Quartus II Handbook Version 10.0

Volume 1: Design and Synthesis

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

QII5V1-10.0.0

Copyright © 2010 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 xix

Section I. Design Flows

Chapter 1. Design Planning with the Quartus II Software

Creating Design Specifications 1-2
Intellectual Property Selection 1-2
System Design 1-3
Device Selection 1-3
Device Migration Planning 1-4
Planning for Device Programming or Configuration 1-5
Early Power Estimation 1-5
Creating Powerplay EPE Spreadsheets 1-6
Early Pin Planning and I/O Analysis 1-6
Simultaneous Switching Noise Analysis 1-8
Selecting Third-Party EDA Tool Flows 1-9
Synthesis Tools 1-9
Simulation Tools 1-9
Formal Verification Tools 1-10
Planning for On-Chip Debugging Options 1-10
Design Practices and HDL Coding Styles 1-11
Design Recommendations 1-11
Recommended HDL Coding Styles 1-12
Managing Metastability 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 1-14
Planning Design Partitions and Floorplan Location Assignments 1-15
Fast Synthesis and Early Timing Estimation 1-16
Conclusion 1-16
Document Revision History 1-17

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

Deciding Whether to Use an Incremental Compilation Flow 2-1
Flat Compilation Flow with No Design Partitions 2-2
Incremental Capabilities Available When Your Design Has No Partitions 2-2
Incremental Compilation Flow with Design Partitions 2-3
Team-Based Design Flows and IP Delivery 2-6
Incremental Compilation Summary 2-7
Steps for Incremental Compilation 2-8
Preparing a Design for Incremental Compilation 2-8
Compiling a Design Using Incremental Compilation 2-8
Common Design Scenarios Using Incremental Compilation 2-8
Reducing Compilation Time When Changing Source Files for One Partition 2-9
Optimizing a Timing-Critical Partition 2-9
Preserving Results While Adding Additional Design Logic 2-10
Debugging Incrementally with the SignalTap II Logic Analyzer 2-11
Deciding which Design Blocks Should Be Design Partitions 2-12

Impact of Design Partitions on Design Optimization 2-14
Design Partition Assignments Compared to Physical Placement Assignments 2-15
Using Partitions with Third-Party Synthesis Tools 2-15
Synopsys Synplify Pro/Premier and Mentor Graphics Precision RTL Plus 2-15
Other Synthesis Tools 2-16
Creating Design Partitions 2-16
Creating Design Partitions in the Project Navigator 2-16
Creating Design Partitions In the Design Partitions Window 2-16
Creating Design Partitions with the Design Partition Planner 2-17
Creating Design Partitions with Tcl Scripting 2-18
Automatically-generated Partitions 2-18
Assessing Partition Quality 2-18
Partition Statistics Reports 2-19
Partition Timing Reports 2-19
Incremental Compilation Advisor 2-20
Specifying the Level of Results Preservation for Subsequent Compilations 2-21
Netlist Type for Design Partitions 2-22
Fitter Preservation Level for Design Partitions 2-23
Where Are the Netlist Databases Saved? 2-24
Deleting Netlists 2-24
What Changes Initiate a Partition's Automatic Resynthesis? 2-25
Resynthesis Due to Source Code Changes 2-26
Forcing Use of the Compilation Netlist When a Partition has Changed 2-27
Exporting Design Partitions from Separate Quartus II Projects 2-27
Preparing the Top-Level Design 2-28
Empty Partitions 2-29
Project Management—Making the Top-Level Design Available to Other Designers 2-29
Distributing the Top-Level Quartus II Project 2-29
Generating Design Partition Scripts 2-31
Exporting Partitions 2-32
Viewing the Contents of a Quartus II Exported Partition File (.qxp) 2-32
Integrating Partitions into the Top-Level Design 2-33
Integrating Assignments from the .qxp File 2-33
Integrating Encrypted IP Cores From .qxp Files 2-34
Advanced Importing Options 2-34
Team-Based Design Optimization and Third-Party IP Delivery Scenarios 2-36
Using an Exported Partition to Send a Design without Including Source Files 2-36
Creating Precompiled Design Blocks (or Hard-Wired Macros) for Reuse 2-37
Designing in a Team-Based Environment 2-39
Enabling Designers on a Team to Optimize Independently 2-40
Performing Design Iterations With Lower-Level Partitions 2-43
Creating a Design Floorplan with LogicLock Regions 2-45
Creating and Manipulating LogicLock Regions 2-46
Changing Partition Placement with LogicLock Changes 2-47
Taking Advantage of the Early Timing Estimator 2-47
Incremental Compilation Restrictions 2-47
When Timing Performance May Not Be Preserved Exactly 2-48
When Placement and Routing May Not Be Preserved Exactly 2-48
Using Incremental Compilation with Quartus II Archive Files 2-48
Limitations for HardCopy Compilation and Migration Flows 2-49
HardCopy ASIC Migration Flows 2-49
HardCopy ASIC Stand-Alone Compilations 2-49
Formal Verification Support 2-50

SignalProbe Pins and Engineering Change Orders . . . 2-50
SignalTap II Logic Analyzer in Exported Partitions . . . 2-50
External Logic Analyzer Interface in Exported Partitions . . . 2-51
Assignments Made in HDL Source Code in Exported Partitions . . . 2-51
Design Partition Script Limitations . . . 2-51
Warnings About Extra Clocks Due to Design Partition Scripts . . . 2-51
Synopsys Design Constraint Files for the TimeQuest Timing Analyzer in Design Partition Scripts . . . 2-51
Wildcard Support in Design Partition Scripts . . . 2-52
Derived Clocks and PLLs in Design Partition Scripts . . . 2-52
Pin Assignments for GXB and LVDS Blocks in Design Partition Scripts . . . 2-52
Virtual Pin Timing Assignments in Design Partition Scripts . . . 2-53
Top-Level Ports that Feed Multiple Lower-Level Pins in Design Partition Scripts . . . 2-53
Restrictions on Megafunction Partitions . . . 2-53
Register Packing and Partition Boundaries . . . 2-54
I/O Register Packing . . . 2-54
Scripting Support . . . 2-55
Preparing a Design for Incremental Compilation . . . 2-55
Creating Design Partitions . . . 2-55
Setting Properties of Design Partitions . . . 2-56
Creating Floorplan Location Assignments—Excluding or Filtering Certain Device Elements (Such as RAM or DSP Blocks) . . . 2-57
Generating Design Partition Scripts . . . 2-57
Command Line Support . . . 2-58
Exporting a Partition . . . 2-58
Importing a Partition into the Top-Level Design . . . 2-59
Makefiles . . . 2-59
Scripting and Command-Line Application Examples . . . 2-60
Reducing Compilation Time When Changing a Source File for One Partition—Command-Line Example . . . 2-60
Optimizing the Placement for a Timing-Critical Partition . . . 2-61
Conclusion . . . 2-61
Document Revision History . . . 2-62

Chapter 3. Quartus II Support for HardCopy Series Devices

HardCopy Series Design Benefits . . . 3-1
Quartus II Features for HardCopy Planning . . . 3-2
HardCopy Development Flow . . . 3-2
Designing the FPGA First . . . 3-3
Designing the HardCopy Device First . . . 3-5
HardCopy Utilities Menu . . . 3-6
Companion Revisions . . . 3-8
Compiling the HardCopy Companion Revision . . . 3-8
Comparing HardCopy and FPGA Companion Revisions . . . 3-9
Generating a HardCopy Handoff Report . . . 3-9
Archiving HardCopy Handoff Files . . . 3-9
HardCopy Advisor . . . 3-10
HardCopy Companion Device Selection . . . 3-10
HardCopy Device Resource Guide . . . 3-11
HardCopy Recommended Settings in the Quartus II Software . . . 3-12
Limit DSP and RAM to HardCopy Device Resources . . . 3-13
Enabling Design Assistant to Run During Compile . . . 3-13
Timing Settings . . . 3-13
Constraints for Clock Effect Characteristics . . . 3-13

Quartus II Software Features Supported for HardCopy Designs . . . 3-15
Physical Synthesis Optimization . . . 3-15
LogicLock Regions . . . 3-15
PowerPlay Power Analyzer . . . 3-15
Incremental Compilation . . . 3-16
HardCopy Design Readiness Check . . . 3-16
Turning the HardCopy Design Readiness Check On and Off . . . 3-16
Setting Check . . . 3-16
Summary . . . 3-16
Global Setting . . . 3-17
Instance Setting . . . 3-17
Operating Setting . . . 3-17
I/O Check . . . 3-17
Input Pin Placement for Global and Regional Clock . . . 3-18
PLL Usage Check . . . 3-18
PLL Real-Time Reconfigurable Check . . . 3-18
PLL Clock Outputs Driving Multiple Clock Network Types Check . . . 3-18
PLL with No Compensation Mode Check . . . 3-18
PLL with Normal or Source Synchronous Mode Feeding Output Pin Check . . . 3-19
RAM Usage Check . . . 3-19
Initialized Memory Dependency Testing . . . 3-19
ALTGX Usage Check . . . 3-20
Performing ECOs with Quartus II Engineering Change Management with the Chip Planner . . . 3-20
Migrating One-to-One Changes . . . 3-21
Migrating Changes that Must be Implemented Differently . . . 3-22
Changes that Cannot be Migrated . . . 3-22
Overall Migration Flow . . . 3-22
Preparing the Revisions . . . 3-22
Applying ECO Changes . . . 3-23
Formal Verification of FPGA and HardCopy Revisions . . . 3-24
HardCopy Floorplan View . . . 3-24
Document Revision History . . . 3-26

Chapter 4. Quartus II Design Separation Flow

Design Flow Overview . . . 4-2
Creating Design Partitions for the Design Separation Flow . . . 4-5
Merging PLL Resources . . . 4-6
Avoiding Child Partitions . . . 4-6
Creating a Design Floorplan with Secured Regions . . . 4-6
Using Security Attributes . . . 4-7
Using Secured Regions . . . 4-9
Adding I/O Pins as Members of Secured Regions . . . 4-10
Using Security Routing Interfaces . . . 4-10
Making Design Separation Flow Location Assignments in the Chip Planner . . . 4-10
Understanding Fencing Regions . . . 4-11
Creating Non-Rectangular Regions . . . 4-14
Guidelines for the Relative Placement of Secure LogicLock Regions . . . 4-15
Creating a Complete Floorplan . . . 4-15
Ensuring Routability Between Regions . . . 4-17
Ensuring Planarity . . . 4-18
Placing Physical Resources . . . 4-20
Making Signal Security Assignments . . . 4-20
Understanding Signal Names . . . 4-21
Working with Global Signals . . . 4-22

Assigning I/O Pins 4-26
Routing Restrictions 4-27
Number of Signals in Routing Interfaces 4-29
Application Example: Modify a Fitter-Generated Floorplan for the Design Separation Flow 4-32
Report Panels 4-35
Secured LogicLock Region Summary 4-35
Security Routing Interfaces 4-35
Secured LogicLock Region Inputs and Outputs 4-36
Security I/O Bank Usage 4-36
Quartus Settings File Syntax 4-36
LL_SECURITY_ROUTING_INTERFACE 4-36
LL_MEMBER_OF_SECURITY_ROUTING_INTERFACE 4-37
LL_SIGNAL_SECURITY_LEVEL 4-37
Document Revision History 4-37

Section II. Design Guidelines

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

Synchronous FPGA Design Practices 5-1
Fundamentals of Synchronous Design 5-2
Hazards of Asynchronous Design 5-3
Design Guidelines 5-3
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
Power Optimization 5-12
Metastability 5-13
Incremental Compilation 5-13
Checking Design Violations With the Design Assistant 5-13
Quartus II Design Flow with the Design Assistant 5-13
Enabling and Disabling Design Assistant Rules 5-15
Viewing Design Assistant Results 5-15
Custom Rules 5-15
XML File Format for Custom Rules 5-16
Specifying the Path to the Custom Rules File 5-17
Custom Rules Coding Examples 5-18
Targeting Clock and Register-Control Architectural Features 5-22
Clock Network Resources 5-22
Reset Resources 5-23
Synchronous Reset 5-23
Asynchronous Reset 5-23
Synchronized Asynchronous Reset 5-24
Register Control Signals 5-26

Targeting Embedded RAM Architectural Features 5-26
Conclusion 5-27
Document Revision History 5-27

Chapter 6. Recommended HDL Coding Styles

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 6-3
Creating a Netlist File for Other Synthesis Tools 6-4
Instantiating Megafunctions Using the Port and Parameter Definition 6-5
Inferring Multiplier and DSP Functions from HDL Code 6-5
Inferring Multipliers from HDL Code 6-5
Inferring Multiply-Accumulators and Multiply-Adders from HDL Code 6-7
Inferring Memory Functions from HDL Code 6-12
Inferring RAM functions from HDL Code 6-13
Use Synchronous Memory Blocks 6-13
Avoid Unsupported Reset and Control Conditions 6-14
Check Read-During-Write Behavior 6-16
Controlling Inference and Implementation in Device RAM Blocks 6-17
Single-Clock Synchronous RAM with Old Data Read-During-Write Behavior 6-18
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
Mixed-Width Dual-Port RAM 6-26
RAM with Byte-Enable Signals 6-29
Specifying Initial Memory Contents at Power-Up 6-31
Inferring ROM Functions from HDL Code 6-33
Shift Registers—Inferring the ALTSHIFT_TAPS Megafunction from HDL Code 6-37
Simple Shift Register 6-38
Shift Register with Evenly Spaced Taps 6-39
Coding Guidelines for Registers and Latches 6-41
Register Power-Up Values in Altera Devices 6-41
Secondary Register Control Signals Such as Clear and Clock Enable 6-43
Latches 6-47
Unintentional Latch Generation 6-47
Inferring Latches Correctly 6-48
General Coding Guidelines 6-51
Tri-State Signals 6-51
Clock Multiplexing 6-52
Adder Trees 6-55
Architectures with 4-Input LUTs in Logic Elements 6-56
Architectures with 6-Input LUTs in Adaptive Logic Modules 6-56
State Machines 6-58
Verilog HDL State Machines 6-59
VHDL State Machines 6-63
Multiplexers 6-65
Quartus II Software Option for Multiplexer Restructuring 6-65
Multiplexer Types 6-66
Implicit Defaults in If Statements 6-68
Default or Others Case Assignment 6-68
Cyclic Redundancy Check Functions 6-68
If Performance is Important, Optimize for Speed 6-69
Use Separate CRC Blocks Instead of Cascaded Stages 6-69

Use Separate CRC Blocks Instead of Allowing Blocks to Merge 6-69
Take Advantage of Latency if Available 6-70
Save Power by Disabling CRC Blocks When Not in Use 6-70
Use the Device Synchronous Load (sload) Signal to Initialize 6-70
Comparators 6-70
Counters 6-71
Designing with Low-Level Primitives 6-72
Conclusion 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
Synchronization Register Chains 7-2
Identifying Synchronizers for Metastability Analysis 7-4
How Timing Constraints Affect Synchronizer Identification and Metastability Analysis 7-4
Metastability and MTBF Reporting 7-5
Metastability Reports 7-5
MTBF Summary Report 7-5
Synchronizer Summary Report 7-6
Synchronizer Chain Statistics Report in the Timing Analyzer 7-7
Synchronizer Data Toggle Rate in MTBF Calculation 7-7
MTBF Optimization 7-8
Synchronization Register Chain Length 7-8
Reducing Metastability Effects 7-9
Apply Complete System-Centric Timing Constraints for the Timing Analyzer 7-9
Force the Identification of Synchronization Registers 7-9
Set the Synchronizer Data Toggle Rate 7-10
Optimize Metastability During Fitting 7-10
Increase the Length of Synchronizers to Protect and Optimize 7-10
Set Fitter Effort to Standard Fit instead of Auto Fit 7-10
Increase the Number of Stages Used in Synchronizers, If Possible 7-10
Select a Faster Speed Grade Device, if Possible 7-11
Scripting Support 7-11
Identifying Synchronizers for Metastability Analysis 7-11
Synchronizer Data Toggle Rate in MTBF Calculation 7-12
report_metastability and Tcl Command 7-12
MTBF Optimization 7-12
Synchronization Register Chain Length 7-12
Conclusion 7-13
Document Revision History 7-13

Chapter 8. Best Practices for Incremental Compilation Partitions and Floorplan Assignments
Overview: Incremental Compilation 8-2
Recommendations for the Netlist Type 8-2
Incremental Design Flows 8-3
Project Management in Team-Based Design Flows 8-4
Why Plan Partitions and Floorplan Assignments for Incremental Compilation? 8-5
Partition Boundaries and Optimization 8-6
General Partitioning Guidelines 8-7
Plan Design Hierarchy and Source Design Files 8-8
Using Partitions with Third-Party Synthesis Tools 8-8

