counter.h>**

Parameters:

Returns: CPU frequency in Hz.

Description: Function `alt_get_cpu_freq()` returns the CPU frequency in Hz.

Referenced Documents

This chapter references the application note, *AN 391: Profiling Nios II Systems*.

Document Revision History

Table 29–4 shows the revision history for this chapter.

Table 29–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Updated the parameter description of the function `perf_print_formatted_report()`.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

30. Avalon Streaming Test Pattern Generator and Checker Cores

QII55007-9.0.0

Core Overview

The data generation and monitoring solution for Avalon® Streaming (Avalon-ST) consists of two components: a test pattern generator core that generates packetized or non-packetized data and sends it out on an Avalon-ST data interface, and a test pattern checker core that receives the same data and checks it for correctness.

The test pattern generator core can insert different error conditions, and the test pattern checker reports these error conditions to the control interface, each via an Avalon Memory-Mapped (Avalon-MM) slave.

Both cores are SOPC Builder-ready and integrate easily into any SOPC Builder-generated system.

This chapter contains the following sections:

- "Resource Utilization and Performance"
- "Test Pattern Generator" on page 30–3
- "Test Pattern Checker" on page 30–5
- "Device Support" on page 30–6
- "Hardware Simulation Considerations" on page 30–6
- "Software Programming Model" on page 30–7
- "Test Pattern Generator API" on page 30–12
- "Test Pattern Checker API" on page 30–16

Resource Utilization and Performance

Resource utilization and performance for the test pattern generator and checker cores depend on the data width, number of channels, and whether the streaming data uses the optional packet protocol.

Table 30–1 provides estimated resource utilization and performance for the test pattern generator core.

Table 30–1. Test Pattern Generator Estimated Resource Utilization and Performance

No. of Channels	Datawidth (No. of 8-bit Symbols Per Beat)	Packet Support	Stratix® II and Stratix II GX			Cyclone® II			Stratix		
			f_{MAX} (MHz)	ALM Count	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)
1	4	Yes	284	233	560	206	642	560	202	642	560
1	4	No	293	222	496	207	572	496	245	561	496
32	4	Yes	276	270	912	210	683	912	197	707	912
32	4	No	323	227	848	234	585	848	220	630	848
1	16	Yes	298	361	560	228	867	560	245	896	560
1	16	No	340	330	496	230	810	496	228	845	496
32	16	Yes	295	410	912	209	954	912	224	956	912
32	16	No	269	409	848	219	842	848	204	912	848

Table 30–2 provides estimated resource utilization and performance for the test pattern checker core.

Table 30–2. Test Pattern Checker Estimated Resource Utilization and Performance

No. of Channels	Datawidth (No. of 8-bit Symbols Per Beat)	Packet Support	Stratix II and Stratix II GX			Cyclone II			Stratix		
			f_{MAX} (MHz)	ALM Count	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)
1	4	Yes	270	271	96	179	940	0	174	744	96
1	4	No	371	187	32	227	628	0	229	663	32
32	4	Yes	185	396	3616	111	875	3854	105	795	3616
32	4	No	221	363	3520	133	686	3520	133	660	3520
1	16	Yes	253	462	96	185	1433	0	166	1323	96
1	16	No	277	306	32	218	1044	0	192	1004	32
32	16	Yes	182	582	3616	111	1367	3584	110	1298	3616
32	16	No	218	473	3520	129	1143	3520	126	1074	3520

© March 2009 Altera Corporation

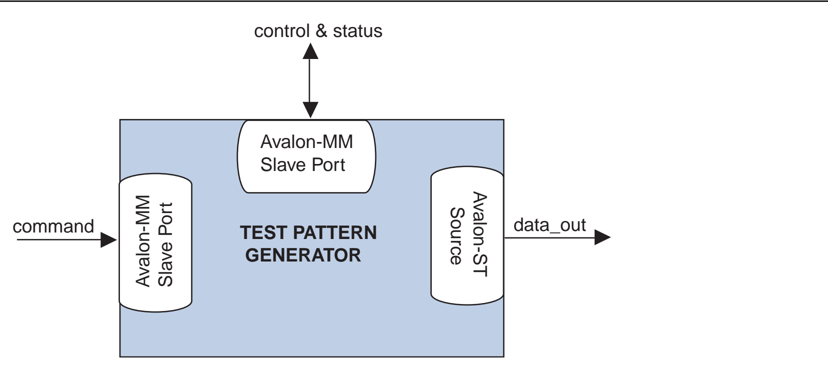
Test Pattern Generator

This section describes the hardware structure and functionality of the test pattern generator core.

Functional Description

The test pattern generator core accepts commands to generate data via an Avalon-MM command interface, and drives the generated data to an Avalon-ST data interface. You can parameterize most aspects of the Avalon-ST data interface such as the number of error bits and data signal width, thus allowing you to test components with different interfaces. Figure 30–1 shows a block diagram of the test pattern generator core.

Figure 30–1. Test Pattern Generator Core Block Diagram



The data pattern is determined by the following equation:
Symbol Value = Symbol Position in Packet XOR Data Error Mask. Non-packetized data is one long stream with no beginning or end.

The test pattern generator core has a throttle register that is set via the Avalon-MM control interface. The value of the throttle register is used in conjunction with a pseudo-random number generator to throttle the data generation rate.

Command Interface

The command interface is a 32-bit Avalon-MM write slave that accepts data generation commands. It is connected to a 16-element deep FIFO, thus allowing a master peripheral to drive a number of commands into the test pattern generator core.