Partition Design by Functionality and Block Size 8-9
Partition Design by Clock Domain and Timing Criticality 8-9
Consider What Is Changing 8-9
Design Partition Guidelines 8-10
Register Partition Inputs and Outputs 8-10
Minimize Cross-Partition-Boundary I/O 8-11
Avoid the Need for Logic Optimization Across Partitions 8-12
Keep Logic in the Same Partition for Optimization and Merging 8-13
Keep Constants in the Same Partition as Logic 8-14
Avoid Unconnected Partition I/O 8-15
Avoid Signals That Drive Multiple Partition I/O or Connect I/O Together 8-16
Invert Clocks in Destination Partitions 8-17
Connect I/O Pin Directly to I/O Register for Packing Across Partition Boundaries 8-17
Do Not Use Internal Tri-States 8-21
Include All Tri-State and Enable Logic in the Same Partition 8-22
Include Bidirectional I/O Registers in the Same Partition (For Older Device Families) 8-23
Summary of Guidelines Related to Logic Optimization Across Partitions 8-23
Consider a Cascaded Reset Structure 8-24
Design Partition Guidelines for Third-Party IP Delivery 8-25
Allocate Logic Resources 8-26
Allocate Global Routing Signals and Clock Networks if Required 8-27
Assign Virtual Pins 8-28
Perform Timing Budgeting if Required 8-29
Drive Clocks Directly 8-29
Recreate PLLs for Lower-Level Partitions if Required 8-30
Checking Partition Quality 8-31
Incremental Compilation Advisor 8-31
Design Partition Planner 8-31
Viewing Design Partition Planner and Floorplan Side-by-Side 8-33
Partition Statistics Report 8-34
Report Partition Timing in the TimeQuest Timing Analyzer 8-35
Check if Partition Assignments Impact the Quality of Results 8-35
Including SDC Constraints from Lower-Level Partitions for Third-Party IP Delivery 8-36
Creating an .sdc with Project-Wide Constraints 8-37
Creating an .sdc with Partition-Specific Constraints 8-38
Consolidating the .sdc in the Top-Level Design 8-39
Introduction to Design Floorplans 8-40
The Difference between Logical Partitions and Physical Regions 8-40
Why Create a Floorplan? 8-41
When to Create a Floorplan 8-42
Early Floorplan 8-42
Late Floorplan 8-43
Design Floorplan Placement Guidelines 8-43
Assigning Partitions to LogicLock Regions 8-44
How to Size and Place Regions 8-44
Modifying Region Size and Origin 8-45
I/O Connections 8-46
LogicLock Resource Exclusions 8-46
Creating Non-Rectangular Regions 8-47
Checking Floorplan Quality 8-48
Incremental Compilation Advisor 8-48
LogicLock Region Resource Estimates 8-48
LogicLock Region Properties Statistics Report 8-48

Locate the Quartus II TimeQuest Timing Analyzer Path in the Chip Planner . . . 8-48
Inter-Region Connection Bundles . . . 8-48
Routing Utilization . . . 8-49
Ensure Floorplan Assignments Do Not Significantly Impact Quality of Results . . . 8-49
Recommended Design Flows and Application Examples . . . 8-49
Create a Floorplan for Major Design Blocks . . . 8-50
Create a Floorplan Assignment for One Design Block with Difficult Timing . . . 8-50
Create a Floorplan as the Project Lead in a Team-Based Flow . . . 8-51
Conclusion . . . 8-52
Revision History . . . 8-53

Section III. Synthesis

Chapter 9. Quartus II Integrated Synthesis

Design Flow . . . 9-1
Language Support . . . 9-3
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-2008 Support . . . 9-10
VHDL Standard Libraries and Packages . . . 9-10
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 . . . 9-20
Quartus II Exported Partition 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 . . . 9-24
Synthesis Attributes . . . 9-25
Synthesis Directives . . . 9-27
Optimization Technique . . . 9-28
Auto Gated Clock Conversion . . . 9-28
Timing-Driven Synthesis . . . 9-30
SDC Constraint Protection . . . 9-31
PowerPlay Power Optimization . . . 9-31
Limiting Resource Usage in Partitions . . . 9-32
Restructure Multiplexers . . . 9-34

Synthesis Effort . . . 9-35
Synthesis Seed . . . 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 . . . 9-38
Safe State Machines . . . 9-39
Power-Up Level . . . 9-41
Inferred Power-Up Levels . . . 9-42
Power-Up Don't Care . . . 9-42
Remove Duplicate Registers . . . 9-42
Preserve Registers . . . 9-43
Disable Register Merging/Don't Merge Register . . . 9-44
Noprune Synthesis Attribute/Preserve Fan-out Free Register Node . . . 9-44
Keep Combinational Node/Implement as Output of Logic Cell . . . 9-45
Disabling Synthesis Netlist Optimizations with dont_retime Attribute . . . 9-46
Disabling Synthesis Netlist Optimizations with dont_replicate Attribute . . . 9-47
Maximum Fan-Out . . . 9-48
Controlling Clock Enable Signals with Auto Clock Enable Replacement and direct_enable . . . 9-49
Inferring Multiplier, DSP, and Memory Functions from HDL Code . . . 9-50
Multiply-Accumulators and Multiply-Adders . . . 9-50
Shift Registers . . . 9-50
RAM and ROM . . . 9-51
Resource Aware RAM, ROM, and Shift-Register Inference . . . 9-52
RAM to Logic Cell Conversion . . . 9-52
RAM Style and ROM Style—for Inferred Memory . . . 9-53
Turning Off the Add Pass-Through Logic to Inferred RAMs no_rw_check Attribute . . . 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 . . . 9-66
Using altera_attribute to Set Quartus II Logic Options . . . 9-67
Analyzing Synthesis Results . . . 9-70
Analysis & Synthesis Section of the Compilation Report . . . 9-70
Project Navigator . . . 9-70
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 . . . 9-75
Node-Naming Conventions for Registers (DFF or D Flipflop Atoms) . . . 9-75
Register Changes During Synthesis . . . 9-76
Synthesis and Fitting Optimizations . . . 9-77
State Machines . . . 9-77
Inferred Adder-Subtractors, Shift Registers, Memory, and DSP Functions . . . 9-78
Packed Input and Output Registers of RAM and DSP Blocks . . . 9-78
Preserving Register Names . . . 9-78
Node-Naming Conventions for Combinational Logic Cells . . . 9-79

Preserving Combinational Logic Names . . . 9-80
Scripting Support . . . 9-80
Adding an HDL File to a Project and Setting the HDL Version . . . 9-82
Assigning a Pin . . . 9-83
Creating Design Partitions for Incremental Compilation . . . 9-84
Quartus II Synthesis Options . . . 9-84
Conclusion . . . 9-87
Document Revision History . . . 9-88

Chapter 10. Synopsys Synplify Support

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
I/O Standard . . . 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 . . . 10-13
Exporting Designs to the Quartus II Software Using NativeLink Integration . . . 10-14
Running the Quartus II Software from within the Synplify Software . . . 10-14
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 . . . 10-17
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
Document Revision History 10-50

Chapter 11. Mentor Graphics Precision Synthesis Support

Altera Device Family Support 11-1
Design Flow 11-2
Creating and Compiling a Project in the Precision Synthesis Software 11-5
Creating a Project 11-5
Compiling the Design 11-5
Mapping the Precision Synthesis Design 11-6
Setting Timing Constraints 11-6
Setting Mapping Constraints 11-7
Assigning Pin Numbers and I/O Settings 11-7
Assigning I/O Registers 11-8
Disabling I/O Pad Insertion 11-9
Preventing the Precision Synthesis Software from Adding I/O Pads 11-9
Preventing the Precision Synthesis Software from Adding an I/O Pad on an Individual Pin . 11-9
Controlling Fan-Out on Data Nets 11-10
Synthesizing the Design and Evaluating the Results 11-10

Obtaining Accurate Logic Utilization and Timing Analysis Reports 11-10
Exporting Designs to the Quartus II Software Using NativeLink Integration 11-11
Running the Quartus II Software from within the Precision Synthesis Software 11-11
Running the Quartus II Software Manually Using the Precision Synthesis-Generated Tcl Script 11-12
Using the Quartus II Software to Run the Precision Synthesis Software 11-12
Passing Constraints to the Quartus II Software 11-13
create_clock 11-13
set_input_delay 11-14
set_output_delay 11-14
set_max_delay 11-15
set_min_delay 11-15
set_false_path 11-16
set_multicycle_path 11-17
Guidelines for Altera Megafunctions and Architecture-Specific Features 11-18
Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager 11-18
Using MegaWizard Plug-In Manager-Generated Verilog HDL Files for Megafunction Instantiation 11-19
Using MegaWizard Plug-In Manager-Generated VHDL Files for Megafunction Instantiation 11-19
Instantiating Intellectual Property Using the MegaWizard Plug-In Manager and IP Toolbench . . . 11-19
Using Generated Verilog HDL Files for Black Box IP Function Instantiation 11-20
Using Generated VHDL Files for Black Box IP Function Instantiation 11-21
Inferring Altera Megafunctions from HDL Code 11-21
Multipliers 11-22
Using the GUI 11-22
Using Attributes 11-22
Multiplier-Accumulators and Multiplier-Adders 11-24
Controlling DSP Block Inference 11-24
RAM and ROM 11-26
Incremental Compilation and Block-Based Design 11-27
Creating a Design with Precision RTL Plus Incremental Synthesis 11-27
Creating Partitions with the incr_partition Attribute 11-28
Creating Multiple EDIF Netlist Files Using Separate Precision Projects or Implementations . . . 11-29
Creating Black Boxes to Create EDIF Netlists 11-31
Creating Black Boxes in Verilog HDL 11-31
Creating Black Boxes in VHDL 11-32
Creating Quartus II Projects for Multiple EDIF Files 11-33
Creating a Single Quartus II Project for a Top-Down Incremental Compilation Flow 11-34
Creating Multiple Quartus II Projects for a Bottom-Up Flow 11-35
Hierarchy and Design Considerations 11-35
Conclusion 11-36
Document Revision History 11-36

Chapter 12. Mentor Graphics LeonardoSpectrum Support

Altera Device Family Support 12-1
Design Flow 12-2
Optimization Strategies 12-4
Timing-Driven Synthesis 12-4
Global PowerTab 12-4
Clock PowerTab 12-5
Input and Output PowerTabs 12-5
Other Constraints 12-5

Encoding Style 12-5
Resource Sharing 12-6
Mapping I/O Registers 12-6
Timing Analysis with the LeonardoSpectrum Software 12-6
Exporting Designs Using NativeLink Integration 12-7
Generating Netlist Files 12-7
Including Design Files for Black Boxed Modules 12-7
Passing Constraints with Scripts 12-8
Integration with the Quartus II Software 12-8
Guidelines for Altera Megafunctions and LPM Functions 12-8
Instantiating Altera Megafunctions 12-9
Inferring Altera Memory Elements 12-9
Inferring Multipliers and DSP Functions 12-10
Simple Multipliers 12-10
Multiplier Accumulators 12-10
Multiplier Adders 12-11
Controlling DSP Block Inference 12-11
Global Attribute 12-12
Module Level Attributes 12-12
Signal Level Attributes 12-13
Guidelines for Using DSP Blocks 12-15
Block-Based Design with the Quartus II Software 12-16
Hierarchy and Design Considerations 12-16
Creating a Design with Multiple .edf Files 12-17
Generating Multiple .edf Files Using the LogicLock Option 12-17
Creating a Quartus II Project for Multiple .edf Files Including LogicLock Regions 12-19
Generating Multiple .edf Files Using Black Boxes 12-20
Black Box Methodology in Verilog HDL 12-22
Black Boxing in VHDL 12-22
Creating a Quartus II Project for Multiple .edf Files 12-24
Incremental Synthesis Flow 12-25
Modifications Required for the LogicLock_Incremental.tcl Script File 12-25
Running the Tcl Script File in LeonardoSpectrum 12-26
Conclusion 12-27
Document Revision History 12-27

Chapter 13. Analyzing Designs with Quartus II Netlist Viewers

When to Use the Netlist Viewers: Analyzing Design Problems 13-1
Quartus II Design Flow with the Netlist Viewers 13-2
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
Netlist Navigator Pane 13-14
State Machine Viewer 13-15
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 13-17

Global Options 13-18
Display Settings 13-18
Tracing 13-19
Customize View 13-20
Shortcut Commands 13-21
Navigating the Schematic View 13-21
Traversing and Viewing the Design Hierarchy 13-21
Flattening the Design Hierarchy 13-21
Viewing the Contents of a Design Hierarchy in the Current Schematic 13-21
Viewing Contents of Atom Primitives 13-22
Viewing the Properties of Instances and Primitives 13-23
Viewing LUT Representations in the Technology Map Viewer 13-23
Grouping Combinational Logic into Logic Clouds 13-25
Logic Clouds in the RTL Viewer 13-25
Logic Clouds in the Technology Map Viewer 13-25
Grouping and Ungrouping Logic Clouds 13-27
Changing the Constant Signal Value Formatting 13-27
Zooming and Magnification 13-27
Schematic Debugging and Tracing Using the Bird's Eye View 13-28
Partitioning the Schematic into Pages 13-29
Moving Between Schematic Pages 13-30
Moving Back and Forward Through Schematic Pages 13-30
Following Nets Across Schematic Pages 13-30
Go to Net Driver 13-31
Filtering in the Schematic View 13-32
Filter Sources Command 13-32
Filter Destinations Command 13-33
Filter Sources and Destinations Command 13-33
Filter Between Selected Nodes Command 13-33
Filter Selected Nodes and Nets Command 13-34
Filter Bus Index Command 13-36
Filter Command Processing 13-36
Filtering Across Hierarchies 13-36
Expanding a Filtered Netlist 13-37
Reducing a Filtered Netlist 13-38
Probing to a Source Design File and Other Quartus II Windows 13-38
Moving Selected Nodes to Other Quartus II Windows 13-39
Probing to the Netlist Viewers from Other Quartus II Windows 13-39
Viewing a Timing Path 13-40
Other Features in the Schematic Viewer 13-42
Tooltips 13-42
Finding Design Elements in the Netlist Viewers 13-44
Exporting and Copying a Schematic Image 13-45
Printing 13-45
Conclusion 13-45
Document Revision History 13-46

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 10.0 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: *July 2010*
Part Number: *QII51016-10.0.0*

Chapter 2 Quartus II Incremental Compilation for Hierarchical and Team-Based Design
Revised: *July 2010*
Part Number: *QII51015-10.0.0*

Chapter 3 Quartus II Support for HardCopy Series Devices
Revised: *July 2010*
Part Number: *QII51004-10.0.0*

Chapter 4 Quartus II Design Separation Flow
Revised: *July 2010*
Part Number: *QII51019-10.0.0*

Chapter 5 Design Recommendations for Altera Devices and the Quartus II Design Assistant
Revised: *July 2010*
Part Number: *QII51006-10.0.0*

Chapter 6 Recommended HDL Coding Styles
Revised: *July 2010*
Part Number: *QII51007-10.0.0*

Chapter 7 Managing Metastability with the Quartus II Software
Revised: *July 2010*
Part Number: *QII51018-10.0.0*

Chapter 8 Best Practices for Incremental Compilation Partitions and Floorplan Assignments
Revised: *July 2010*
Part Number: *QII51017-10.0.0*

Chapter 9 Quartus II Integrated Synthesis
Revised: *July 2010*
Part Number: *QII51008-10.0.0*

Chapter 10 Synopsys Synplify Support
Revised: *July 2010*
Part Number: *QII51009-10.0.0*

Chapter 11 Mentor Graphics Precision Synthesis Support
Revised: *July 2010*
Part Number: *QII51011-10.0.0*

Chapter 12 Mentor Graphics LeonardoSpectrum Support
Revised: *July 2010*
Part Number: *QII51010-10.0.0*

Chapter 13 Analyzing Designs with Quartus II Netlist Viewers
Revised: *July 2010*
Part Number: *QII51013-10.0.0*

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 is an introduction to design planning. It documents various specialized design flows in the following chapters:

- Chapter 1, Design Planning with the Quartus II Software

  This chapter is an overview of various design planning considerations: device selection, early power estimation, I/O pin planning, and design planning. To help you improve design productivity, it provides recommendations and describes various tools available for Altera FPGAs.

- 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.

- Chapter 3, Quartus II Support for HardCopy Series Devices

  With the Quartus II software, you can use 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.

- Chapter 4, Quartus II Design Separation Flow

  This chapter describes rules and guidelines for creating a floorplan with the Design Separation flow. The Quartus II Design Separation flow provides the ability to design physically independent structures on a single device. This allows system designers to achieve a higher level of integration on a single FPGA, and alleviates increasingly strict Size Weight and Power (SWaP) requirements.

1. Design Planning with the Quartus II Software

QII51016-10.0.0

This chapter discusses key FPGA design planning considerations, provides recommendations, and describes various tools available for you to improve your design productivity with Altera® FPGAs. Advanced FPGAs allow you flexibility in how you want to implement pin layout, power consumption, area utilization, and timing performance in each design block. Because of the significant increase in FPGA device densities over the last few years, designs are more complex and can involve multiple designers. System architects must resolve these design issues when integrating design blocks, often leading to problems that affect the overall time to market and thereby increase cost. You can solve many potential problems earlier in the design cycle by following the design planning considerations provided in this chapter.

This chapter contains the following sections:

- "Creating Design Specifications" on page 1–2
- "Intellectual Property Selection" on page 1–2
- "Device Selection" on page 1–3
- "Planning for Device Programming or Configuration" on page 1–5
- "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 1–9
- "Planning for On-Chip Debugging Options" on page 1–10
- "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 1–16

Before reading the design planning guidelines discussed in this chapter, consider your design priorities. You should know the most important priorities for your design. More device features, density, or performance requirements can increase system cost. Signal integrity and board issues might impact I/O pin locations. Power, timing performance, and area utilization affect each other, and compilation time is affected optimizing these priorities.

The Quartus® II software optimizes designs for the best results on average, but you can change the settings to focus on one aspect of your design. Certain tools or debugging options can lead to restrictions in your design flow. Your design priorities help you choose the tools, features, and methodologies that you should use for 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 guidelines to help ensure a productive and successful FPGA design flow.

This chapter only provides an introduction to various design 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 information about specific Quartus II features and methodologies, this chapter provides references to other appropriate chapters in the *Quartus II Handbook*.

After you select a device family, to check if additional guidelines are available, refer to the design guidelines section of the appropriate device handbook.

Creating Design Specifications

Before you create your logic design or complete your system design, create detailed design specifications that define the system, specify the I/O interfaces for the FPGA, identify the different clock domains, and include a block diagram of basic design functions.

Creating a test plan also helps you to design for testability and manufacturability. For example, if you want to perform any built-in self-test functions to drive interfaces, you can use a UART interface with a Nios® II processor inside the FPGA device. You might need to validate all the design interfaces. For guidelines related to analyzing and debugging the device after it is in the system, refer to "Planning for On-Chip Debugging Options" on page 1–10.