The command interface maps to the following registers: `cmd_lo` and `cmd_hi`. The command is pushed into the FIFO when the register `cmd_lo` (address 0) is written to. When the FIFO is full, the command interface asserts the `waitrequest` signal. You can create errors by writing to the register `cmd_hi` (address 1). The errors are only cleared when 0 is written to this register or its respective fields. See page "Test Pattern Generator Command Registers" on page 30–9 for more information on the register fields.

Control and Status Interface

The control and status interface is a 32-bit Avalon-MM slave that allows you to enable or disable the data generation as well as set the throttle.

This interface also provides useful generation-time information such as the number of channels and whether or not packets are supported.

Output Interface

The output interface is an Avalon-ST interface that optionally supports packets. You can configure the output interface to suit your requirements.

Depending on the incoming stream of commands, the output data may contain interleaved packet fragments for different channels. To keep track of the current symbol's position within each packet, the test pattern generator core maintains an internal state for each channel.

Instantiating the Test Pattern Generator in SOPC Builder

Use the MegaWizard™ interface for the test pattern generator core in SOPC Builder to configure the core. The following sections list the available options in the MegaWizard interface.

Functional Parameter

The functional parameter allows you to configure the test pattern generator as a whole: **Throttle Seed**—The starting value for the throttle control random number generator. Altera recommends a value which is unique to each instance of the test pattern generator and checker cores in a system.

Output Interface

You can configure the output interface of the test pattern generator core using the following parameters:

- **Number of Channels**—The number of channels that the test pattern generator core supports. Valid values are 1 to 256.
- **Data Bits Per Symbol**—The number of bits per symbol for the input and output interfaces. Valid values are 1 to 256. Example—For typical systems that carry 8-bit bytes, set this parameter to 8.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat. Valid values are 1 to 256.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Error Signal Width (bits)**—The width of the `error` signal on the output interface. Valid values are 0 to 31. A value of 0 indicates that the `error` signal is not used.

Test Pattern Checker

This section describes the hardware structure and functionality of the test pattern checker core.

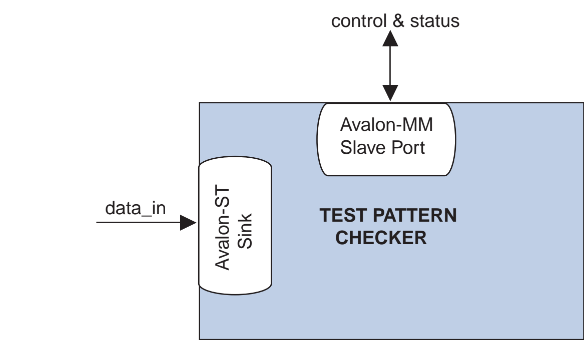
Functional Description

The test pattern checker core accepts data via an Avalon-ST interface, checks it for correctness against the same predetermined pattern used by the test pattern generator core to produce the data, and reports any exceptions to the control interface. You can parameterize most aspects of the test pattern checker's Avalon-ST interface such as the number of error bits and the data signal width, thus allowing you to test components with different interfaces.

The test pattern checker has a throttle register that is set via the Avalon-MM control interface. The value of the throttle register controls the rate at which data is accepted.

Figure 30–2 shows a block diagram of the test pattern checker core.

Figure 30–2. Test Pattern Checker



The test pattern checker core detects exceptions and reports them to the control interface via a 32-element deep internal FIFO. Possible exceptions are data error, missing start-of-packet (SOP), missing end-of-packet (EOP) and signalled error.

As each exception occurs, an exception descriptor is pushed into the FIFO. If the same exception occurs more than once consecutively, only one exception descriptor is pushed into the FIFO. All exceptions are ignored when the FIFO is full. Exception descriptors are deleted from the FIFO after they are read by the control and status interface.

Input Interface

The input interface is an Avalon-ST interface that optionally supports packets. You can configure the input interface to suit your requirements.

Incoming data may contain interleaved packet fragments. To keep track of the current symbol's position, the test pattern checker core maintains an internal state for each channel.

Control and Status Interface

The control and status interface is a 32-bit Avalon-MM slave that allows you to enable or disable data acceptance as well as set the throttle. This interface provides useful generation-time information such as the number of channels and whether the test pattern checker supports packets.

The control and status interface also provides information on the exceptions detected by the test pattern checker core. The interface obtains this information by reading from the exception FIFO.

Instantiating the Test Pattern Checker in SOPC Builder

Use the MegaWizard interface for the test pattern checker core in SOPC Builder to configure the core. The following sections list the available options in the MegaWizard interface.

Functional Parameter

The functional parameter allows you to configure the test pattern checker as a whole: **Throttle Seed**—The starting value for the throttle control random number generator. Altera recommends a unique value to each instance of the test pattern generator and checker cores in a system.

Input Parameters

You can configure the input interface of the test pattern checker core using the following parameters:

- **Data Bits Per Symbol**—The number of bits per symbol for the input interface. Valid values are 1 to 256.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat. Valid values are 1 to 32.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Number of Channels**—The number of channels that the test pattern checker core supports. Valid values are 1 to 256.
- **Error Signal Width (bits)**—The width of the `error` signal on the input interface. Valid values are 0 to 31. A value of 0 indicates that the `error` signal is not in use.

Device Support

The test pattern generator and checker cores support all Altera© device families.