If more than one designer works on your design, you should consider a common design directory structure or source control system to make design integration easier. For more suggestions on team-based designs, refer to "Planning for Hierarchical and Team-Based Design" on page 1–13.

Intellectual Property Selection

Altera and its third-party intellectual property (IP) partners offer a large selection of off-the-shelf IP cores optimized for Altera devices. The IP you select often affects system design, especially if the FPGA interfaces with other devices in the system. Consider which I/O interfaces or other blocks in your system design are implemented using IP cores, and plan to incorporate these cores in your FPGA design.

The OpenCore Plus feature, which is available for many IP cores, allows you to program the FPGA to verify your design in the hardware before you purchase the IP license. The evaluation supports the following modes:

- Untethered—The design runs for a limited time.
- Tethered—The design requires an Altera serial JTAG cable connected between the JTAG port on your board and a host computer running the Quartus II Programmer for the duration of the hardware evaluation period.

For descriptions of available IP cores, refer to the Intellectual Property page on the Altera website.

System Design

If your design uses Altera IP, consider using the Quartus II SOPC Builder as a system development tool. With SOPC Builder, you can specify system components in a GUI, and SOPC Builder generates the interconnect logic automatically. SOPC Builder outputs HDL files that define all components of the system, and a top-level HDL design file that connects all the components together. Because SOPC Builder changes your design entry methodology, you should plan to start developing your design within SOPC Builder from the beginning of the design cycle so that you do not need to make changes later on.

SOPC Builder is commonly used as the tool for creating systems based on the Nios II processor. However, SOPC Builder is a general-purpose tool for creating SOPC designs that may or may not contain a processor. SOPC Builder components use Avalon® interfaces for the physical connection of components, and you can use SOPC Builder to connect any logical device (either on-chip or off-chip) that has an Avalon interface. The Avalon Memory-Mapped interface allows a component to use an address mapped read or write protocol that enables flexible topologies for connecting master components to any slave components. The Avalon Streaming interface enables point-to-point connections between streaming components that send and receive data using a high-speed, unidirectional system interconnect between source and sink ports.

For information about using SOPC Builder to improve your productivity, refer to the Literature: SOPC Builder page on the Altera website.

Device Selection

To plan a design, first choose the best device for your application. The device you choose affects board specification and layout. This section provides guidelines in the planning process.

Choose the device family that best suits your design requirements. Families differ in 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 or 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 page on the Altera website. Each device family also has a device handbook or set of data sheets that documents the device features in detail. You can also see a summary of the resources for each device in the **Device** dialog box in the Quartus II software.

Carefully study the device density requirements for your design. Devices with more logic resources and higher I/O counts can implement larger and potentially more complex designs, but might cost more. Smaller devices use lower static power. Select a device that has some extra capacity that meets your design requirements, in case you want to add more logic later in the design cycle to upgrade or expand your design, or reserve logic and memory for on-chip debugging (refer to "Planning for On-Chip Debugging Options" on page 1–10). 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 older designs that target an Altera device, you can use their resource utilization as an estimate for your new design. Compile existing designs in the Quartus II software with the **Auto device selected by the Fitter** option in the **Settings** dialog box. Review the resource utilization to find out which device density fits the design. Consider coding style, device architecture, and the optimization options used in the Quartus II software, which can significantly affect the resource utilization and timing performance of your design.

To obtain resource utilization estimates for certain configurations of Altera's IP designs, refer to the user guides for Altera megafunctions and IP MegaCores on Literature: IP and Megafunctions section of the Altera website.

Device Migration Planning

Determine whether you want the option of migrating your design to another device density to allow flexibility when your design nears completion, or whether you want to migrate to a HardCopy® ASIC when your 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 meet their design requirements. 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 smaller device after prototyping. Similarly, many designers compile and optimize their design for an FPGA device and then migrate to a HardCopy ASIC when the design is complete and ready for higher-volume production. If you want the flexibility to migrate your design, you should specify these migration options in the Quartus II software at the beginning of your design cycle.

For more information about specifying the target migration devices, refer to *Specifying Devices for Device Migration* in Quartus II Help.

Selecting a migration device impacts pin placement because some pins may serve different functions in different device densities or package sizes. If you make pin assignments in the Quartus II software, the Pin Migration View in the Pin Planner highlights pins that change function between your migration devices. (For more information, refer to "Early Pin Planning and I/O Analysis" on page 1–6.) Selecting a companion device might 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 might require design changes to fit into the new target device. Altera recommends that you consider these issues early in the design cycle rather than at the end, when the design is near completion and ready for migration.

In addition, if you use a HardCopy ASIC, review HardCopy guidelines early in the design cycle for any Quartus II settings that you should use or other restrictions you should consider. You must 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*, and the *Quartus II Support for HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook*.

Planning for Device Programming or Configuration

You should plan how you want to program or configure the device in your system, which allows system and board designers to determine what companion devices, if any, your system requires. Your board layout also depends on the type of programming or configuration method you plan to use for programmable devices. Many programming options require a JTAG interface to connect to the devices, so you might have to set up a JTAG chain on the board. In addition, the Quartus II software uses the settings for the configuration scheme, configuration device, and configuration device voltage to enable the appropriate dual purpose pins as regular I/O pins after you complete configuration. The Quartus II software performs voltage compatibility checks of those pins during I/O assignment analysis and compilation of your design. Click the **Configuration** tab of the **Device and Pin Options** dialog box and select your configuration scheme.

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.

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 create any source code, or when you have a preliminary version of the design source code, and then perform the most accurate analysis with the PowerPlay Power Analyzer when you complete the design.

You must accurately estimate device power consumption to develop an appropriate power budget and to design the power supplies, voltage regulators, heat sink, and cooling system. Power estimation and analysis helps you satisfy two important planning requirements:

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

Power consumption in FPGA devices depends on the logic design. Altera's PowerPlay Early Power Estimator (EPE) spreadsheet allows you to estimate power utilization before you complete the design. To use the EPE, you must provide information about the device resources that you use 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 Quartus II software power estimator file can provide input to the EPE spreadsheet to specify information about your current design (refer to "Creating Powerplay EPE Spreadsheets").

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

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

When you complete the design, perform a complete power analysis to check the power consumption more accurately. The PowerPlay Power Analyzer tool in the Quartus II software provides an accurate estimation of power, ensuring that thermal and supply limitations are not violated. For the most accurate power estimation, use gate-level simulation results from a Verilog Value Change Dump File (**.vcd**) with the PowerPlay Power Analyzer.

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

Creating Powerplay EPE Spreadsheets

You can use PowerPlay EPE spreadsheets to perform a preliminary thermal analysis and power consumption estimate for your design. You can enter the data manually, or you can use the tools in the Quartus II software to assist you in generating the device resources usage information for your design.

If you manually enter data into the EPE spreadsheet, you can enter the device resources, operating frequency, toggle rates, and other parameters for your design. If you do not have an existing design, estimate the number of device resources used in your design and enter them manually.

If you have an existing design or a partially completed design, you can use the Quartus II software to generate the PowerPlay EPE file to assist you in completing the PowerPlay EPE spreadsheet.

For more information about generating the power estimation file, refer to *Performing an Early Power Estimate Using the PowerPlay Early Power Estimator* in Quartus II Help.

The PowerPlay EPE spreadsheet includes the Import Data macro that parses the information in the power estimation file and transfers the information into the spreadsheet. If you do not want to use the macro, manually transfer the data into the EPE spreadsheet. For example, after importing the PowerPlay EPE file information into the PowerPlay EPE spreadsheet, you can add additional devices resource information. If the existing Quartus II project represents only a portion of your full design, you can manually enter the additional device resources you use in the final design.

Early Pin Planning and I/O Analysis

In many design environments, FPGA designers want to plan the top-level FPGA I/O pins early in the design to help board designers begin the PCB design and layout. The I/O capabilities and board layout guidelines of the FPGA device influence pin locations and other types of assignments. If the board design team specifies an FPGA pin-out, it is crucial that the pin locations are verified in the FPGA placement and routing software to avoid board design changes.

Without early pin planning features, designers and system architects could not check the validity of FPGA pin assignments until the design was completed. You can create a preliminary pin-out for an Altera FPGA with the Quartus II Pin Planner before you develop the source code, based on standard I/O interfaces (such as memory and bus interfaces) and any other I/O-related assignments defined by system requirements. The Quartus II I/O Assignment Analysis checks that the pin locations and assignments are supported in the target FPGA architecture. You can then use I/O Assignment Analysis to validate I/O-related assignments that you create or modify throughout the design process. When you compile your design in the Quartus II software, I/O Assignment Analysis is run automatically in the Fitter to validate that the assignments meet all the device requirements and generates messages if there are problems.

This section describes pin planning and I/O analysis features at these different stages of the design flow.

Early in the design process, before the source code is created, the system architect has information about the standard I/O interfaces (such as memory and bus interfaces), the IP cores that are used in the design, and any other I/O-related assignments defined by system requirements. 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. You can configure how the functions and cores are connected to each other by specifying matching node names for selected ports in the **Set Up Top-Level Design File** dialog box. Create any other I/O-related assignments for these interfaces or other design I/O pins in the Pin Planner, as described in this section. When you have entered as much I/O-related 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 even before the design is created, 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 you complete initial pin planning, you can create a Quartus II Revision based on the Quartus II-generated netlist. You can then 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.

During this initial pin planning, after you have generated a top-level design file, or when you have developed your design source code, you can assign pin locations and assignments using the Pin Planner.

The Pin Planner enables easy I/O pin assignment planning, assignment, and validation. You can use the View menu in the Pin Planner to create pin location and other assignments using a device package view instead of pin numbers.

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 1–4, the **Pin Migration View** highlights the pins that have changed functions in the migration device when compared to the currently selected device. Selecting the 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 can 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 **.csv** containing your I/O assignments for spreadsheet use when you assign all pins.

When you complete your pin planning, you can pass pin location information 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 suggested changes. Your pin assignments must match between the Quartus II software and your schematic and board layout tools to ensure the FPGA works correctly on the board, especially if you must make changes to the pin-out. The system architect uses the Quartus II software to pass pin information to team members designing individual logic blocks, allowing them to achieve better timing closure when they compile their design.

Start FPGA planning before you complete the HDL design to improve the confidence in early board layouts, reduce the chance of error, and improve the overall time to market of the design. When you complete the design, use the Fitter reports for the final sign-off of pin assignments. After compilation, the Quartus II software generates the Pin-Out File (**.pin**), and you can use this file to verify that each pin is correctly connected in board schematics.

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*.

Simultaneous Switching Noise Analysis

Simultaneous switching noise (SSN) is a noise voltage inducted onto a victim I/O pin of a device due to the switching behavior of other aggressor I/O pins in the device. SSN often leads to the degradation of signal integrity by causing signal distortion, thereby reducing the noise margin of a system. The best approach is to address SSN with estimation early in your system design, to reduce the chance of any later board design changes. When the design is complete, perform a complete SSN analysis of your FPGA in the Quartus II software to verify the board design.

Altera provides 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.

You can use the ESE tool to estimate SSN in your FPGA design, which is available for various device families.

For more information and device support for the ESE spreadsheet tool, refer to Altera's Signal Integrity Center on the Altera website. For more information about the SSN Analyzer, refer to the *Simultaneous Switching Noise (SSN) Analysis and Optimizations* chapter in volume 2 of the *Quartus II Handbook*.

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

The Quartus II software includes integrated synthesis that supports Verilog HDL and VHDL, as well as the Altera hardware description language (AHDL) and schematic design entry. You can also use supported standard third-party EDA synthesis tools to synthesize your Verilog HDL or VHDL design, and then compile the resulting output netlist file in the Quartus II software. Different synthesis tools may give different results for each design. To assess the best-performing tool for your application, you can experiment by synthesizing typical designs for your specific application and coding style. Perform placement and routing in the Quartus II software to get accurate timing analysis and logic utilization results.

Because tool vendors frequently add new features, fix tool issues, and enhance performance for Altera devices, Altera recommends using the most recent version of third-party synthesis tools. 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 Quartus II software.

The synthesis tool you choose may allow you 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 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. For more information, refer to "Incremental Compilation with Design Partitions" on page 1–14.

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 Starter Edition with the Quartus II software. You can also purchase the ModelSim-Altera Edition to support large designs and achieve faster simulation performance. The Quartus II software can generate both functional and timing netlist files for ModelSim and other third-party simulators.

Use the simulator version that is supported with your Quartus II version for best results. You should also 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. The *Quartus II Software Release Notes* list the version of each simulation tool that is supported with that particular version of the Quartus II software.

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

Consider whether your desired formal verification flow is supported, and whether the flow impacts the design and compilation stages of your design.

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

Using a formal verification flow can impact performance results because the flow requires turning off certain logic optimizations, such as register retiming, and forces you to preserve hierarchy blocks, which can restrict optimization. Formal verification treats memory blocks as black boxes. Therefore, you should keep memory in a separate hierarchy block so other logic does not get incorporated into the black box for verification. Other restrictions may limit your design, so consult the documentation for details. If formal verification is important to your design, plan for limitations and restrictions in the beginning rather 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.

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. You should 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. You can reduce debugging time and avoid later changes to accommodate your preferred debugging methodologies.

For more information about debugging tools, refer to *Section IV. In-System Debugging* in volume 3 of the Quartus II Handbook. For an overview of debugging options that can help you decide which option to use, refer to the *System Debugging Tools Overview* chapter 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 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 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 use incremental debugging with the SignalTap II Logic Analyzer, you must turn on the **Full incremental compilation** option. If you do not turn on incremental compilation, 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 Logic Analyzer greatly reduces the compilation time required for debugging.

The 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 later. Keep in mind that the Logic Analyzer Interface can multiplex signals with design I/O pins if required. Ensure that your board supports a 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 and incorporate it as part of the design process.

The In-System Sources and Probes feature requires that you instantiate a megafunction in your HDL code. In addition, you can instantiate the SignalTap II Logic Analyzer as a megafunction, so you can manually connect the SignalTap II Logic Analyzer to nodes in your design 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.

Design Practices and HDL Coding Styles

When you develop complex FPGA designs, design practices and coding styles have an enormous impact on the timing performance, logic utilization, and system reliability of your device.

Design Recommendations

You can use synchronous design practices to consistently meet your design goals. Problems with asynchronous 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. When you meet all register timing requirements, 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.

Clock signals have a large effect on the timing accuracy, performance, and reliability of your design. 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 you have PLLs 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. The Design Assistant checks your design for adherence to Altera-recommended design guidelines. 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.

You should also understand the target architecture of your device to target your design to take advantage of those features. For example, the control signals should use the dedicated control signals in the device architecture, so in some cases you might need 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** (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. If you design memory and DSP functions, you should understand the target architecture of your device so you can use 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*. For any additional tool-specific guidelines, refer to the documentation of your synthesis tool.

Managing Metastability

Metastability problems can occur in digital design when a signal is transferred between circuitry in unrelated or asynchronous clock domains, because the designer cannot guarantee that the signal meets the setup and hold time requirements during the signal transfer. Designers commonly use a synchronization chain to minimize the occurrence of metastable events. Ensure that your design accounts for synchronization between any asynchronous clock domains. Consider using a synchronizer chain of more than two registers for high-frequency clocks and frequently-toggling data signals to reduce the chance of a metastability failure.

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. 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. Determine an acceptable target MTBF given the context of your entire system and the fact that MTBF calculations are statistical estimates.

The Quartus II software can help you determine whether you have enough synchronization registers in your design to produce a high enough MTBF at your clock and data frequencies.

For information about the industry-leading 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*.

Planning for Hierarchical and Team-Based Design

If you want to create a hierarchical design so that you can use compilation-time savings and performance preservation of the Quartus II software 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. 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 optionally creating a design floorplan.

For information 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 the flat compilation flow with no design partitions in the Quartus II software, the entire design is compiled together in a "flat" netlist. Use this flow if you do not create any design partitions. Your source code can have hierarchy, but the design is flattened during compilation and all 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.

Incremental Compilation with Design Partitions

In an incremental compilation flow, the system architect splits a large design into partitions. Incremental compilation preserves the compilation results and performance of unchanged partitions in the 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 empty partitions to indicate that parts of your design are incomplete or missing. Third-party IP designers can also export logic blocks to be incorporated into the top-level. 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 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.

Alternatively, different designers or IP providers may be working on different blocks of the design separately, and you may want to export these blocks to incorporate in the top-level design.

Designers of lower-level blocks can export a post-synthesis or post-fitting netlist for their design, along with a set of assignments. The system architect then incorporates each design block as a design partition in a top-level project.

With exported partitions, the system architect must provide guidance to designers or IP providers to ensure that each partition uses the appropriate device resources. Because the designs may be developed independently, each 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, is 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 allow third-party designs to access the top-level project framework. By designing within a copy of the top-level project (or by checking out the project files in a source control environment), the designers of the lower-level block have full information about the entire project, which helps to ensure optimal results.

When you are 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. For information about support for Quartus II incremental compilation, refer to your synthesis tool documentation. The netlists are then considered the source files for incremental compilation.

Planning Design Partitions and Floorplan Location Assignments

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.

If you have timing-critical partitions that are changing through the design flow, or partitions exported from another Quartus II project, you can create design floorplan assignments to constrain the placement of the affected partitions. Creating location assignments ensures that no location conflicts occur between different partitions. In addition, design floorplan assignments help 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 in which most resources are claimed. You can use the Quartus II Chip Planner to create floorplan assignments using LogicLock™ region assignments for affected design partitions. With a basic design framework for the top-level design, you can 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 partitions meet top-level design requirements when integrated with the rest of the design, reducing time spent integrating and verifying the timing of the top-level design.

For detailed guidelines about creating design partitions and organizing your source code, as well as information about when and how to create floorplan assignments for incremental compilations, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan* chapter in volume 1 of the *Quartus II Handbook*.