Hardware Simulation Considerations

The test pattern generator and checker cores do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard SOPC Builder simulation flow to simulate the component design files inside an SOPC Builder system.

Software Programming Model

This section describes the software programming model for the test pattern generator and checker cores.

HAL System Library Support

For Nios II processor users, Altera provides HAL system library drivers that enable you to initialize and access the test pattern generator and checker cores. Altera recommends you to use the provided drivers to access the cores instead of accessing the registers directly.

For Nios II IDE users, copy the provided drivers from the following installation folders to your software application directory:

- *<IP installation directory>* **/ip /sopc_builder_ip /altera_avalon_data_source/HAL**
- *<IP installation directory>***/ip/sopc_builder_ip/ altera_avalon_data_sink/HAL**

This instruction does not apply if you use the Nios II command-line tools.

Software Files

The following software files define the low-level access to the hardware, and provide the routines for the HAL device drivers. Application developers should not modify these files.

- Software files provided with the test pattern generator core:
  - **data_source_regs.h**—The header file that defines the test pattern generator's register maps.
  - **data_source_util.h, data_source_util.c**—The header and source code for the functions and variables required to integrate the driver into the HAL system library.
- Software files provided with the test pattern checker core:
  - **data_sink_regs.h**—The header file that defines the core's register maps.
  - **data_sink_util.h, data_sink_util.c**—The header and source code for the functions and variables required to integrate the driver into the HAL system library.

Register Maps

This section describes the register maps for the test pattern generator and checker cores.

Test Pattern Generator Control and Status Registers

Table 30–3 shows the offset for the test pattern generator control and status registers. Each register is 32 bits wide.

Table 30–3. Test Pattern Generator Control and Status Register Map

Offset	Register Name
base + 0	status
base + 1	control
base + 2	fill

Table 30–4 describes the status register bits.

Table 30–4. Status Field Descriptions

Bit(s)	Name	Access	Description
[15:0]	ID	RO	A constant value of 0x64.
[23:16]	NUMCHANNELS	RO	The configured number of channels.
[30:24]	NUMSYMBOLS	RO	The configured number of symbols per beat.
[31]	SUPPORTPACKETS	RO	A value of 1 indicates packet support.

Table 30–5 describes the control register bits

Table 30–5. Control Field Descriptions

Bit(s)	Name	Access	Description
[0]	ENABLE	RW	Setting this bit to 1 enables the test pattern generator core.
[7:1]	Reserved		
[16:8]	THROTTLE	RW	Specifies the throttle value which can be between 0–256, inclusively. This value is used in conjunction with a pseudorandom number generator to throttle the data generation rate. Setting THROTTLE to 0 stops the test pattern generator core. Setting it to 256 causes the test pattern generator core to run at full throttle. Values between 0–256 result in a data rate proportional to the throttle value.
[17]	SOFT RESET	RW	When this bit is set to 1, all internal counters and statistics are reset. Write 0 to this bit to exit reset.
[31:18]	Reserved		

Table 30–6 describes the fill register bits.

Table 30–6. Fill Field Descriptions (Part 1 of 2)

Bit(s)	Name	Access	Description
[0]	BUSY	RO	A value of 1 indicates that data transmission is in progress, or that there is at least one command in the command queue.
[6:1]	Reserved		

Table 30–6. Fill Field Descriptions (Part 2 of 2)

Bit(s)	Name	Access	Description
[15:7]	FILL	RO	The number of commands currently in the command FIFO.
[31:16]	Reserved		

Test Pattern Generator Command Registers

Table 30–7 shows the offset for the command registers. Each register is 32 bits wide.

Table 30–7. Test Pattern Command Register Map

Offset	Register Name
base + 0	cmd_lo
base + 1	cmd_hi

Table 30–8 describes the cmd_lo register bits. The command is pushed into the FIFO only when the cmd_lo register is written to.

Table 30–8. cmd_lo Field Descriptions

Bit(s)	Name	Access	Description
[15:0]	SIZE	RW	The segment size in symbols. Except for the last segment in a packet, the size of all segments must be a multiple of the configured number of symbols per beat. If this condition is not met, the test pattern generator core inserts additional symbols to the segment to ensure the condition is fulfilled.
[29:16]	CHANNEL	RW	The channel to send the segment on. If the channel signal is less than 14 bits wide, the low order bits of this register are used to drive the signal.
[30]	SOP	RW	Set this bit to 1 when sending the first segment in a packet. This bit is ignored when packets are not supported.
[31]	EOP	RW	Set this bit to 1 when sending the last segment in a packet. This bit is ignored when packets are not supported.

Table 30–9 describes the cmd_hi register bits.

Table 30–9. cmd_hi Field Descriptions

Bit(s)	Name	Access	Description
[15:0]	SIGNALLED ERROR	RW	Specifies the value to drive the error signal. A non-zero value creates a signalled error.
[23:16]	DATA ERROR	RW	The output data is XORed with the contents of this register to create data errors. To stop creating data errors, set this register to 0.
[24]	SUPRESS SOP	RW	Set this bit to 1 to suppress the assertion of the startofpacket signal when the first segment in a packet is sent.
[25]	SUPRESS EOP	RW	Set this bit to 1 to suppress the assertion of the endofpacket signal when the last segment in a packet is sent.

Test Pattern Checker Control and Status Registers

Table 30–10 shows the offset for the control and status registers. Each register is 32 bits wide.

Table 30–10. Test Pattern Checker Control and Status Register Map

Offset	Register Name
base + 0	`status`
base + 1	`control`
base + 2	Reserved
base + 3	
base + 4	
base + 5	`exception_descriptor`
base + 6	`indirect_select`
base + 7	`indirect_count`

Table 30–11 describes the `status` register bits.

Table 30–11. Status Field Descriptions

Bit(s)	Name	Access	Description
[15:0]	`ID`	RO	Contains a constant value of `0x65`.
[23:16]	`NUMCHANNELS`	RO	The configured number of channels.
[30:24]	`NUMSYMBOLS`	RO	The configured number of symbols per beat.
[31]	`SUPPORTPACKETS`	RO	A value of 1 indicates packet support.

Table 30–12 describes the `control` register bits.

Table 30–12. Control Field Descriptions

Bit(s)	Name	Access	Description
[0]	`ENABLE`	RW	Setting this bit to 1 enables the test pattern checker.
[7:1]	Reserved		
[16:8]	`THROTTLE`	RW	Specifies the throttle value which can be between 0–256, inclusively. This value is used in conjunction with a pseudorandom number generator to throttle the data generation rate. Setting `THROTTLE` to 0 stops the test pattern generator core. Setting it to 256 causes the test pattern generator core to run at full throttle. Values between 0–256 result in a data rate proportional to the throttle value.
[17]	`SOFT RESET`	RW	When this bit is set to 1, all internal counters and statistics are reset. Write 0 to this bit to exit reset.
[31:18]	Reserved		

Table 30–13 describes the `exception_descriptor` register bits. If there is no exception, reading this register returns 0.

Table 30–13. exception_descriptor Field Descriptions

Bit(s)	Name	Access	Description
[0]	DATA ERROR	RO	A value of 1 indicates that an error is detected in the incoming data.
[1]	MISSINGSOP	RO	A value of 1 indicates missing start-of-packet.
[2]	MISSINGEOP	RO	A value of 1 indicates missing end-of-packet.
[7:3]	Reserved		
[15:8]	SIGNALLED ERROR	RO	The value of the `error` signal.
[23:16]	Reserved		
[31:24]	CHANNEL	RO	The channel on which the exception was detected.

Table 30–14 describes the `indirect_select` register bits.

Table 30–14. indirect_select Field Descriptions

Bit	Bits Name	Access	Description
[7:0]	INDIRECT CHANNEL	RW	Specifies the channel number that applies to the INDIRECT PACKET COUNT, INDIRECT SYMBOL COUNT, and INDIRECT ERROR COUNT registers.
[15:8]	Reserved		
[31:16]	INDIRECT ERROR	RO	The number of data errors that occurred on the channel specified by INDIRECT CHANNEL.

Table 30–15 describes the `indirect_count` register bits.

Table 30–15. indirect_count Field Descriptions

Bit	Bits Name	Access	Description
[15:0]	INDIRECT PACKET COUNT	RO	The number of packets received on the channel specified by INDIRECT CHANNEL.
[31:16]	INDIRECT SYMBOL COUNT	RO	The number of symbols received on the channel specified by INDIRECT CHANNEL.

Test Pattern Generator API

This section describes the application programming interface (API) for the test pattern generator core. All API functions are currently not available from the interrupt service routine (ISR).

data_source_reset()

Prototype: `void data_source_reset(alt_u32 base);`

Thread-safe: No.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: `void`.

Description: This function resets the test pattern generator core including all internal counters and FIFOs. The control and status registers are not reset by this function.

data_source_init()

Prototype: `int data_source_init(alt_u32 base, alt_u32 command_base);`

Thread-safe: No.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

`command_base`—The base address of the command slave.

Returns: 1—Initialization is successful.

0—Initialization is unsuccessful.

Description: This function performs the following operations to initialize the test pattern generator core:

- Resets and disables the test pattern generator core.
- Sets the maximum throttle.
- Clears all inserted errors.

data_source_get_id()

Prototype: `int data_source_get_id(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The test pattern generator core’s identifier.

Description: This function retrieves the test pattern generator core’s identifier.

data_source_get_supports_packets()

Prototype:	`int data_source_init(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status slave.
Returns:	1—Packets are supported. 0—Packets are not supported.
Description:	This function checks if the test pattern generator core supports packets.

data_source_get_num_channels()

Prototype:	`int data_source_get_num_channels(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status slave.
Returns:	The number of channels supported.
Description:	This function retrieves the number of channels supported by the test pattern generator core.

data_source_get_symbols_per_cycle()

Prototype:	`int data_source_get_symbols(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status slave.
Returns:	The number of symbols transferred in a beat.
Description:	This function retrieves the number of symbols transferred by the test pattern generator core in each beat.

data_source_set_enable()

Prototype:	`void data_source_set_enable(alt_u32 base, alt_u32 value);`
Thread-safe:	No.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status slave. `value`—The `ENABLE` bit is set to the value of this parameter.
Returns:	`void.`
Description:	This function enables or disables the test pattern generator core. When disabled, the test pattern generator core stops data transmission but continues to accept commands and stores them in the FIFO.

data_source_get_enable()

Prototype:	`int data_source_get_enable(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status slave.
Returns:	The value of the `ENABLE` bit.
Description:	This function retrieves the value of the `ENABLE` bit.

data_source_set_throttle()

Prototype:	`void data_source_set_throttle(alt_u32 base, alt_u32 value);`
Thread-safe:	No.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status slave. `value`—The throttle value.
Returns:	`void.`
Description:	This function sets the throttle value, which can be between 0–256 inclusively. The throttle value, when divided by 256 yields the rate at which the test pattern generator sends data.

data_source_get_throttle()

Prototype:	`int data_source_get_throttle(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status slave.
Returns:	The throttle value.
Description:	This function retrieves the current throttle value.

data_source_is_busy()

Prototype:	`int data_source_is_busy(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status slave.
Returns:	1—The test pattern generator core is busy. 0—The core is not busy.
Description:	This function checks if the test pattern generator is busy. The test pattern generator core is busy when it is sending data or has data in the command FIFO to be sent.

data_source_fill_level()

Prototype: `int data_source_fill_level(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_source_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The number of commands in the command FIFO.