For more information about creating placement assignments in the Chip Planner, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Fast Synthesis and Early Timing Estimation

It is more cost-effective 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.

If you are designing individual blocks separately, you can use these features as you develop the design. Any issues highlighted in the lower-level design blocks are communicated to the system architect. Resolving these issues might require allocating additional device resources to the individual block or changing its timing budget.

If you are a top-level designer, you can also use fast synthesis and early timing estimation to prototype the entire design. Incomplete partitions are 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.

Running early timing estimations helps you to 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, you can reduce time in later stages of the process. Use 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 cost and development time.

Document Revision History

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

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

Date	Version	Changes Made
July 2010	10.0.0	■ Added new section "System Design" on page 1–3 ■ Removed details about debugging tools from "Planning for On-Chip Debugging Options" on page 1–10 and referred to other handbook chapters for more information ■ Updated information on recommended design flows in "Incremental Compilation with Design Partitions" on page 1–14 and removed "Single-Project Versus Multiple-Project Incremental Flows" heading ■ Merged the "Planning Design Partitions" section with the "Creating a Design Floorplan" section. Changed heading title to "Planning Design Partitions and Floorplan Location Assignments" on page 1–15 ■ Removed "Creating a Design Floorplan" section ■ Removed "Referenced Documents" section ■ Minor updates throughout chapter
November 2009	9.1.0	■ Added details to "Creating Design Specifications" on page 1–2 ■ Added details to "Intellectual Property Selection" on page 1–2 ■ Updated information on "Device Selection" on page 1–3 ■ Added reference to "Device Migration Planning" on page 1–4 ■ Removed information from "Planning for Device Programming or Configuration" on page 1–4 ■ Added details to "Early Power Estimation" on page 1–5 ■ Updated information on "Early Pin Planning and I/O Analysis" on page 1–6 ■ Updated information on "Creating a Top-Level Design File for I/O Analysis" on page 1–8 ■ Added new "Simultaneous Switching Noise Analysis" section ■ Updated information on "Synthesis Tools" on page 1–9 ■ Updated information on "Simulation Tools" on page 1–9 ■ Updated information on "Planning for On-Chip Debugging Options" on page 1–10 ■ Added new "Managing Metastability" section ■ Changed heading title "Top-Down Versus Bottom-Up Incremental Flows" to "Single-Project Versus Multiple-Project Incremental Flows" ■ Updated information on "Creating a Design Floorplan" on page 1–18 ■ Removed information from "Fast Synthesis and Early Timing Estimation" on page 1–18
March 2009	9.0.0	■ No change to content

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

Date	Version	Changes Made
November 2008	8.1.0	■ Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	■ 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 and Floorplan Assignments chapter ■ Added reference to the Quartus II Language Templates

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

Take an online survey to provide feedback about this handbook chapter.

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

QII51015-10.0.0

This chapter provides information and design examples to help you partition your design to take advantage of the Quartus II incremental compilation feature, and to help you create a design floorplan using LogicLock™ regions to support an incremental flow when recommended.

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 an average of 75% for small changes in large design changes, so that you can perform more design iterations per day and achieve timing closure efficiently.
- Facilitates modular hierarchical and team-based design flows, as well as design reuse and intellectual property (IP) delivery.

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 ASIC Migration Flows" on page 2–49). Incremental Compilation is not supported for Stratix V devices in the Quartus II software version 10.0.

This document contains the following sections:

- "Deciding Whether to Use an Incremental Compilation Flow" on page 2–1
- "Incremental Compilation Summary" on page 2–7
- "Deciding which Design Blocks Should Be Design Partitions" on page 2–12
- "Creating Design Partitions" on page 2–16
- "Specifying the Level of Results Preservation for Subsequent Compilations" on page 2–21
- "Exporting Design Partitions from Separate Quartus II Projects" on page 2–27
- "Creating a Design Floorplan with LogicLock Regions" on page 2–45
- "Incremental Compilation Restrictions" on page 2–47
- "Scripting Support" on page 2–55

Deciding Whether to Use an Incremental Compilation Flow

The Quartus II incremental compilation feature enhances the standard Quartus II design flow by allowing you to preserve satisfactory compilation results and performance of unchanged blocks of your design.

This section outlines the flat compilation flow with no design partitions in "Flat Compilation Flow with No Design Partitions", and the incremental flow when you divide the design into partitions "Incremental Compilation Flow with Design Partitions" on page 2–3, 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. A discussion about incremental and team design flows in "Team-Based Design Flows and IP Delivery" on page 2–6 describes how it is beneficial to keep your design project within one Quartus II project, as well as when it might be necessary for other team members or IP providers to develop particular design blocks or partitions separately, and then later integrate their partitions into the top-level design.

Flat Compilation Flow with No Design Partitions

In the flat compilation flow with no design partitions, all the source code is processed with the Analysis and 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 ensure optimal push-button 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.

Refer to the next subsection for ways to reduce compilation time when you use a flat compilation for your design.

Incremental Capabilities Available When Your Design Has No Partitions

The Quartus II software has incremental compilation capabilities available even when you do not partition your design, including, Smart Compilation, incremental debugging, and Rapid Recompile. These features work with design partitions as well, if you do follow an incremental design flow.

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 turned on, the compiler skips the Analysis and Synthesis module if all the design source files are unchanged. The Smart Compilation feature skips entire compiler stages and cannot make incremental changes within a given stage of the compilation flow. You can turn on Smart Compilation in the **Settings** dialog box on the **Compilation Process Settings** page.

During the debugging stage of the design cycle, you can use incremental compilation to add the SignalTap II Logic Analyzer incrementally to your design, even if the design does not have partitions. To preserve the compilation netlist for the entire design, set the Netlist Type to Post-Fit for the automatically-created "Top" partition. Refer to "Debugging Incrementally with the SignalTap II Logic Analyzer" on page 2–11 for more information.

The Quartus II software also includes a Rapid Recompile feature that instructs the compiler to reuse the compatible compilation results if most of the design has not changed since the last compilation. This feature reduces compilation times for small and isolated design changes. You do not have control over which parts of the design are recompiled using this option; the compiler determines which parts of the design must be recompiled. You can turn on the **Rapid Recompile** option on the **Incremental Compilation** page in the **Settings** dialog box.

Incremental Compilation Flow with Design Partitions

In the standard incremental compilation design flow, the top-level design is divided into partitions, which can be compiled and optimized together in the top-level Quartus II project. You can preserve fitting results and performance for completed partitions while other parts of the design are changing, which reduces the compilation times for each design iteration.

Incremental compilation is recommended for large designs and high resource densities when preserving results is important to achieve timing closure. The incremental compilation feature also facilitates team-based design flows that allow designers to create and optimize design blocks independently, when necessary. Refer to the next section "Team-Based Design Flows and IP Delivery" on page 2–6 for more information.

To take advantage of incremental compilation, start by splitting your design along any of its hierarchical boundaries into design blocks to be compiled incrementally, and assign each block as a design partition. The Quartus II software synthesizes each individual hierarchical design partition separately, and then merges the partitions into a complete netlist for subsequent stages of the compilation flow. When recompiling your design, you can use source code, post-synthesis results, or post-fitting results to preserve satisfactory results for each partition. Refer to "Incremental Compilation Summary" on page 2–7 and subsequent sections for more details.

In a team-based environment, part of your design may be incomplete, or it may have been developed by another designer or IP provider. In this scenario, you can add the completed partitions to the design incrementally. Alternatively, other designers or IP providers can develop and optimize partitions independently, and then the project lead can later integrate the partitions into the top-level design. Refer to "Team-Based Design Flows and IP Delivery" on page 2–6 for more details.

Table 2–1 shows a summary of the impact the Quartus II incremental compilation feature has on compilation results.

Table 2–1. Impact Summary of Using Incremental Compilation (Part 1 of 2)

Characteristic	Impact of Incremental Compilation with Design Partitions
Compilation Time Savings	Typically saves an average of 75% compilation time for small design changes in large designs when post-fit netlists are preserved; there are savings in both Quartus II Integrated Synthesis and the Fitter. *(1)*
Performance Preservation	Excellent performance preservation when timing 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.

Table 2–1. Impact Summary of Using Incremental Compilation (Part 2 of 2)

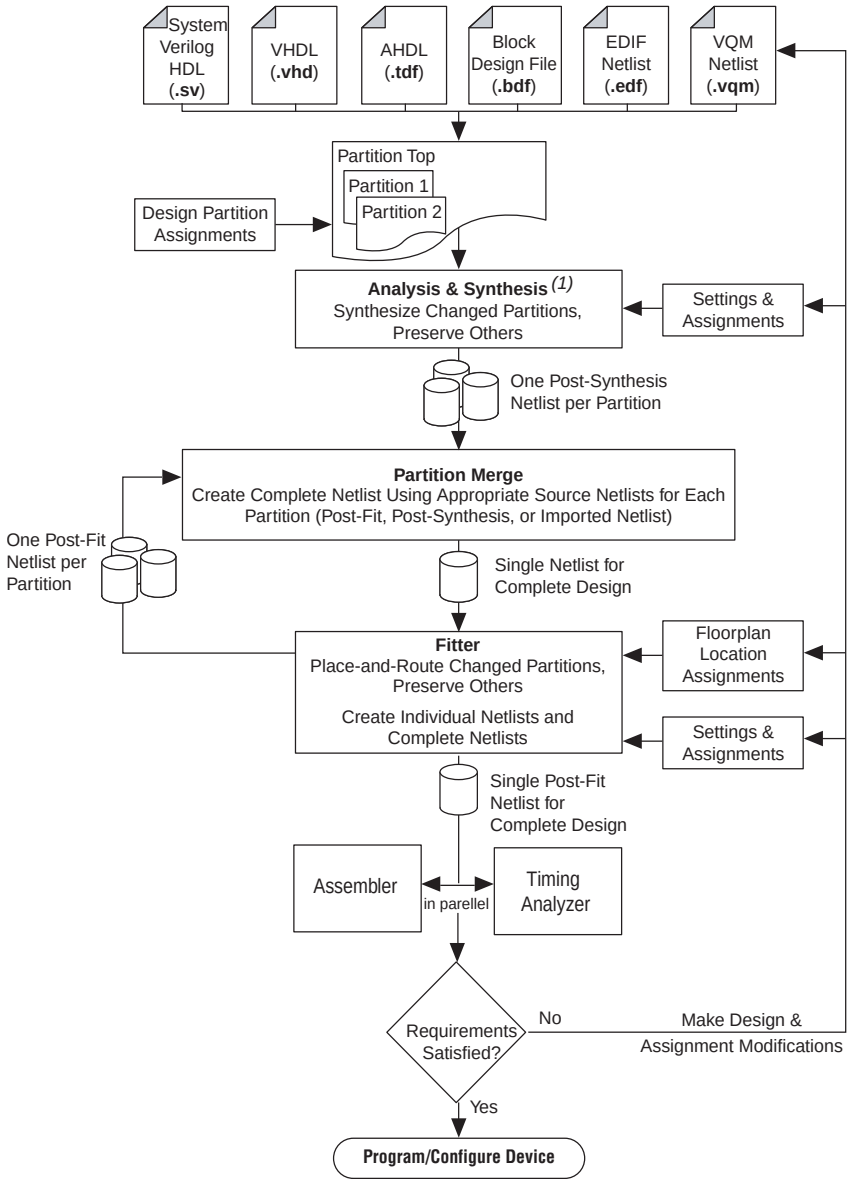
Characteristic	Impact of Incremental Compilation with Design Partitions
f_{MAX} Changes	The design's maximum frequency might be reduced because cross-boundary optimizations are no longer possible. If the design is partitioned and the floorplan location assignments are created appropriately, there might be no negative impact on f_{MAX}.
Notes to Table 2–1:	
(1) Quartus II incremental compilation does not reduce processing time for the early "pre-fitter" operations, such as determining pin locations and clock routing, so the feature cannot reduce compilation time if runtime is dominated by those operations.	

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

For more information and recommendations on how to prepare your design to use the Quartus II incremental compilation feature, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.

Figure 2–1 shows a diagram of the Quartus II design flow using incremental compilation with design partitions.

Figure 2–1. Quartus II Design Flow Using Incremental Compilation



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.

The diagram in Figure 2–1 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 exported from other Quartus II projects, depending on the netlist type that you specify for each partition.

The Fitter then processes the merged netlist, preserves the placement and routing of unchanged partitions, and refits only those partitions that have changed. The Fitter generates the complete netlist for use in future stages of the compilation flow, including timing analysis and programming file generation, which can take place in parallel if more than one processor is enabled for use in the Quartus II software. The Fitter also generates individual netlists for each partition so that 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.

Team-Based Design Flows and IP Delivery

The Quartus II software supports various design flows to enable team-based design and third-party IP delivery. A top-level design can include one or more partitions that are designed or optimized by different designers or IP providers, as well as partitions that will be developed as part of a standard incremental methodology.

In a team-based environment, part of your design may be incomplete because it is being developed elsewhere. The project lead or system architect can create empty placeholders in the top-level design for partitions that are not yet complete. Designers or IP providers can create and verify HDL code separately, and then the project lead later integrates the code into the single top-level Quartus II project. In this scenario, you can add the completed partitions to the design incrementally, however, the design flow allows all design optimization to occur in the top-level design for easiest design integration. Altera recommends using a single Quartus II project whenever possible because using multiple projects can add significant up-front and debugging time to the development cycle.

Alternatively, partition designers can design their partition in a copy of the top-level design, or in a separate project. Designers export their completed partition as either a post-synthesis netlist, or optimized placed and routed netlist, or both, along with assignments such as LogicLock™ regions, as appropriate. The project lead then integrates each design block as a design partition into the top-level design. Altera recommends that designers export and reuse post-synthesis netlists, unless optimized post-fit results are required in the top-level design, to simplify design optimization.

Teams following a bottom-up design approach often want to optimize placement and routing of individual design partitions independently. Optimizing design partitions in separate Quartus II projects, and then later integrating the results into a top-level design, can have the following potential drawbacks that require careful planning:

- Achieving timing closure for the full design may be more difficult if you compile partitions independently without information about other partitions in the design. This problem may be avoided by careful timing budgeting and special design rules, such as always registering the ports at the module boundaries.
- Resource budgeting and allocation may be required to avoid resource conflicts and overuse. Creating a floorplan with LogicLock regions is recommended when design partitions are developed independently in separate Quartus II projects.

A unique challenge of team-based design and IP delivery for FPGAs is the fact that the partitions being developed independently must share a common set of resources. To minimize issues that might arise from sharing a common set of resources, you can design partitions within a single Quartus II project, or a copy of the top-level design.

A common project ensures that designers have a consistent view of the top-level project framework.

For timing-critical partitions being developed and optimized by another designer, it is important that each designer has complete information about the top-level design in order to maintain timing closure during integration, and to obtain the best results. When you want to integrate partitions from separate Quartus II projects, the project lead can perform most of the design planning, and then pass the top-level design constraints to the partition designers. Preferably, partition designers can obtain a copy of the top-level design by checking out the required files from a source control system. Alternatively, the project lead can provide a copy of the top-level project framework, or pass design information using Quartus II-generated design partition scripts. In the case that a third-party designer has no information about the top-level design, developers can export their partition from an independent project if required.

For more information about managing team-based design flows, refer to"Exporting Design Partitions from Separate Quartus II Projects" on page 2–27 and the subsection "Project Management—Making the Top-Level Design Available to Other Designers" on page 2–29.

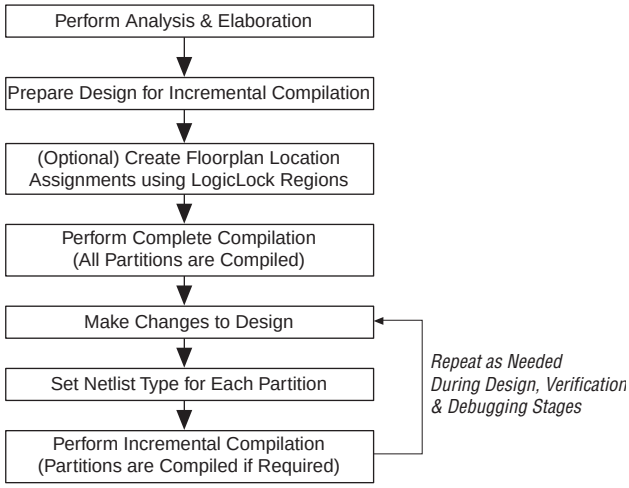
Exporting partitions is not supported in HardCopy or FPGA companion device compilations when there is a migration device setting. For details, refer to "HardCopy ASIC Migration Flows" on page 2–49.

Incremental Compilation Summary

This section provides a summary of the standard incremental compilation design flow and includes common recommended design scenarios.

Figure 2–2 illustrates the incremental compilation design flow when all partitions are contained in one top-level design.

Figure 2–2. Summary of Standard Incremental Compilation Design Flow



Steps for Incremental Compilation

The following sections describe how to prepare a design to use the incremental compilation feature, and then how to preserve satisfactory results and performance in subsequent incremental compilations.

For an interactive introduction to implementing an incremental compilation design flow, refer to the **Getting Started Tutorial** on the Help menu in the Quartus II software. For a step-by-step introduction on how to use incremental compilation, refer to *Using the Incremental Compilation Design Flow* in Quartus II Help.

Preparing a Design for Incremental Compilation

To begin, elaborate your design, or run any compilation flow (such as a full compilation) that includes the elaboration step. Elaboration is the part of the synthesis process that identifies your design's hierarchy.

Next, designate specific instances in the design hierarchy as design partitions. You can create design partitions in the Project Navigator, the Design Partitions window, or with the Design Partition Planner. You can also use the Auto-Partition feature in the Design Partition Planner to create partitions based on the size and connectivity of the hierarchical design blocks.

For more information about design partitions and what part of your design can be specified as a design partition, refer to "Deciding which Design Blocks Should Be Design Partitions" on page 2–12. For details on how to create design partitions, refer to "Creating Design Partitions" on page 2–16.

If required for your design flow, create a floorplan with LogicLock regions location assignments for timing-critical partitions that change with future compilations. Assigning a partition to a physical region on the device can help maintain quality of results and avoid conflicts in certain situations. Refer to "Creating a Design Floorplan with LogicLock Regions" on page 2–45 for details about LogicLock region assignments.

Compiling a Design Using Incremental Compilation

The first compilation after making partition assignments is a full compilation, and prepares the design for subsequent incremental compilations. In subsequent compilations of your design, you can preserve satisfactory compilation results and performance of unchanged partitions with Netlist Type setting in the Design Partitions window.

For step-by-step information on compiling a design using incremental compilation to recompile only the changed parts of the design, refer to *Using the Incremental Compilation Design Flow* in Quartus II Help.

Common Design Scenarios Using Incremental Compilation

This section provides recommended applications of the incremental compilation flow and assumes that you have set up your design for incremental compilation as described in the previous section, "Preparing a Design for Incremental Compilation".

The following sections describe how to prepare a design to use the incremental compilation feature, and then how to preserve satisfactory results and performance in subsequent incremental compilations.

Reducing Compilation Time When Changing Source Files for One Partition

Use the flow in this example 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 flow also preserves the performance for these blocks, which reduces additional timing closure efforts.

Scenario background: You set up your design to include partitions for several of the major design blocks, and now you have just performed a lengthy compilation of the entire design. 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 a single source file:

1. Apply and save the fix to the HDL source file.
2. On the Assignments menu, open the Design Partitions window.
3. For the partitions that should be preserved, change the Netlist Type to **Post-Fit**. For the partition that contains the fix, you can change the netlist type to **Source File**.

   ☞ Using 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. Simulate the design 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

Use the flow in this example to optimize the results of one partition when the other partitions in the design have 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.

Scenario 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 to recompile a timing-critical partition with new optimization settings:

1. Open the 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**. You can optionally set the **Fitter Preservation Level** on the **Advanced** tab in the **Design Partitions Properties** dialog box 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 procedure.
2. Save the project and run the Design Space Explorer.

Preserving Results While Adding Additional Design Logic

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

Scenario 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 in isolation, 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, in the Design Partitions window, set the 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. You can set the **Fitter Preservation Level** on the **Advanced** tab in the **Design Partitions Properties** dialog box to **Placement** to allow more flexibility during routing if exact placement and routing preservation is not required.
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.

The flow in this example is similar to design flows in which a module is implemented separately and is later merged into the top-level. Generally, optimization in this flow works only if each critical path is contained within a single partition due to the effects described in "Deciding which Design Blocks Should Be Design Partitions" on page 2–12. Ensure that if there are any partitions representing a design file that is missing from the project, you create a placeholder wrapper file to define the port interface. Refer to "Empty Partitions" on page 2–29 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 to add the SignalTap II Logic Analyzer to your design without recompiling the original source code.

Use this flow to reduce compilation times when you add 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 design partitions in order 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 Logic Analyzer in an incremental compilation flow:

1. Open the 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 have 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 to tap directly to the pre-synthesis node names that you choose. In this case, the partition is refit, so that the placement is typically different from previous fitting results.

   Do not use the **Post-Synthesis** netlist type 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*.

Deciding which Design Blocks Should Be Design Partitions

The incremental compilation design flow requires 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.

This section includes the following topics:

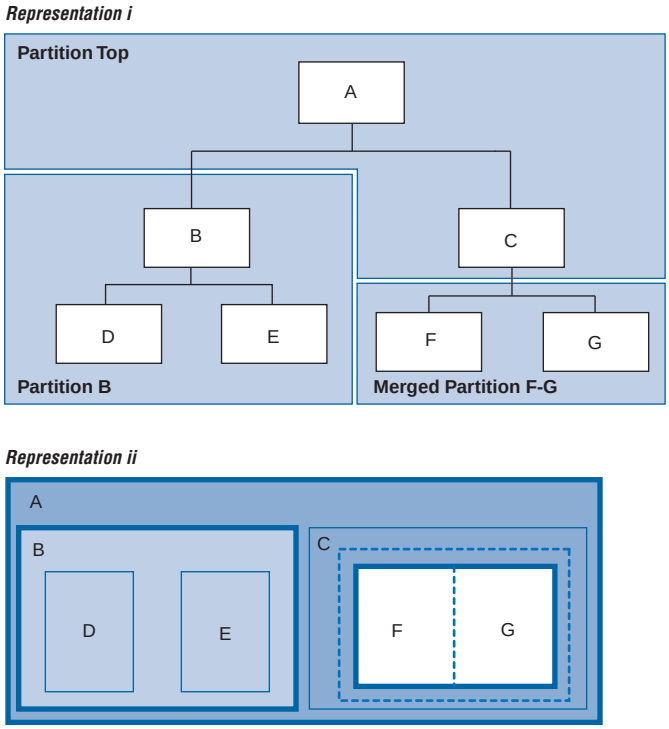
- "Impact of Design Partitions on Design Optimization" on page 2–14
- "Design Partition Assignments Compared to Physical Placement Assignments" on page 2–15
- "Using Partitions with Third-Party Synthesis Tools" on page 2–15

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 automatically consider each design entity or instance to be a design partition for incremental compilation; 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 2–14. 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. You can merge partitions that have the same immediate parent partition to create a single partition that includes more than one hierarchical entity in the design. 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 2–3, a complete design is made up of instances **A**, **B**, **C**, **D**, **E**, **F**, and **G**. The shaded boxes in Representation i indicate design partitions in a "tree" representation of the hierarchy. In Representation ii, 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, partition **B** contains the logic in instances **B**, **D**, and **E**. Entities **F** and **G** were first identified as separate partitions, and then merged together to create a partition **F-G**. The partition for the top-level entity **A**, called "Top", includes the logic in one of its lower-level instances, **C**, because **C** was not defined as part of any other partition.

Figure 2–3. Partitions in a Hierarchical Design



You can create 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.

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. You can also merge two or more partitions to allow cross-boundary optimizations for paths that cross between the partitions, as long as the partitions have the same parent partition. Merging related logic from different hierarchy blocks into one partition can be useful if you cannot change the design hierarchy to accommodate partition assignments.

The Design Partition Planner can help you create good assignments, as described in "Creating Design Partitions" on page 2–16. 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. 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 that the entire timing path meets its requirements. In addition, because each partition is optimized independently during synthesis, you may have to perform resource allocation to ensure that each partition uses an appropriate number of device resources. If design partitions are compiled in separate Quartus II projects, there may be conflicts related to global routing resources for clock signals when the design is integrated into the top-level design. You can use the Global Signal logic option to specify which clocks should use global or regional routing, use the ALTCLK_CTRL megafunction to instantiate a clock control block and connect it appropriately in both the partitions being developed in separate Quartus II 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 allocation, 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*.

Design Partition Assignments Compared to Physical Placement Assignments

Design partitions for incremental compilation are logical partitions, which is 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 that 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.

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 logic placement 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 region 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 for timing-critical design blocks that will change in subsequent compilations, or to improve the quality of results and avoid placement conflicts in some cases. Creating floorplan location assignments for design partitions using LogicLock regions is discussed in "Creating a Design Floorplan with LogicLock Regions" on page 2–45.

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*.

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.

Creating Design Partitions

There are several ways to designate a design instance as a design partition. This section provides an overview of tools you can use to create and analyze partitions in the Quartus II software.

Creating Design Partitions in the Project Navigator

You can right-click on an instance in the list under the **Hierarchy** tab in the **Project Navigator** and use the sub-menu to create and delete design partitions.

For detailed information about how to create design partitions in the Project Navigator, refer to *Creating Design Partitions* in Quartus II Help.

Creating Design Partitions In the Design Partitions Window

The Design Partitions window, available from the Assignments menu, allows you to create, delete, and merge partitions, and is the main window for setting the Netlist Type to specify the level of results preservation for each partition on subsequent compilations.

For information about how to set the Netlist Type and the available settings, refer to "Netlist Type for Design Partitions" on page 2–22.

The Design Partitions window also lists recommendations at the bottom of the window with links to the Incremental Compilation Advisor, where you can view additional recommendations about partitions. The **Color** column indicates the color of each partition as it appears in the Design Partition Planner and Chip Planner.

You can right-click on a partition in the window to perform various common tasks, such as viewing property information about a partition, including the time and date of the compilation netlists and the partition statistics. The **Advanced tab** in the **Design Partitions Properties** dialog box includes the Fitter Preservation Level setting.

For information about how to set the Fitter Preservation Level and the available settings, refer to "Fitter Preservation Level for Design Partitions" on page 2–23.

When you create a partition, the Quartus II software automatically generates a name based on the instance name and hierarchy path. You can edit the partition name in the Design Partitions Window so that you avoid referring to them by their hierarchy path, which can sometimes be long. This is especially useful when using command-line commands or assignments, or when you merge partitions to give the partition a meaningful name. 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.

For more information about how to create and manage design partitions in the Design Partitions window, refer to *Creating Design Partitions* in Quartus II Help.

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.

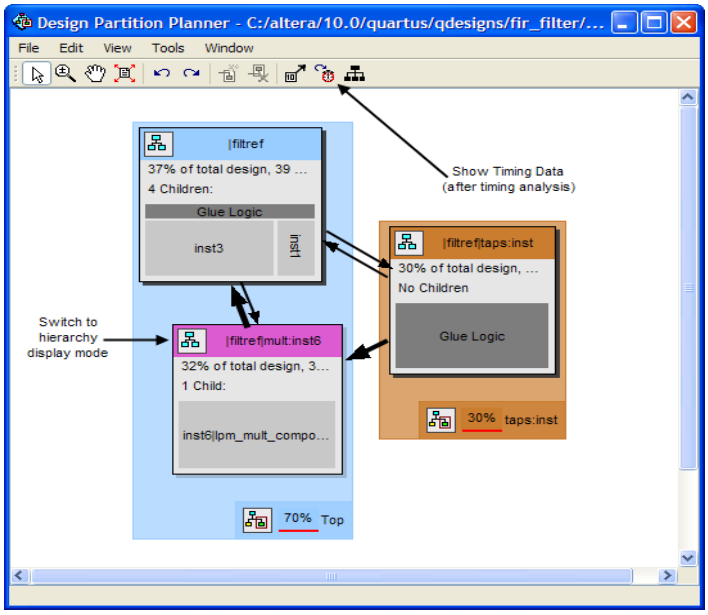
The Design Partition Planner displays a visual representation of design connectivity and hierarchy, as well as partitions and entity relationships. You can explore the connectivity between entities in the design, evaluate existing partitions with respect to connectivity between entities, and try new partitioning schemes in "what if" scenarios.

When you extract design blocks from the top-level design and drag them into the Design Partition Planner, connection bundles are drawn between entities, showing the number of connections existing between pairs of entities. In the Design Partition Planner, you can then set extracted design blocks as design partitions.

The Design Partition Planner also has an Auto-Partition feature that creates partitions based on the size and connectivity of the hierarchical design blocks.

Figure 2–4 shows the Design Partition Planner after making a design partition assignment to one instance (in the pale brown shaded box), and dragging another instance away from the top-level block within the same partition (two design blocks in the blue shaded box). The figure shows the number of connections between each partition and information about the size of each design instance.

Figure 2–4. Design Partition Planner



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

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 2–55.

Automatically-generated Partitions

The compiler creates some partitions automatically as part of the compilation process, which appear in some post-compilation reports. For example, the `sld_hub` partition is created for tools that use JTAG hub connections, such as the SignalTap II Logic Analyzer. The `hard_block` partition is created to contain certain "hard" or dedicated logic blocks in the device that are implemented in a separate partition so that they can be shared throughout the design.

Assessing Partition Quality

The Quartus II software provides various tools to assess the quality of your assigned design partitions. You can 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.

For more information about ensuring good partition quality, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.

This section provides information about the following Quartus II design tools:

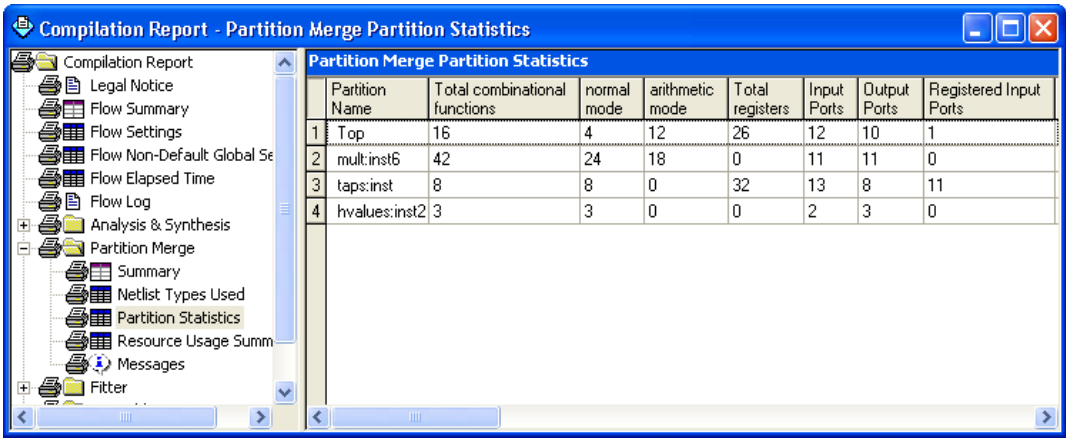
- "Partition Statistics Reports" on page 2–19
- "Partition Timing Reports" on page 2–19
- "Incremental Compilation Advisor" on page 2–20

Partition Statistics Reports

After compilation, you can view statistics about design partitions in the Partition Merge **Partition Statistics** compilation report, and on 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, 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



You can also view post-compilation statistics about the resource usage and port connections for a particular partition on the **Statistics** tab in the **Design Partition 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.

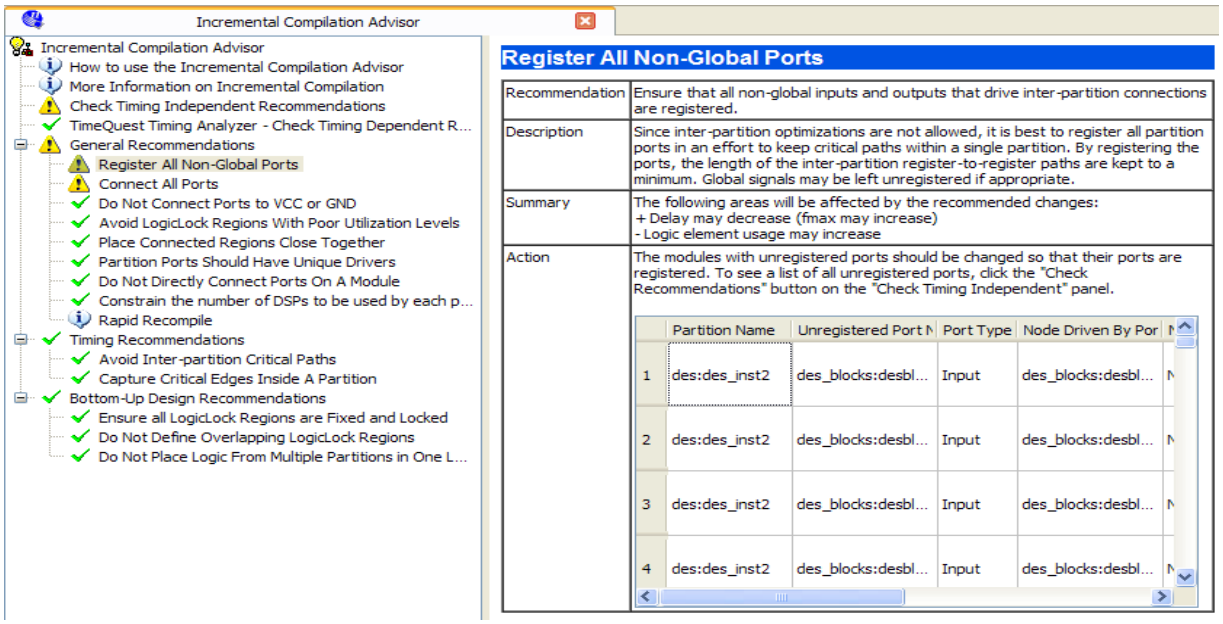
The Partition Timing Details report shows the number of failing partition-to-partition paths and worst-case slack for partition-to-partition paths, to provide a more detailed breakdown of where the critical paths in the design are located with respect to design partitions.

Incremental Compilation Advisor

You can use the Incremental Compilation Advisor to check that your design follows Altera's recommendations for creating design partitions and floorplan location assignments.

As shown in Figure 2–6, recommendations are split into **General Recommendations**, **Timing Recommendations**, and **Bottom-Up Design Recommendations** that apply to design flows in which partitions are compiled independently in separate Quartus II projects before being integrated into the top-level design. Each recommendation provides an explanation, describes the effect of the recommendation, and provides the action required to make a 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 2–6. Incremental Compilation Advisor



To verify that your design follows the recommendations, go to the **Timing Independent Recommendations** page or the **Timing Dependent Recommendations** page, and then 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 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.

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 Non-Global Ports** recommendation, the advisor displays a list of unregistered ports with the partition name and the node name associated with the port.

When the advisor provides a list of nodes, you can right-click on a node, and then 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.

Specifying the Level of Results Preservation for Subsequent Compilations

The Netlist Type is a property of each design partition that allows you to specify the level of results preservation. The Netlist Type property determines which type of netlist or source file the Partition Merge stage uses in the next incremental compilation.

When you choose to preserve a post-fit compilation netlist, the default level of Fitter preservation is the highest degree of placement and routing preservation supported by the device family. The advanced Fitter Preservation Level setting allows you to specify the amount of information that you want to preserve from the post-fit netlist file.