Description: This function retrieves the number of commands currently in the command FIFO.

data_source_send_data()

Prototype: `int data_source_send_data(alt_u32 cmd_base, alt_u32 channel, alt_u32 size, alt_u32 flags, alt_u32 error, alt_u32 data_error_mask);`

Thread-safe: No.

Include: **<data_source_util.h>**

Parameters: `cmd_base`—The base address of the command slave.

`channel`—The channel to send the data on.

`size`—The data size.

`flags`**—**Specifies whether to send or suppress SOP and EOP signals. Valid values are `DATA_SOURCE_SEND_SOP`, `DATA_SOURCE_SEND_EOP`, `DATA_SOURCE_SEND_SUPRESS_SOP` and `DATA_SOURCE_SEND_SUPRESS_EOP`.

`error`—The value asserted on the `error` signal on the output interface.

`data_error_mask`—This parameter and the data are `XOR`ed together to produce erroneous data.

Returns: Always returns 1.

Description: This function sends a data fragment to the specified channel.

If packets are supported, user applications must ensure the following conditions are met:

SOP and EOP are used consistently in each channel.

Except for the last segment in a packet, the length of each segment is a multiple of the data width.

If packets are not supported, user applications must ensure the following conditions are met:

No SOP and EOP indicators in the data.

The length of each segment in a packet is a multiple of the data width.

Test Pattern Checker API

This section describes the API for the test pattern checker core. The API functions are currently not available from the ISR.

data_sink_reset()

Prototype: `void data_sink_reset(alt_u32 base);`

Thread-safe: No.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: `void`.

Description: This function resets the test pattern checker core including all internal counters.

data_sink_init()

Prototype: `int data_source_init(alt_u32 base);`

Thread-safe: No.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: 1—Initialization is successful.

0—Initialization is unsuccessful.

Description: This function performs the following operations to initialize the test pattern checker core:

- Resets and disables the test pattern checker core.
- Sets the throttle to the maximum value.

data_sink_get_id()

Prototype: `int data_sink_get_id(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The test pattern checker core's identifier.

Description: This function retrieves the test pattern checker core's identifier.

data_sink_get_supports_packets()

Prototype: `int data_sink_init(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: 1—Packets are supported.

0—Packets are not supported.

Description: This function checks if the test pattern checker core supports packets.

data_sink_get_num_channels()

Prototype:	`int data_sink_get_num_channels(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status slave.
Returns:	The number of channels supported.
Description:	This function retrieves the number of channels supported by the test pattern checker core.

data_sink_get_symbols_per_cycle()

Prototype:	`int data_sink_get_symbols(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status slave.
Returns:	The number of symbols received in a beat.
Description:	This function retrieves the number of symbols received by the test pattern checker core in each beat.

data_sink_set enable()

Prototype:	`void data_sink_set_enable(alt_u32 base, alt_u32 value);`
Thread-safe:	No.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status slave. `value`—The `ENABLE` bit is set to the value of this parameter.
Returns:	`void.`
Description:	This function enables the test pattern checker core.

data_sink_get_enable()

Prototype:	`int data_sink_get_enable(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status slave.
Returns:	The value of the `ENABLE` bit.
Description:	This function retrieves the value of the `ENABLE` bit.

data_sink_set_throttle()

Prototype: `void data_sink_set_throttle(alt_u32 base, alt_u32 value);`

Thread-safe: No.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.
`value`—The throttle value.

Returns: `void.`

Description: This function sets the throttle value, which can be between 0–256 inclusively. The throttle value, when divided by 256 yields the rate at which the test pattern checker receives data.

data_sink_get_throttle()

Prototype: `int data_sink_get_throttle(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The throttle value.

Description: This function retrieves the throttle value.

data_sink_get_packet_count()

Prototype: `int data_sink_get_packet_count(alt_u32 base, alt_u32 channel);`

Thread-safe: No.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.
`channel`—Channel number.

Returns: The number of packets received on the given channel.

Description: This function retrieves the number of packets received on a given channel.

data_sink_get_symbol_count()

Prototype: `int data_sink_get_symbol_count(alt_u32 base, alt_u32 channel);`

Thread-safe: No.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.
`channel`—Channel number.

Returns: The number of symbols received on the given channel.

Description: This function retrieves the number of symbols received on a given channel.

data_sink_get_error_count()

Prototype: `int data_sink_get_error_count(alt_u32 base, alt_u32 channel);`

Thread-safe: No.

Include: <**data_sink_util.h**>

Parameters: `base`—The base address of the control and status slave.
`channel`—Channel number.

Returns: The number of errors received on the given channel.

Description: This function retrieves the number of errors received on a given channel.

data_sink_get_exception()

Prototype: `int data_sink_get_exception(alt_u32 base);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `base`—The base address of the control and status slave.

Returns: The first exception descriptor in the exception FIFO.
0—No exception descriptor found in the exception FIFO.

Description: This function retrieves the first exception descriptor in the exception FIFO and pops it off the FIFO.

data_sink_exception_is_exception()

Prototype: `int data_sink_exception_is_exception(int exception);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `exception`—Exception descriptor

Returns: 1—Indicates an exception.
0—No exception.

Description: This function checks if a given exception descriptor describes a valid exception.

data_sink_exception_has_data_error()

Prototype: `int data_sink_exception_has_data_error(int exception);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `exception`—Exception descriptor.

Returns: 1—Data has errors.
0—No errors.

Description: This function checks if a given exception indicates erroneous data.

data_sink_exception_has_missing_sop()

Prototype: `int data_sink_exception_has_missing_sop(int exception);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `exception`—Exception descriptor.

Returns: 1—Missing SOP.

0—Other exception types.

Description: This function checks if a given exception descriptor indicates missing SOP.

data_sink_exception_has_missing_eop()

Prototype: `int data_sink_exception_has_missing_eop(int exception);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `exception`—Exception descriptor.

Returns: 1—Missing EOP.

0—Other exception types.

Description: This function checks if a given exception descriptor indicates missing EOP.

data_sink_exception_signalled_error()

Prototype: `int data_sink_exception_signalled_error(int exception);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `exception`—Exception descriptor.

Returns: The signalled error value.

Description: This function retrieves the value of the signalled error from the exception.

data_sink_exception_channel()

Prototype: `int data_sink_exception_channel(int exception);`

Thread-safe: Yes.

Include: **<data_sink_util.h>**

Parameters: `exception`—Exception descriptor.

Returns: The channel number on which the given exception occurred.

Description: This function retrieves the channel number on which a given exception occurred.

 © March 2009 Altera Corporation

Document Revision History

Table 30–16 shows the revision history for this chapter.

Table 30–16. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	No change from previous release.	—
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	Updated the section on HAL System Library Support.	Updates made to comply with the Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

Section VI. Clock Control Peripherals

This section describes clock control peripherals provided by Altera for SOPC Builder systems.

This section includes the following chapter:

- Chapter 31, PLL Cores

For information about the revision history for chapters in this section, refer to each individual chapter for that chapter's revision history.

31. PLL Cores

NII53002-9.0.0

Core Overview

The PLL cores, Avalon ALTPLL and PLL, provide a means of accessing the dedicated on-chip PLL circuitry in the Altera® Stratix® and Cyclone® series FPGAs. Both cores are a component wrapper around the Altera ALTPLL megafunction.

The Avalon ALTPLL core is a newer generation of the PLL cores. Altera recommends that you use this new core in your design as the older PLL core will be phased out in the near future.

The core takes an SOPC Builder system clock as its input and generates PLL output clocks locked to that reference clock.

The PLL cores support the following features:

- All PLL features provided by Altera's ALTPLL megafunction. The exact feature set depends on the device family.
- Access to status and control signals via Avalon Memory-Mapped (Avalon-MM) registers or top-level signals on the SOPC Builder system module.

The PLL output clocks are made available in two ways:

- As sources to system-wide clocks in your SOPC Builder system.
- As output signals on your SOPC Builder system module.

For details about the ALTPLL megafunction, refer to the *ALTPLL Megafunction User Guide*.

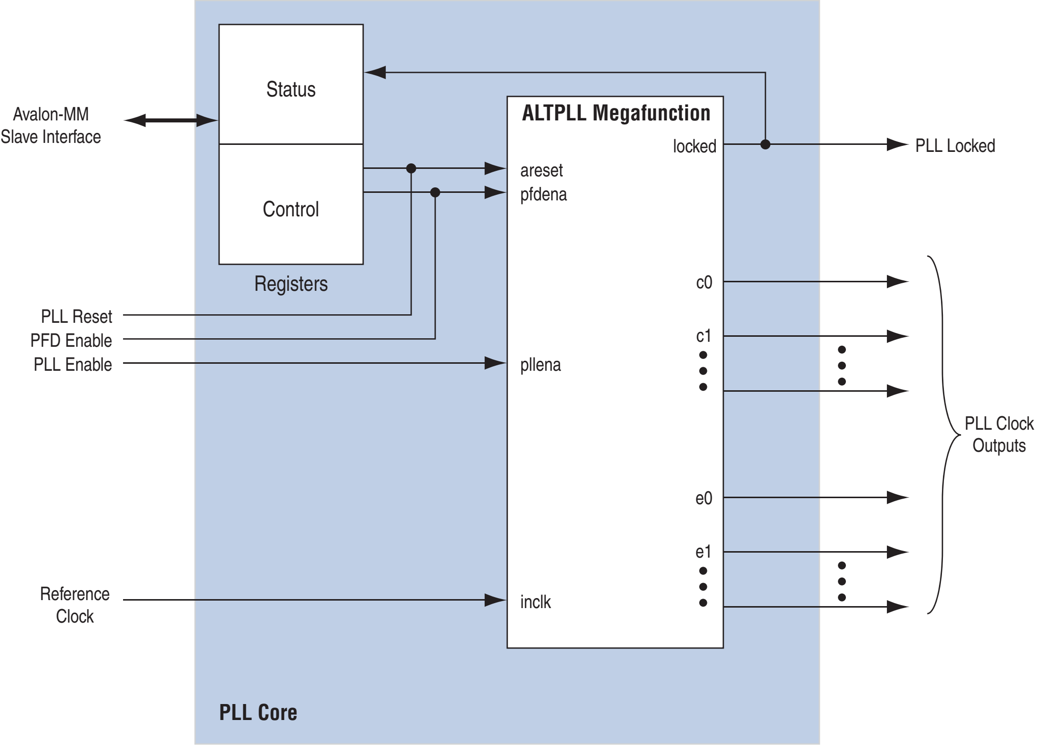
The PLL core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- "Functional Description"
- "Device Support" on page 31–3
- "Instantiating the Cores in SOPC Builder" on page 31–3
- "Hardware Simulation Considerations" on page 31–5
- "Register Definitions and Bit List" on page 31–5