This section includes the following topics:

- "Netlist Type for Design Partitions" on page 2–22
- "Fitter Preservation Level for Design Partitions" on page 2–23
- "Where Are the Netlist Databases Saved?" on page 2–24
- "Deleting Netlists" on page 2–24
- "What Changes Initiate a Partition's Automatic Resynthesis?" on page 2–25

Netlist Type for Design Partitions

Before starting a new compilation, ensure that the appropriate Netlist Type is set for each partition to preserve the desired level of compilation results. Table 2–2 describes the 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.

Table 2–2. Partition Netlist Type Settings (Part 1 of 2)

Netlist Type	Quartus II Behavior for Partition During Compilation
Source File	Always compiles the partition using the associated design source file(s). *(1)* 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 initiates an automatic resynthesis has been made to the partition since the previous synthesis. *(2)* For details, refer to "What Changes Initiate a Partition's Automatic Resynthesis?" on page 2–25. Compiles the partition from the source files if resynthesis is initiated or if a post-synthesis netlist is not available. *(1)* 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 initiates an automatic resynthesis has been made to the partition since the previous fitting. *(2)* For details, refer to "What Changes Initiate a Partition's Automatic Resynthesis?" 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. Compiles the partition from the source files if resynthesis is initiated. *(1)* The Fitter Preservation Level specifies what level of information is preserved from the post-fit netlist. For details, refer to "Fitter Preservation Level for Design Partitions" on page 2–23. Assignment changes, such as Fitter optimization settings, do not cause a partition set to Post-Fit to recompile.

Table 2–2. Partition Netlist Type Settings (Part 2 of 2)

Netlist Type	Quartus II Behavior for Partition During Compilation
Empty	Uses an empty placeholder netlist for the partition. The partition's port interface information is required during Analysis and Synthesis to connect the partition correctly to other logic and partitions in the design, and peripheral nodes in the source file including pins and PLLs are preserved to help connect the empty partition to the rest of the design and preserve timing of any lower-level non-empty partitions within empty partitions. If the source file is not available, you can create a wrapper file that defines the design block and specifies the input, output, and bidirectional ports. In Verilog HDL: a module declaration, and in VHDL: 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 and the software reuses the previous netlist information without recompiling from the source files. You can use this netlist type to skip the compilation of a partition that is incomplete or missing from the top-level design. You can also set an empty partition if you want to compile only some partitions in the design, such as to optimize the placement of a timing-critical block such as an IP core before incorporating other design logic, or if the compilation time is large for one partition and you want to exclude it.

Notes to Table 2–2:

(1) If you turn on the **Rapid Recompile** option, the Quartus II software may not recompile the entire partition from the source code as described in this table; it will reuse compatible results if there have been only small changes to the logic in the partition. Refer to "Incremental Capabilities Available When Your Design Has No Partitions" on page 2–2 for more information.

(2) You can turn on the **Ignore changes in source files and strictly use the specified netlist, if available** option on the **Advanced** tab in the **Design Partitions Properties** dialog box to specify whether the Compiler should ignore source file changes when deciding whether to recompile the partition.

Fitter Preservation Level for Design Partitions

The Fitter Preservation Level is an advanced setting. The default Fitter Preservation Level for partitions with a **Post-Fit** netlist type is the highest level of preservation available for the target device family and provides the most compilation time reduction.

You can change the Fitter Preservation Level to provide more flexibility in the Fitter during placement and routing. You can set the Fitter Preservation Level on the **Advanced** tab in the **Design Partitions Properties** dialog box. Table 2–3 describes the Fitter Preservation Level settings.

Table 2–3. Fitter Preservation Level Settings (Part 1 of 2)

Fitter Preservation Level	Quartus II Behavior for Partition During Compilation
Placement and Routing	Preserves the design partition's netlist atoms and their placement and routing. *(1)* 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. You can turn on the Preserve high-speed tiles when preserving placement and routing option on the Advanced tab in the Design Partitions Properties dialog box to preserve high-speed optimization for programmable power tiles contained within the selected partition. This option applies to devices that support high-speed and low-power tiles.

Table 2–3. Fitter Preservation Level Settings (Part 2 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.
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.

For detailed information about how to set the Netlist Type and Fitter Preservation Level in the Quartus II software, refer to *Setting the Netlist Type and Fitter Preservation Level for Design Partitions* in Quartus II Help.

Where Are the Netlist Databases Saved?

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 2–48.

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 Quartus II Settings File (**.qxp)** for each partition in the design that will be integrated into the top-level design. Refer to "Exporting Design Partitions from Separate Quartus II Projects" on page 2–27 for more details about how to create a **.qxp** file for a partition within your design.

Deleting Netlists

You can choose to abandon all levels of results preservation and remove all netlists that exist for a particular partition with the **Delete Netlists** command in the Design Partitions window. When you delete netlists for a partition, the partition is compiled using the associated design source file(s) in the next compilation. Resetting the Netlist Type for a partition to **Source** would have the same effect, though the netlists would not be permanently deleted and would be available for use in subsequent compilations. For an imported partition, the **Delete Netlists** command also optionally allows you to remove the imported **.qxp** file.

What Changes Initiate 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**. Additionally, certain changes to a partition initiate 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 do not want this resynthesis to occur automatically, refer to "Forcing Use of the Compilation Netlist When a Partition has Changed" on page 2–27.

The following list explains the changes that initiate 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 "Resynthesis Due to Source Code Changes" on page 2–26 for details.
- The partition boundary was changed by an addition, removal, or change to the port boundaries of a partition (for example, a new partition has been 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.
- You have moved the project and compiled database between a Windows and Linux system. Due to the differences in the way new line feeds are handled between the operating systems, the internal checksum algorithm may detect a design file change in this case.

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. 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)

You can use the LogicLock Origin location assignment to change or fine-tune the previous Fitter results from a Post-Fit netlist. For details about how you can affect placement with LogicLock regions, refer to "Changing Partition Placement with LogicLock Changes" on page 2–47.

Resynthesis 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 description files used to create the design, and include Memory Initialization Files as well as **.qxp** files from exported partitions. When design files in a partition have dependencies on other files, changing one file may initiate 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 turn on the **Rapid Recompile** option, the Quartus II software may not recompile the entire partition from the source code as described in this section; it will reuse compatible results if there have been only small changes to the logic in the partition. Refer to "Incremental Capabilities Available When Your Design Has No Partitions" on page 2–2 for more information.

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 Compilation Netlist When a Partition has Changed

Forcing the use of a post-compilation 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, or if you are adding simple comments to the source file but you know the design logic itself is not being changed and you want to keep the previous compilation results.

To force the Fitter to use a previously generated netlist even when there are changes to the source files, right-click the partition in the Design Partitions window and then click **Design Partition Properties**. On the **Advanced** tab, turn on the **Ignore changes in source files and strictly use the specified netlist, if available** option.

Turning on this option can result in the generation of a functionally incorrect netlist when source design files change, because source file updates will not be recompiled. Use caution when setting this option.

Exporting Design Partitions from Separate Quartus II Projects

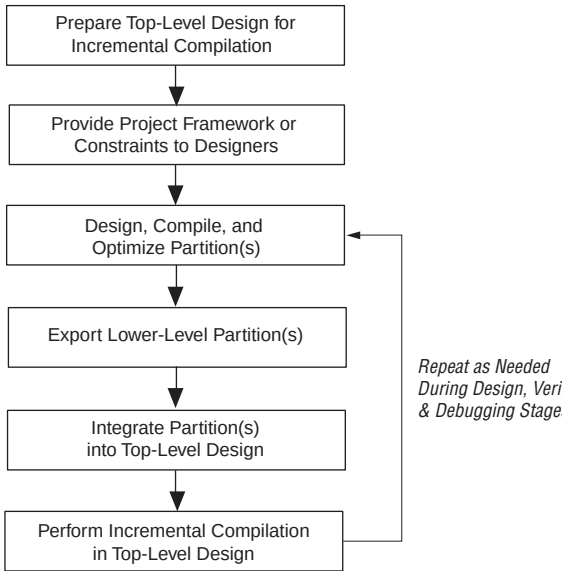
Partitions that are developed by other designers or team members in the same company or third-party IP providers can be exported as design partitions to a Quartus II Exported Partition File (**.qxp**), and then integrated into a top-level design. A **.qxp** file is a binary file that contains compilation results describing the exported design partition and includes a post-synthesis netlist, a post-fit netlist, or both, and a set of assignments, sometimes including LogicLock placement constraints. The **.qxp** file does not contain the source design files from the original Quartus II project.

To enable team-based development and third-party IP delivery, you can design and optimize partitions in separate copies of the top-level Quartus II project framework, or even in isolation. If the designers have access to the top-level project framework through a source control system, they can access project files as read-only and develop their partition within the source control system. If designers do not have access to a source control system, the project lead can provide the designer with a copy of the top-level project framework to use as they develop their partitions. The project lead also has the option to generate design partition scripts to manage resource and timing budgets in the top-level design when partitions are developed outside the top-level project framework.

The exported compilation results of completed partitions are given to the project lead, preferably using a source control system, who is then responsible for integrating them into the top-level design to obtain a fully functional design. This type of design flow is required only if partition 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.

The diagram in Figure 2–7 illustrates the team-based incremental compilation design flow using a methodology in which partitions are compiled in separate Quartus II projects before being integrated into the top-level design. This flow can be used when partitions are developed by other designers or IP providers.

Figure 2–7. Summary of Team-Based Incremental Compilation Flow



You cannot export or import partitions that have been merged. For more information about merged partitions, refer to "Deciding which Design Blocks Should Be Design Partitions" on page 2–12.

The following topics provide a description of the team-based design flow using exported partitions, describe how to generate a **.qxp** file for a design partition, and explain how to integrate the **.qxp** file into the top-level design:

- "Preparing the Top-Level Design" on page 2–28
- "Project Management—Making the Top-Level Design Available to Other Designers" on page 2–29
- "Exporting Partitions" on page 2–32
- "Viewing the Contents of a Quartus II Exported Partition File (.qxp)" on page 2–32
- "Integrating Partitions into the Top-Level Design" on page 2–33
- "Team-Based Design Optimization and Third-Party IP Delivery Scenarios" on page 2–36

There are some additional restrictions related to design flows using exported partitions, described in "Incremental Compilation Restrictions" on page 2–47.

Preparing the Top-Level Design

To prepare your design to incorporate exported partitions, first create the top-level project framework of the design to define the hierarchy for the subdesigns that will be implemented by other team members, designers, or IP providers.

In the top-level design, create project-wide settings, for example, device selection, global assignments for clocks and device I/O ports, and any global signal constraints to specify which signals can use global routing resources.

Next, create the appropriate design partition assignments and set the Netlist Type for each design partition that will be developed in a separate Quartus II project to **Empty**. Refer to "Empty Partitions" below for details. It may be necessary to constrain the location of partitions with LogicLock region assignments if they are timing-critical and are expected to change in future compilations, or if the designer or IP provider wants to place and route their design partition independently, to avoid location conflicts For details, refer to "Creating a Design Floorplan with LogicLock Regions" on page 2–45.

Finally, provide the top-level project framework to the partition designers, preferably through a source control system. Refer to "Project Management—Making the Top-Level Design Available to Other Designers" on page 2–29 for more information.

Empty Partitions

You can use a design flow in which some partitions are set to an **Empty** Netlist Type to develop pieces of the design separately, and then integrate them into the top-level design at a later time. In a team-based design environment, you can set the Netlist Type to **Empty** for partitions in your design that will be developed by other designers or IP providers. The **Empty** setting directs the Compiler to skip the compilation of a partition and use an empty placeholder netlist for the partition.

When a Netlist Type is set to **Empty**, peripheral nodes including pins and PLLs are preserved and all other logic is removed. The peripheral nodes including pins help connect the empty partition to the design, and the PLLs help preserve timing of non-empty partitions within empty partitions.

When you set a design partition to **Empty**, a design file is required during Analysis and Synthesis to specify the port interface information so that it can connect the partition correctly to other logic and partitions in the design. If a partition is exported from another project, the **.qxp** file contains this information. If there is no **.qxp** file or design file to represent the design entity, you must create a wrapper 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.

Project Management—Making the Top-Level Design Available to Other Designers

In team-based incremental compilation flows, whenever possible, all designers or IP providers should work within the same top-level project framework. Using the same project framework among team members ensures that designers have the settings and constraints needed for their partition, and makes timing closure easier when integrating the partitions into the top-level design. If other designers do not have access to the top-level project framework, the Quartus II software provides tools for passing project information to partition designers.

This section includes the following topics:

- "Distributing the Top-Level Quartus II Project" on page 2–29
- "Generating Design Partition Scripts" on page 2–31

Distributing the Top-Level Quartus II Project

There are several methods that the project lead can use to distribute the "skeleton" or top-level project framework to other partition designers or IP providers.

- If partition designers have access to the top-level project framework, the project will already include all the settings and constraints needed for the design. This framework should include PLLs and other interface logic if this information is important to optimize partitions.
  - If designers are part of the same design environment, they can check out the required project files from the same source control system.
  - Otherwise, the project lead can provide a copy of the top-level project framework so that each design develops their partition within the same project framework.
- If a partition designer does not have access to the top-level project framework, the project lead can give the partition designer a Tcl script or other documentation to create the separate Quartus II project and all the assignments from the top-level design.

For details about project management scripts you can create with the Quartus II software, refer to "Generating Design Partition Scripts" on page 2–57.

If the partition designers provide the project lead with a post-synthesis **.qxp** and fitting is performed in the top-level design, integrating the design partitions should be quite easy. If you plan to develop a partition in a separate Quartus II project and integrate the optimized post-fitting results into the top-level design, use the following guidelines to improve the integration process:

- Ensure that a LogicLock region constrains the partition placement and uses only the resources allocated by the 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, turn off **Auto Global Clock and Auto Global Register Controls** under **More Settings** on the Fitter page in 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 partition, but use global routing in the top-level design.
- Use the Virtual Pin assignment to indicate pins of a partition that do not drive pins in the top-level design. This is critical when a partition 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 partition ports, such as location and timing assignments.

- When partitions are compiled independently without any information about each other, you might need to 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 partition to perform timing budgeting.

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

Generating Design Partition Scripts

In a typical team-based design flow in which designers want to optimize their design blocks independently and do not have access to a shared project framework, the project lead must perform some or all of the following tasks to ensure successful integration of the design blocks:

- Manually determine which assignments should be propagated from the top-level design to the partitions. This requires detailed knowledge of which assignments are required to set up low-level designs.
- Manually communicate the top-level assignments to the partitions. 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 team-based design. This requires examination of the logic in the partitions to determine appropriate timing constraints.
- Perform final timing closure and resource conflict avoidance in the top-level design. Because the partitions 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 partition implementation may not be available.

Design partition scripts automate the process of transferring the top-level project framework to partition designers in a flow where each design block is developed in separate Quartus II projects before being integrated into the top-level design. If the project lead cannot provide each designer with a copy of the top-level project framework, the Quartus II software provides an interface for managing resources and timing budgets in the top-level design. Design partition scripts make it easier for partition designers to implement the instructions from the project lead, and avoid conflicts between projects when integrating the partitions into the top-level design. This flow also helps to reduce the need to further optimize the designs after integration.

You can use options in the **Generate Design Partition Scripts** dialog box to choose which types of assignments you want to pass down and create in the partitions being developed in separate Quartus II projects.

For an example design scenario using design partition scripts, refer to "Enabling Designers on a Team to Optimize Independently" on page 2–40.

For step-by-step information on how to generate design partition scripts, and a description of each option that can be included in design partition scripts, refer to *Generating Design Partition Scripts for Project Management*, and *Generate Design Partition Scripts Dialog Box* in Quartus II Help.

Exporting Partitions

When partition designers achieve the design requirements in their separate Quartus II projects, each designer can export their design as a partition so it can be integrated into the top-level design by the project lead. The **Export Design Partition** dialog box, available from the Project menu, allows designers to export a design partition to a Quartus II Exported Partition File (**.qxp**) with a post-synthesis netlist, a post-fit netlist, or both. The project lead then adds the **.qxp** file to the top-level design to integrate the partition.

A designer developing a timing-critical partition or who wants to optimize their partition on their own would opt to export their completed partition with a post-fit netlist, allowing for the partition to more reliably meet timing requirements after integration. In this case, you must ensure that resources are allocated appropriately to avoid conflicts. If the placement and routing optimization can be performed in the top-level design, exporting a post-synthesis netlist allows the most flexibility in the top-level design and avoids potential placement or routing conflicts with other partitions.

When designing the partition logic to be exported into another project, you can add logic around the design block to be exported as a design partition. You can instantiate additional design components for the Quartus II project so that it matches the top-level design environment, especially in cases where you do not have access to the full top-level design project. For example, you can include a top-level PLL in the project, outside of the partition to be exported, 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 partition is complete and accurate. You can export the partition for the top-level design without any auxiliary components that are instantiated outside the partition being exported.

If your design team uses makefiles and design partition scripts, the project lead can use the **make** command with the **master_makefile** command created by the scripts to export the partitions and create **.qxp** files. When a partition has been compiled and is ready to be integrated into the top-level design, you can export the partition with option on the **Export Design Partition** dialog box, available from the Project menu.

For more information about how to export a design partition, refer to *Using a Team-Based Incremental Compilation Design Flow* in the Quartus II Help.

Viewing the Contents of a Quartus II Exported Partition File (.qxp)

The QXP report allows you to view a summary of the contents in a **.qxp** file when you open the file in the Quartus II software. The **.qxp** file is a binary file that contains compilation results so the file cannot be read in a text editor. The QXP report opens in the main Quartus II window and contains summary information including a list of the I/O ports, resource usage summary, and a list of the assignments used for the exported partition.

Integrating Partitions into the Top-Level Design

To integrate a partition developed in a separate Quartus II project into the top-level design, you can simply add the **.qxp** file as a source file in your top-level design (just like a Verilog or VHDL source file). You can also use the **Import Design Partition** dialog box to import the partition, in certain situations, described in "Advanced Importing Options" below.

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

When you use a **.qxp** file as a source file in this way, you can choose whether you want the **.qxp** file to be a partition in the top-level design. If you do not designate the **.qxp** instance as a partition, the software reuses just the post-synthesis compilation results from the **.qxp**, removes unconnected ports and unused logic just like a regular source file, and then performs placement and routing.

If you assigned the **.qxp** instance as a partition, you can set the Netlist Type in the Design Partitions Window to choose the level of results to preserve from the **.qxp** file. To preserve the placement and routing results from the exported partition, set the Netlist Type to **Post-Fit** for the **.qxp** file partition in the top-level design. If you assign the instance as a design partition, the partition boundary is preserved, as discussed in "Impact of Design Partitions on Design Optimization" on page 2–14.

Integrating Assignments from the .qxp File

The Quartus II software filters assignments from **.qxp** files to include appropriate assignments in the top-level design. The assignments in the .**qxp** file are treated like assignments made in an HDL source file, and are not listed in the Quartus II Settings File (**.qsf**) for the top-level design. Most assignments from the .**qxp** can be overridden by assignments in the top-level design.

The following subsections provide more details about specific assignment types:

Design Partition Assignments Within the Exported Partition

Design partition assignments defined within a separate Quartus II project are not added to the top-level design. All logic under the exported partition in the project hierarchy is treated as single instance 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 File (**.sdc**) in the lower-level partition project are not added to the top-level design. Ensure that the top-level design includes all of the timing requirements for the entire project.

For recommendations about managing SDC constraints for the top-level design and independent lower-level partition projects, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.

Global Assignments

The project lead should make all global project-wide assignments in the top-level design. Global assignments from the exported partition's project are not added to the top-level design. When it is possible for a particular constraint, the global assignment is converted to an instance-specific assignment for the exported design partition.

Clock settings for the Quartus II Classic Timing Analyzer are global assignments, and are not copied into the top-level design.

LogicLock Region Assignments

The project lead typically creates LogicLock region assignments in the top-level design for any lower-level partition designs where designer or IP providers plan to export post-fit information to be used in the top-level design, to help avoid placement conflicts between partitions. When you use the **.qxp** file as a source file, LogicLock constraints from the exported partition are applied in the top-level design, but will not appear in your **.qsf** file or LogicLock Regions window for you to view or edit. The LogicLock region itself is not required to constrain the partition placement in the top-level design if the Netlist Type is set to **Post-Fit**, because the netlist contains all the placement information. For information on how to control LogicLock region assignments for exported partitions, refer to the "Advanced Importing Options" on page 2–34.

Integrating Encrypted IP Cores From .qxp Files

Proper license information is required to compile encrypted IP cores. If an IP core is exported as a **.qxp** file from another Quartus II project, the top-level designer instantiating the **.qxp** file must have the correct license. The software requires 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.

Advanced Importing Options

You can use advanced options in the **Import Design Partition** dialog box to integrate a partition developed in a separate Quartus II project into the top-level design. The import process adds more control than using the **.qxp** file as a source file, and is useful only in the following circumstances:

- **If you want LogicLock regions in your top-level design (.qsf)**—If you have regions in your partitions that are not also in the top-level design, the regions will be added to your **.qsf** file during the import process.
- **If you want different settings or placement for different instantiations of the same entity**—You can control the setting import process with the advanced import options, and specify different settings for different instances of the same **.qxp** design block.

When you use the **Import Design Partition** dialog box to integrate a partition into the top-level design, the import process sets the partition's Netlist Type to **Imported** in the Design Partitions window.

After you compile the entire design, if you make changes to the place-and-route results (such as movement of an imported LogicLock region), use the **Post-Fit** Netlist Type on subsequent compilations. To discard an imported netlist and recompile from source code, you can compile the partition with the Netlist Type set to **Source File** and be sure to include the relevant source code in the top-level design. Refer to "Netlist Type for Design Partitions" on page 2–22 for details. 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 Fitter Preservation Level is set to **Placement**, but you can change it to the **Netlist Only** value.

For more information about preserving previous compilation results, refer to "Netlist Type for Design Partitions" on page 2–22 and "Fitter Preservation Level for Design Partitions" on page 2–23.

When you import a partition from a **.qxp** file, the **.qxp** file itself is not part of the top-level deign because the netlists from the file have been imported into the project database. Therefore if a new version of a **.qxp** file is exported, the top-level designer must perform another import of the new **.qxp**.

When you import a partition into a top-level design with the **Import Design Partition** dialog box, the software imports relevant assignments from the partition into the top-level design, as described for the source file integration flow in "Integrating Assignments from the .qxp File" on page 2–33. If required, you can change the way some assignments are imported, as described in the following subsections.

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. For details, refer to "Changing Partition Placement with LogicLock Changes" on page 2–47. To preserve changes made to a partition after compilation, use the **Post-Fit** Netlist Type.

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.

Advanced Import Settings

The **Advanced Import Settings** dialog box allows you to disable assignment import and specify additional options that control how assignments and regions are integrated when importing a partition into a top-level design, including how to resolve assignment conflicts.

For descriptions of the advanced import options available, refer to *Advanced Import Settings Dialog Box* in Quartus II Help.

Team-Based Design Optimization and Third-Party IP Delivery Scenarios

This section includes the following design flows with step-by-step descriptions when you have partitions being developed in other Quartus II projects, or by a third-party IP provider.

- "Using an Exported Partition to Send a Design without Including Source Files" on page 2–36
- "Creating Precompiled Design Blocks (or Hard-Wired Macros) for Reuse" on page 2–37
- "Designing in a Team-Based Environment" on page 2–39
- "Enabling Designers on a Team to Optimize Independently" on page 2–40
- "Performing Design Iterations With Lower-Level Partitions" on page 2–43

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.

Scenario 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 scenario perform the following steps to export a design block:

1. Provide the device family name to the recipient. 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 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, as described in the following:
   - 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`.
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 designer sending the design block. Instantiate the design block using the port information provided, and then incorporate the design block into a top-level design.

Add the **.qxp** file from the IP designer as a source file in your Quartus II project to replace any empty wrapper file. If you want to use just the post-synthesis information, you can choose whether you want the file to be a partition in the top-level design. To use the post-fit information from the **.qxp**, assign the instance as a design partition and set the Netlist Type to **Post-Fit**. Refer to "Creating Design Partitions" on page 2–16 and "Netlist Type for Design Partitions" on page 2–22.

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

Use this design flow to create a precompiled IP block (sometimes known as a hard-wired macro) 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 block into another design.

Scenario 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.

   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 and ensures that there are no conflicts with the top-level design logic. Reserved space also helps ensure the IP core does not affect the timing performance of other logic in the top-level design. Additionally, with a LogicLock region, you can preserve placement either absolutely or relative to the origin of the associated region. 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 design 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 will exported as an IP core.

5. Optimize the design and close timing to meet the design specifications.
6. Export the level of hierarchy for the IP core into a single **.qxp** file.
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 the **.qxp** 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. Perform Analysis and 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 2–55) with the Netlist Type set to **Empty** (refer to "Netlist Type for Design Partitions" on page 2–22).
4. You can now continue work on your part of the design and accept the IP core from the IP provider when it is ready.
5. Include the **.qxp** file from the IP provider in your project to replace the empty wrapper-file for the IP instance. Or, if you are importing multiple copies of the design block and want to import relative placement, follow these additional steps:
   a. Use the **Import** command to select each 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`.
   b. When you have multiple instances of the IP block, you can set the imported LogicLock regions to floating, or move them to a new location, and the software preserves the relative placement for each of the imported modules (relative to the origin of the LogicLock region). Routing information is preserved whenever possible. Refer to "Changing Partition Placement with LogicLock Changes" on page 2–47

   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.

6. You can control the level of results preservation with the Netlist Type. Refer to "Netlist Type for Design Partitions" on page 2–22.

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.

Designing in a Team-Based Environment

This scenario describes how to use incremental compilation in a team-based design environment where each designer has access to the top-level project framework, but wants to optimize their design in a separate Quartus II project using a bottom-up design flow before integrating their design block into the top-level design.

Scenario background: A project includes several lower-level design blocks that are developed separately by different designers and instantiated exactly once in the top-level design.

As the project lead in this scenario, perform the following steps to prepare the top-level design:

1. Create a new Quartus II project to ultimately contain the full implementation of the entire design and include a "skeleton" or framework 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.
2. Make project-wide settings. Select the device, make global assignments such as device I/O ports, define the top-level timing constraints, and make any global signal allocation constraints to specify which signals can use global routing resources.
3. 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.
4. Create LogicLock regions to create a design floorplan for each of the partitions that will be developed separately. 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.
5. Provide the top-level project framework to partition designers using one of the following procedures:
   - Allow access to the full project for all designers through a source control system. Each designer can check out the projects files as read-only and work on their blocks independently. This design flow provides each designer with the most information about the full design, which helps avoid resource conflicts and makes design integration easy.
   - Provide a copy of the top-level Quartus II project framework for each designer. You can use the **Copy Project** command on the Project menu or create a project archive.

As the designer of a lower-level design block in this scenario, design and optimize your partition in your copy of the top-level design, and then follow these steps when you have achieved the desired compilation results:

1. On the Project menu, click **Export Design Partition**.

2. In the **Export Design Partition** dialog box, choose the netlist(s) to export. You can export a Post-synthesis netlist if placement or performance preservation is not required, to provide the most flexibility for the Fitter in the top-level design. Select Post-fit netlist to preserve the placement and performance of the lower-level design block, and turn on **Export routing** to include the routing information, if required. One **.qxp** file can include both post-synthesis and post-fitting netlists.
3. Provide the **.qxp** file to the project lead.

Finally, as the project lead in this scenario, perform these steps to integrate the **.qxp** files received from designers of each partition:

1. Add the **.qxp** as a source file in the Quartus II project, to replace any empty wrapper file for the previously **Empty** partition.
2. Change the Netlist Type for the partition from **Empty** to the required level of results preservation.

Enabling Designers on a Team to Optimize Independently

This scenario describes how to use incremental compilation in a team-based design environment where designers or IP developers want to fully optimize the placement and routing of their design independently in a separate Quartus II project using a bottom-up design flow before sending the design to the project lead. In this scenario, the IP developers do not have access to the top-level project framework. This design flow requires more planning and careful resource allocation because design blocks are developed independently.

Scenario background: A project consists of several lower-level design blocks that are developed separately by different designers who do not have access to a shared top-level project framework. This scenario is similar to the "Creating Precompiled Design Blocks (or Hard-Wired Macros) for Reuse" on page 2–37 scenario, but assumes that there are several design blocks being developed independently (instead of just one IP block), and the project lead can provide some information about the design to the individual designers. If the designers have shared access to the top-level design, use the previous scenario "Designing in a Team-Based Environment" on page 2–39. As the project lead in this scenario, perform the following steps to prepare the top-level design:

1. Create a new Quartus II project to ultimately contain the full implementation of the entire design and include a "skeleton" or framework 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.
2. Make project-wide settings. Select the device, make global assignments such as device I/O ports, define the top-level timing constraints, and make any global signal constraints to specify which signals can use global routing resources.
3. 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.
4. Create LogicLock regions. 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.

5. Provide the constraints from the top-level design to partition designers using one of the following procedures:.
   - Use design partition scripts to pass constraints and generate separate Quartus II projects. On the Project menu, use the **Generate Bottom-Up Design Partition Scripts** command, or run the script generator from a Tcl or command prompt. Make changes to the default script options as required for your project. Altera recommends that you pass all the default constraints, including LogicLock regions, for all partitions and virtual pin location assignments. If partitions 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. Provide each partition designer with the Tcl file to create their project with the appropriate constraints. If you are using makefiles, provide the makefile for each partition.
   - Use documentation or manually-created scripts to pass all constraints and assignments to each partition designer.

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

If you are using makefiles with the design partition scripts, 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 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. If you are using design partition scripts, source the Tcl script provided by the Project Lead to create a project with the required settings:
   - To source the Tcl script in the Quartus II software, on the Tools menu, click **Utility Windows** to open the Tcl console. Navigate to the script's directory, and type the following command: `source <filename> r`
   - To source the Tcl script at the system command prompt, type the following command: `quartus_cdb -t <filename>.tcl r`

2. If you are not using design partition scripts, create a new Quartus II project for the subdesign, and then apply the following settings and constraints to ensure successful integration:
   - Make LogicLock region assignments and global assignments (including clock settings) as specified by the project lead.
   - Make Virtual Pin assignments for ports which represent connections to core logic instead of external device pins in the top-level design.
   - Make floorplan location assignments to the Virtual Pins so they are placed in their corresponding regions as determined by the top-level design. 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.
3. Proceed to compile and optimize the design as needed.
4. When you have achieved the desired compilation results, on the Project menu, click **Export Design Partition**.
5. In the **Export Design Partition** dialog box, choose the netlist(s) to export. You can export a Post-synthesis netlist instead if placement or performance preservation is not required, to provide the most flexibility for the Fitter in the top-level design. Select **Post-fit** to preserve the placement and performance of the lower-level design block, and turn on Export routing to include the routing information, if required. One **.qxp** file can include both post-synthesis and post-fitting netlists.
6. Provide the **.qxp** file to the project lead.

Finally, as the project lead in this scenario, perform the appropriate set of steps to import the **.qxp** files received from designers of each partition.

If you are using makefiles with the design partition scripts, perform the following steps:

1. Use the `master_makefile` command to export each 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. Add the **.qxp** as a source file in the Quartus II project, to replace any empty wrapper file for the previously Empty partition.
2. Change the Netlist Type for the partition from Empty to the required level of results preservation.

Resolving Assignment Conflicts During Integration

When integrating lower-level design blocks, the project lead may notice some assignment conflicts. This can occur, for example, if the lower-level design block 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 design by the project lead. The project lead can address these conflicts by explicitly importing the partitions into the

top-level design, and using options in the **Advanced Import Settings** dialog box, as described in "Advanced Importing Options" on page 2–34. After the project lead obtains the **.qxp** file for each lower-level design block from the other designers, use the **Import Design Partition** command on the Project menu and specify the partition in the top-level design that is represented by the lower-level design block **.qxp** file. Repeat this import process for each partition in the design. After you have imported each partition once, you can select all the design partitions and use the **Reimport using latest import files at previous locations** option to import all the files from their previous locations at one time. To address assignment 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.

If the placement of different lower-level design blocks 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 design scenario, one of the lower-level design blocks is instantiated more than once in the top-level design. The designer of the lower-level design block may want to compile and optimize the entity once under a partition, and then import the results as multiple partitions in the top-level design.

If you import multiple instances of a lower-level design block into 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 Iterations With Lower-Level Partitions

Use this flow if you re-optimize partitions exported from separate Quartus II projects by incorporating additional constraints from the integrated top-level design.

Scenario 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 this example, integration at the top level has failed because the timing requirements are not met. The timing requirements might have been met in each individual lower-level project, but critical inter-partition paths in the top-level design are causing timing requirements to fail.

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

The best way to provide top-level design information to designers of lower-level partitions is to provide the complete top-level project framework using the following steps:

1. For all partitions other than the one(s) being optimized by a designer(s) in a separate Quartus II project(s), set the netlist type to **Post-Fit.**
2. Make the top-level design directory available in a shared source control system, if possible. Otherwise, copy the entire top-level design project directory (including database files), or create a project archive including the post-compilation database.
3. Provide each partition designer with a checked-out version or copy of the top-level design.
4. The partition designers recompile their designs within the new project framework that includes the rest of the design's placement and routing information as well top-level resource allocations and assignments, and optimize as needed.
5. When the results are satisfactory and the timing requirements are met, export the updated partition as a **.qxp** file.

If this design flow is not possible, you can generate partition-specific scripts for individual designs to provide information about the top-level project framework with these steps:

1. In the top-level design, on the Project menu, click **Generate Design Partition Scripts**, or launch the script generator from Tcl or the command line.
2. If lower-level projects have already been created for each partition, you can turn off the **Create lower-level project if one does not exist** option.
3. Make additional changes to the default script options, as necessary. 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 design, 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 partition 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 partition designers recompile their designs with the new project information or assignments and optimize as needed. When the results are satisfactory and the timing requirements are met, export the updated partition as a **.qxp** file.

The project lead obtains the updated **.qxp** files from the partition designers and adds them to the top-level design. When a new **.qxp** file is added to the files list, the software will detect the change in the "source file" and use the new **.qxp** results during the next compilation. If the project uses the advanced import flow, the project lead must perform another import of the new **.qxp**.

You can now analyze the design to determine whether the timing requirements have been achieved. Because the 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 a Design Floorplan with LogicLock Regions

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 to improve the quality of results when using the incremental compilation design flow. Creating a design floorplan is not a requirement to use an incremental compilation flow, but it is 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 timing-critical partitions

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 compiled as part of the top-level design. 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). If you have a compilation result for a partitioned design with no LogicLock regions, you can use the Chip Planner with the Design Partition Planner to view the partition placement in the device floorplan. You can draw regions in the floorplan that match the general location and size of the logic in each partition. Or, initially, you can set each region with the default settings of **Auto** size and **Floating** location to allow the Quartus II software to determine the preliminary 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. Alternatively, 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 merging 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, as described in "Incremental Compilation Advisor" on page 2–20.

Creating and Manipulating LogicLock Regions

Options in the **LogicLock Regions Properties** dialog box, available from the Assignments menu, allow you to enter specific sizing and location requirements for a region. You can also view and refine the size and location of LogicLock regions in the Quartus II Chip Planner. You can select a region in the graphical interface in the Chip Planner and use handles to move or resize the region.

Options in the **Layer Settings** panel in the Chip Planner allow you to create, delete, and modify tasks to determine which objects, including LogicLock regions and design partitions, to display in the Chip Planner.

For more information about creating and viewing LogicLock regions in the LogicLock Regions window and Chip Planner, refer to *Creating and Manipulating LogicLock Regions* in Quartus II Help.