Functional Description

Figure 31–1 shows a block diagram of the PLL cores and their connection to the PLL circuitry inside an Altera FPGA. The following sections describe the components of the core.

Figure 31–1. PLL Core Block Diagram



ALTPLL Megafunction

The PLL cores consist of an ALTPLL megafunction instantiation and an Avalon-MM slave interface. This interface can optionally provide access to status and control registers within the cores. The ALTPLL megafunction takes an SOPC Builder system clock as its reference, and generates one or more phase-locked output clocks.

Clock Outputs

Depending on the target device family, the ALTPLL megafunction can produce two types of output clock:

- internal (c)—clock outputs that can drive logic either inside or outside the SOPC Builder system module. Internal clock outputs can also be mapped to top-level FPGA pins. Internal clock outputs are available on all device families.
- external (e)—clock outputs that can only drive dedicated FPGA pins. They cannot be used as on-chip clock sources. External clock outputs are not available on all device families.

The Avalon ALTPLL core, however, doesn't differentiate the internal and external clock outputs and allows the external clock outputs to be used as on-chip clock sources.

To determine the exact number and type of output clocks available on your target device, refer to the *ALTPLL Megafunction User Guide*.

PLL Status and Control Signals

Depending on how the ALTPLL megafunction is parameterized, there can be a variable number of status and control signals. You can choose to export certain status and control signals to the top-level SOPC Builder system module. Alternatively, Avalon-MM registers can provide access to the signals. Any status or control signals which are not mapped to registers are exported to the top-level module. For details, refer to the "Instantiating the Cores in SOPC Builder" on page 31–3.

System Reset Considerations

At FPGA configuration, the PLL cores reset automatically. PLL-specific reset circuitry guarantees that the PLL locks before releasing reset for the overall SOPC Builder system module.

Resetting the PLL resets the entire SOPC Builder system module.

Device Support

The PLL cores support all Altera device families.

Instantiating the Cores in SOPC Builder

The PLL cores contain an instantiation of the ALTPLL megafunction. The MegaWizard™ interface for the PLL cores allows you to configure the ALTPLL megafunction, and specify connections to selected status and control signals of the megafunction.

For details about using the ALTPLL MegaWizard Plug-In Manager, refer to the *ALTPLL Megafunction User Guide*.

Instantiating the Avalon ALTPLL Core

When you instantiate the Avalon ALTPLL core, the MegaWizard Plug-In Manager is automatically launched for you to parameterize the ALTPLL megafunction. There are no additional parameters that you can configure in SOPC Builder.

The `pfdena` signal of the ALTPLL megafunction is not exported to the top level of the SOPC Builder module. You can drive this port by writing to the `PFDENA` bit in the `control` register.

The `locked`, `pllena/extclkena`, and `areset` signals of the megafunction are always exported to the top level of the SOPC Builder module. You can read the `locked` signal and reset the core by manipulating respective bits in the registers. See "Register Definitions and Bit List" on page 31–5 for more information on the registers.

Instantiating the PLL Core

This section describes the options available in the MegaWizard interface for the PLL core in SOPC Builder.

PLL Settings Page

The **PLL Settings** page contains a button that launches the ALTPLL MegaWizard Plug-In Manager. Use the MegaWizard Plug-In Manager to parameterize the ALTPLL megafunction. The set of available parameters depends on the target device family.

You cannot click **Finish** in the PLL wizard nor configure the PLL interface until you parameterize the ALTPLL megafunction.

Interface Page

The **Interface** page configures the access modes for the optional advanced PLL status and control signals.

For each advanced signal present on the ALTPLL megafunction, you can select one of the following access modes:

- **Export**—Exports the signal to the top level of the SOPC builder system module.
- **Register**—Maps the signal to a bit in a status or control register.

The advanced signals are optional. If you choose not to create any of them in the ALTPLL MegaWizard Plug-In, the PLL's default behavior is as shown in Table 31–1.

You can specify the access mode for the advanced signals shown in Table 31–1. The ALTPLL core signals, not displayed in this table, are automatically exported to the top level of the SOPC Builder system module.

Table 31–1. ALTPLL Advanced Signal

ALTPLL Name	Input / Output	Avalon-MM PLL Wizard Name	Default Behavior	Description
`areset`	input	PLL Reset Input	The PLL is reset only at device configuration.	This signal resets the entire SOPC Builder system module, and restores the PLL to its initial settings.
`pllena`	input	PLL Enable Input	The PLL is enabled.	This signal enables the PLL. `pllena` is always exported.
`pfdena`	input	PFD Enable Input	The phase-frequency detector is enabled.	This signal enables the phase-frequency detector in the PLL, allowing it to lock on to changes in the clock reference.
`locked`	output	PLL Locked Output	—	This signal is asserted when the PLL is locked to the input clock.

Asserting `areset` resets the entire SOPC Builder system module, not just the PLL.

Finish

Click **Finish** to insert the PLL into the SOPC Builder system. The PLL clock output(s) appear in the clock settings table on the SOPC Builder **System Contents** tab.

If the PLL has external output clocks, they appear in the clock settings table like other clocks; however, you cannot use them to drive components within the SOPC Builder system.

For details about using external output clocks, refer to the *ALTPLL Megafunction User Guide*.

The SOPC Builder automatically connects the PLL's reference clock input to the first available clock in the clock settings table.

If there is more than one SOPC Builder system clock available, verify that the PLL is connected to the appropriate reference clock.

Hardware Simulation Considerations

The HDL files generated by SOPC Builder for the PLL cores are suitable for both synthesis and simulation. The PLL cores support the standard SOPC Builder simulation flow, so there are no special considerations for hardware simulation.

Register Definitions and Bit List

Table 31–2 shows the register map for the PLL cores. Device drivers can control and communicate with the cores through two memory-mapped registers, `status` and `control`. The width of these registers are 32 bits in the Avalon ALTPLL core but only 16 bits in the PLL core.

In the PLL core, the `status` and `control` bits shown in Table 31–2 are present only if they have been created in the ALTPLL MegaWizard Plug-In Manager, and set to **Register** on the **Interface** page in the PLL wizard. These registers are always created in the Avalon ALTPLL core.

Table 31–2. PLL Cores Register Map

Offset	Register Name	R/W	Bit Description				
			31/15 *(2)*	...	2	1	0
0	`status`	R/O	*(1)*				`LOCKED`
1	`control`	R/W	*(1)*			`PFDENA`	`ARESET`

Note to Table 31–2:

(1) Reserved. Read values are undefined. When writing, set reserved bits to zero.

(2) The registers are 32-bit wide in the Avalon ALTPLL core and 16-bit wide in the PLL core.

Status Register

Embedded software can access the PLL status via the `status` register. Writing to `status` has no effect. Table 31–3 describes the function of each bit.

Table 31–3. Status Register Bits

Bit Number	Bit Name	Value after reset	Description
0	`LOCKED`	1	Connects to the `locked` signal on the ALTPLL. The `LOCKED` bit is high when valid clocks are present on the output of the PLL.

Table 31–3. Status Register Bits

Bit Number	Bit Name	Value after reset	Description
1 .. 15/31 *(1)*	—	—	Reserved. Read values are undefined.

Note to Table 31–3:

(1) The `status` register is 32-bit wide in the Avalon ALTPLL core and 16-bit wide in the PLL core.

Control Register

Embedded software can control the PLL via the `control` register. Software can also read back the status of control bits. Table 31–4 describes the function of each bit.

Table 31–4. Control Register Bits

Bit Number	Bit Name	Value after reset	Description
[0]	`ARESET`	0	Connects to the `areset` signal on the ALTPLL. Writing a 1 to this bit asserts the `areset` signal for one clock cycle.
[1]	`PFDENA`	1	Connects to the `pfdena` signal on the ALTPLL.
[2:15/31] *(1)*	—	—	Reserved. Read values are undefined. When writing, set reserved bits to zero.

Note to Table 31–3:

(1) The `control` register is 32-bit wide in the Avalon ALTPLL core and 16-bit wide in the PLL core.

Referenced Documents

This chapter references the *ALTPLL Megafunction User Guide*.

Document Revision History

Table 31–5 shows the revision history for this chapter.

Table 31–5. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009	Added information on the new Avalon ALTPLL core.	A new PLL core, Avalon ALTPLL, is released and the chapter is updated accordingly to include the new core.
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	—
May 2008 v8.0.0	No change from previous release.	—

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.

© March 2009 Altera Corporation

About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 9.0.

How to Contact Altera

For the most up-to-date information about Altera products, see the following table.

Contact (Note 1)	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note:

(1) You can also contact your local Altera sales office or sales representative.

Third-Party Software Product Information

Third-party software products described in this handbook are not Altera products, are licensed by Altera from third parties, and are subject to change without notice. Updates to these third-party software products may not be concurrent with Quartus II software releases. Altera has assumed responsibility for the selection of such third-party software products and its use in the Quartus II 9.0 software release. To the extent that the software products described in this handbook are derived from third-party software, no third party warrants the software, assumes any liability regarding use of the software, or undertakes to furnish you any support or information relating to the software. EXCEPT AS EXPRESSLY SET FORTH IN THE APPLICABLE ALTERA PROGRAM LICENSE SUBSCRIPTION AGREEMENT UNDER WHICH THIS SOFTWARE WAS PROVDED TO YOU, ALTERA AND THIRD-PARTY LICENSORS DISCLAIM ALL WARRANTIES WITH RESPECT TO THE USE OF SUCH THIRD-PARTY SOFTWARE CODE OR DOCUMENTATION IN THE SOFTWARE, INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT. For more information, including the latest available version of specific third-party software products, refer to the documentation for the software in question.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, file names, file name extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>*.**pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
`Courier type`	Signal and port names are shown in lowercase Courier type. Examples: `data1`, `tdi`, `input`. Active-low signals are denoted by suffix `n`, e.g., `resetn`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\qdesigns\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
	The hand points to information that requires special attention.
CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information on a particular topic.