Changing Partition Placement with LogicLock Changes

When a partition is assigned to a LogicLock region as part of a design floorplan, you can modify the placement of a post-fit partition by moving the LogicLock region. As described in "What Changes Initiate a Partition's Automatic Resynthesis?" on page 2–25, most assignment changes do not initiate a recompilation of a partition if the Netlist Type specifies that Fitter results should be preserved. For example, changing a pin assignment does not initiate a 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 initiate refitting if a partition's placement is preserved with the **Post-Fit** Netlist Type setting or with **.qxp** file 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.

Taking Advantage of the Early Timing Estimator

When creating a floorplan you can take 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 without having to run a full compilation. 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.

Incremental Compilation Restrictions

This section documents restrictions and limitations that you may encounter when using incremental compilation, including interactions with other Quartus II features. Some additional restrictions apply only to exported partitions.

The following restrictions and limitations are covered in this section:

- "When Timing Performance May Not Be Preserved Exactly" on page 2–48
- "When Placement and Routing May Not Be Preserved Exactly" on page 2–48
- "Using Incremental Compilation with Quartus II Archive Files" on page 2–48
- "Formal Verification Support" on page 2–50
- "SignalProbe Pins and Engineering Change Orders" on page 2–50
- "SignalTap II Logic Analyzer in Exported Partitions" on page 2–50
- "External Logic Analyzer Interface in Exported Partitions" on page 2–51
- "Assignments Made in HDL Source Code in Exported Partitions" on page 2–51
- "Design Partition Script Limitations" on page 2–51
- "Restrictions on Megafunction Partitions" on page 2–53
- "Register Packing and Partition Boundaries" on page 2–54
- "I/O Register Packing" on page 2–54

When Timing Performance May Not Be Preserved Exactly

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.

If you want to re-use post-synthesis or post-fitting results, 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 **Incremental 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.

For new device families with advanced support, a version-compatible database might not be available. In this case, the archive will not include the compilation database. If you require the database files to reproduce the compilation results in the same Quartus II version, you can use the following command-line option to archive a full database:

```
quartus_sh --archive -use_file_set full_db [-revision <revision
name>] <project name>
```

Limitations for HardCopy Compilation and Migration Flows

HardCopy ASIC Migration Flows

Incremental compilation within a single Quartus II project 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 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).

Exporting and importing partitions 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. Importing partitions does not support this requirement. You can often use **Empty** partitions to implement behavior similar to an exported partition 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 all incremental compilation flows 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 incremental compilation design flow.

Formal Verification Support

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

SignalProbe Pins and Engineering Change Orders

ECO and SignalProbe changes are performed only during ECO and SignalProbe compilations. Other compilation flows do not preserve these netlist changes.

When incremental compilation is turned on and your design contains one or more design partitions, partition boundaries are ignored while making ECO changes and SignalProbe signal settings. However, the presence of ECO and/or SignalProbe changes does not affect partition boundaries for incremental compilation. During subsequent compilations, ECO and SignalProbe changes are not preserved regardless of the Netlist Type or Fitter Preservation setting. To recover ECO changes and SignalProbe signals, you must use the Change Manager to re-apply the ECOs after compilation.

For partitions developed independently in separate Quartus II projects, the exported netlist includes all currently saved ECO changes and SignalProbe signals. If you make any ECO or SignalProbe 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. After integrating the **.qxp** partition into the top-level design, the ECO changes will not appear in the Change Manager.

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*.

SignalTap II Logic Analyzer in Exported Partitions

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

When incremental compilation is turned on, debugging logic is added to your design incrementally and you can tap post-fitting nodes and modify triggers and configuration without recompiling the full design. 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.

If incremental compilation is turned off, the debugging logic is added to the design during Analysis and Elaboration, and you cannot tap post-fitting nodes or modify debug settings without fully compiling the design.

For design partitions that are being developed independently in separate Quartus II projects and contain the logic analyzer, when you export the partition, the Quartus II software automatically removes the SignalTap II logic analyzer and related SLD_HUB logic. You can tap any nodes in a Quartus II project, including nodes within **.qxp** partitions. Therefore, you can use the logic analyzer within the full top-level design to tap signals from the **.qxp** partition.

You can also 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 to include the logic analyzer in the top-level design.

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*.

External Logic Analyzer Interface in Exported Partitions

You can use the Logic Analyzer Interface in any project that you can compile and program into an Altera device. You cannot export a partition that uses the Logic Analyzer Interface. 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*.

Assignments Made in HDL Source Code in Exported Partitions

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.

Design Partition Script Limitations

The Quartus II software has some additional limitations related to the design partition scripts described in "Generating Design Partition Scripts" on page 2–31.

Warnings About Extra Clocks Due to Design Partition Scripts

The generated scripts include applicable clock information for all clock signals in the top-level design. 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 Design Partition Scripts

After you have compiled a design using TimeQuest constraints, and the timing assignments option is turned on in the scripts, a separate Tcl script is generated to create an **.sdc** file for each lower-level project. This script includes 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. For suggestions on managing SDC constraints between top-level and lower-level projects, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.

Wildcard Support in 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 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.

If the lower-level design uses the top-level project framework from the project lead, the design will have all the required information about the clock and PLL settings. Otherwise, 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. Alternatively, you can 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.

Pin Assignments for GXB and LVDS Blocks in 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 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 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.

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.

For more information and examples of cross-partition boundary I/O packing, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* 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 *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. To access command-line help in a browser window, type the following command at the command prompt:

```
quartus_sh --qhelp ↵
```

For more information about Tcl 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*.

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
- `IMPORTED`—Imported
- `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

To preserve high-speed optimization for tiles contained within the selected partition, use the following command:

- `PARTITION_PRESERVE_HIGH_SPEED_TILES`

To specify that the software should use the specified netlist and ignore source file changes, even if the source file has changed since the netlist was created, use the following command:

```
set_global_assignment -name
PARTITION_IGNORE_SOURCE_FILE_CHANGES ON -section_id "<partition
name>".
```

For details about these partition properties, refer to "Specifying the Level of Results Preservation for Subsequent Compilations" on page 2–21.

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 2–4.

Table 2–4. 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 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.

For more information about Tcl options and their formats, refer to *Generate Design Partition Scripts Dialog Box* in Quartus II Help.

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 2–5.

Table 2–5. Options for Generating Design Partition Scripts

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
`--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 2–5:

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

Exporting a Partition

You can use the Tcl command: `export_partition` from the `::quartus::incremental_compilation` package to export a design partition with a Tcl script.

In the following example, the `export_partition` command exports the partition, <*`my_partition`*> to a **.qxp** file named, <*`top.qxp`*>. The **.qxp** file contains both a post-synthesis and post-fit netlist, with routing:

```
package require ::quartus::incremental_compilation
project_open my_design
export_partition -partition my_partition -qxp top.qxp -post_fit
on -post_synth on -routing on
project_close
```

The following describes the available options for the `export_partition` command.

```
export_partition [-h | -help] [-long_help] -partition
<partition name> -qxp <.qxp filename> [-post_fit <on|off>]
[-post_synth <on|off>] [-routing <on|off>]
```

The `-post_synth` and `-post_fit` options control the type of netlist to export. The `-routing` option controls whether routing should be included in the **.qxp** file, and can only be enabled if a post-fit netlist is exported. The values `<on|off>` are supported for the `-post_synth` and `-post_fit` and `-routing` options.

Importing a Partition into the Top-Level Design

If you want to use advanced import options, instead of including a qxp as a source file in your 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 team-based incremental compilation 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 team-based incremental compilation design flow, the **Generate Design Partition Scripts** feature can write makefiles that automatically export lower-level design partitions and import them into the top-level design whenever design files change.

Scripting and Command-Line Application Examples

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 "Scripting and Command-Line Application Examples" on page 2–60. 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.

Document Revision History

Table 2–6 shows the revision history for this chapter.

Table 2–6. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Removed the explanation of the "bottom-up design flow" where designers work completely independently, and replaced with Altera's recommendations for team-based environments where partitions are developed in the same top-level project framework, plus an explanation of the bottom-up process for including independent partitions from third-party IP designers. ■ Expanded the Merge command explanation to explain how it now accommodates cross-partition boundary optimizations. ■ Restructured Altera recommendations for when to use a floorplan. ■ Added "Viewing the Contents of a Quartus II Exported Partition File (.qxp)" on page 2–32 section. ■ Reorganized chapter to make design flow scenarios more visible; integrated into various sections rather than at the end of the chapter.
October 2009	9.1.0	■ Redefined the bottom-up design flow as team-based and reorganized previous design flow examples to include steps on how to pass top-level design information to lower-level designers. ■ Moved SDC Constraints from Lower-Level Partitions section to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*. ■ Reorganized the "Conclusion" on page 2–61 section. ■ Removed HardCopy APEX and HardCopy Stratix Devices section.
March 2009	9.0.0	■ Split up Netlist Types table ■ Moved "Team-Based Incremental Compilation Design Flow" into the "Including or Integrating partitions into the Top-Level Design" section. ■ Added new section "Including or Integrating Partitions into the Top-Level Design". ■ Removed "Exporting a Lower-Level Partition that Uses a JTAG Feature" restriction ■ Other edits throughout chapter
November 2008	8.1.0	■ 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

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

Take an online survey to provide feedback about this handbook chapter.

3. Quartus II Support for HardCopy Series Devices

QII51004-10.0.0

This chapter describes Quartus® II support for HardCopy® series devices.

Altera® HardCopy ASICs are the lowest risk, lowest total cost ASICs. The HardCopy system development methodology offers fast time-to-market, low risk, and with 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.

This chapter discusses the following topics:

- "HardCopy Development Flow" on page 3–2
- "HardCopy Utilities Menu" on page 3–6
- "HardCopy Companion Device Selection" on page 3–10
- "HardCopy Device Resource Guide" on page 3–11
- "HardCopy Recommended Settings in the Quartus II Software" on page 3–12
- "HardCopy Design Readiness Check" on page 3–16
- "Performing ECOs with Quartus II Engineering Change Management with the Chip Planner" on page 3–20
- "Formal Verification of FPGA and HardCopy Revisions" on page 3–24

For more information about HardCopy series devices, refer to the respective HardCopy device handbook, which is available on the Literature page of the Altera website at www.altera.com.

HardCopy Series Design Benefits

Designing with HardCopy ASICs offers the following 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 and HardCopy designs 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, which are described in further detail later in this chapter, for HardCopy series device planning:

- **HardCopy Companion Device Assignment**—Identifies compatible HardCopy series devices for use with the FPGA prototype device currently selected.

  This feature constrains the pins of your FPGA prototype, making it compatible with your HardCopy device. The feature also constrains the correct resources available for the HardCopy device, ensuring the compatibility of your FPGA design. You must compile the design targeting the HardCopy device to ensure that the design fits, routes, and meets timing requirements.
- **HardCopy Utilities**—The HardCopy Utilities menu provides a variety of functions to create or overwrite HardCopy companion revisions, set current revisions, and compare revisions for equivalency.
- **HardCopy Advisor**—The HardCopy Advisor guides you through the steps necessary to successfully submit a HardCopy design to the Altera HardCopy Design Center.
- **HardCopy Floorplan**—The Chip Planner can show a preliminary floorplan view of your HardCopy design's Fitter placement results.
- **HardCopy Device Preliminary Timing**—The TimeQuest Timing Analyzer performs a timing analysis of HardCopy devices based on preliminary timing models and Fitter placements. Final timing results for HardCopy devices are provided by the Altera HardCopy Design Center.
- **HardCopy Design Readiness Check**—The Quartus II software 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 the Altera 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 the Altera HardCopy Design Center.
- **Formal Verification**—Cadence Encounter Conformal software performs formal verification between the source RTL design files and post-compilation gate-level netlist from a HardCopy design.

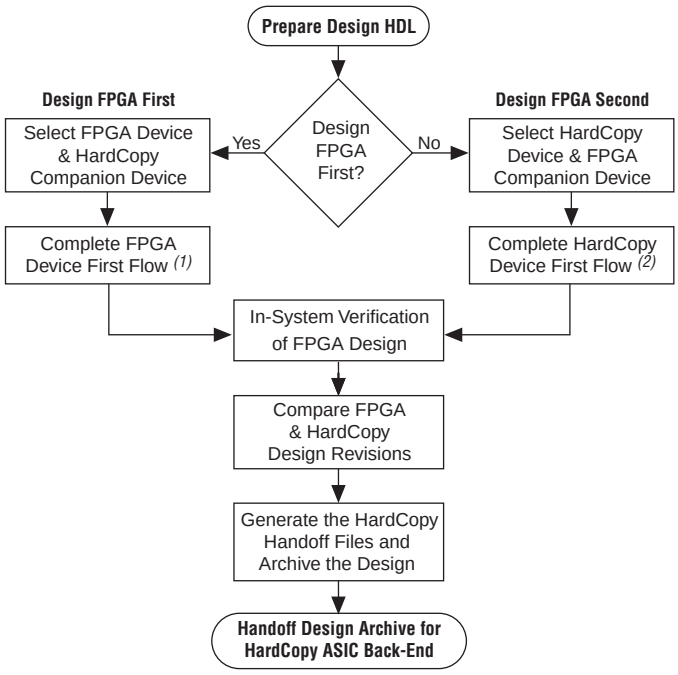
HardCopy Development Flow

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 3–1. The added features in the HardCopy Utilities menu help you complete your HardCopy design for submission to the Altera HardCopy Design Center for back-end implementation.

Figure 3–1. HardCopy Flow in Quartus II Software



Notes to Figure 3–1:

(1) Refer to Figure 3–2 on page 3–4 for an expanded description of this process.

(2) Refer to Figure 3–3 on page 3–6 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.

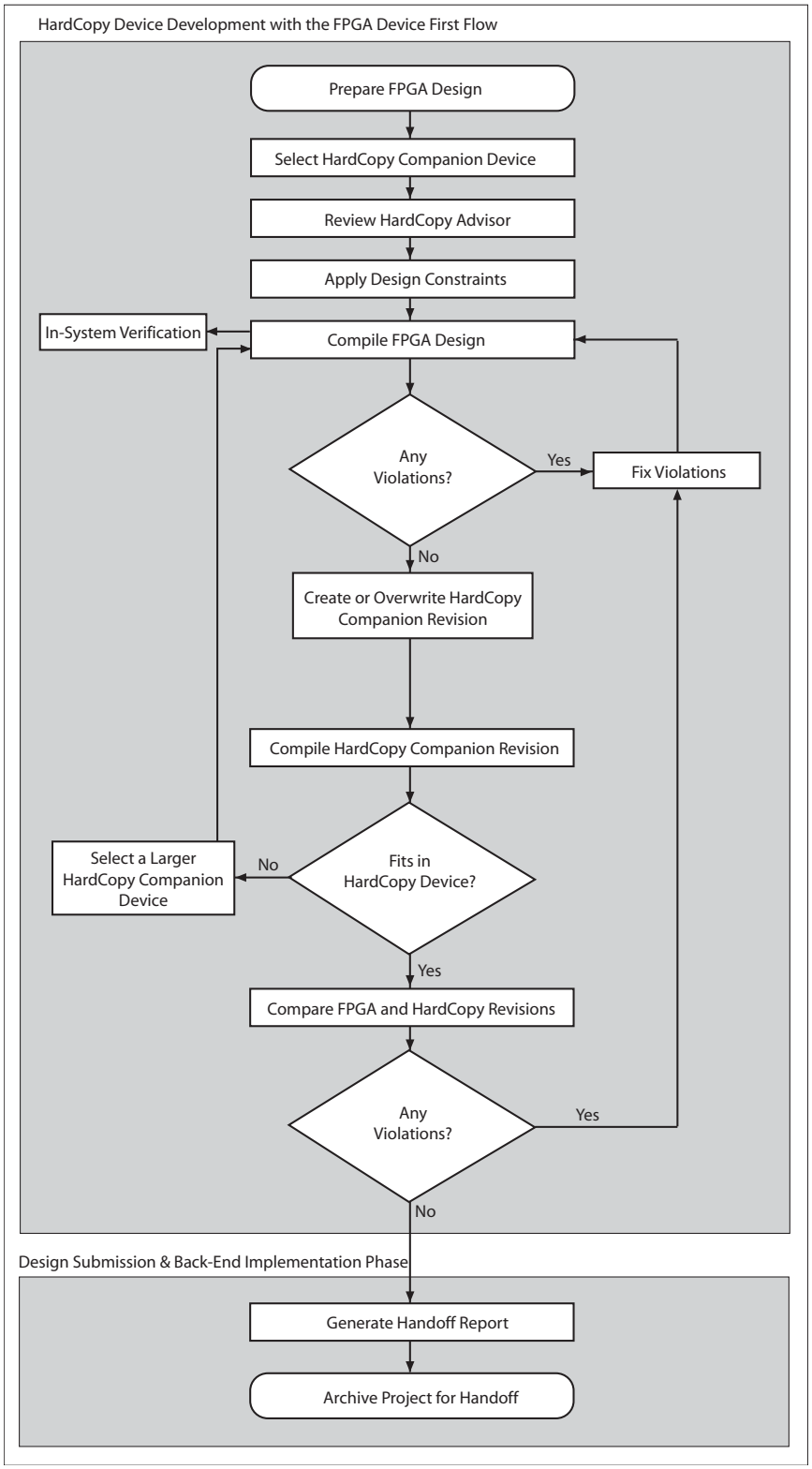
Designing the FPGA First

The HardCopy FPGA first flow development flow begins with seamless FPGA prototyping and is identical to the traditional FPGA design flow; plus a few additional tasks necessary 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 3–2 provides an overview of the development process for designing with an FPGA first and creating a HardCopy companion device second.

Figure 3–2. 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 to a HardCopy device.

During the early stages of the design process, selecting the right HardCopy device may be difficult. The HardCopy Device Resource Guide can assist you in the selection process. 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 device selections have been finalized, perform the following tasks:

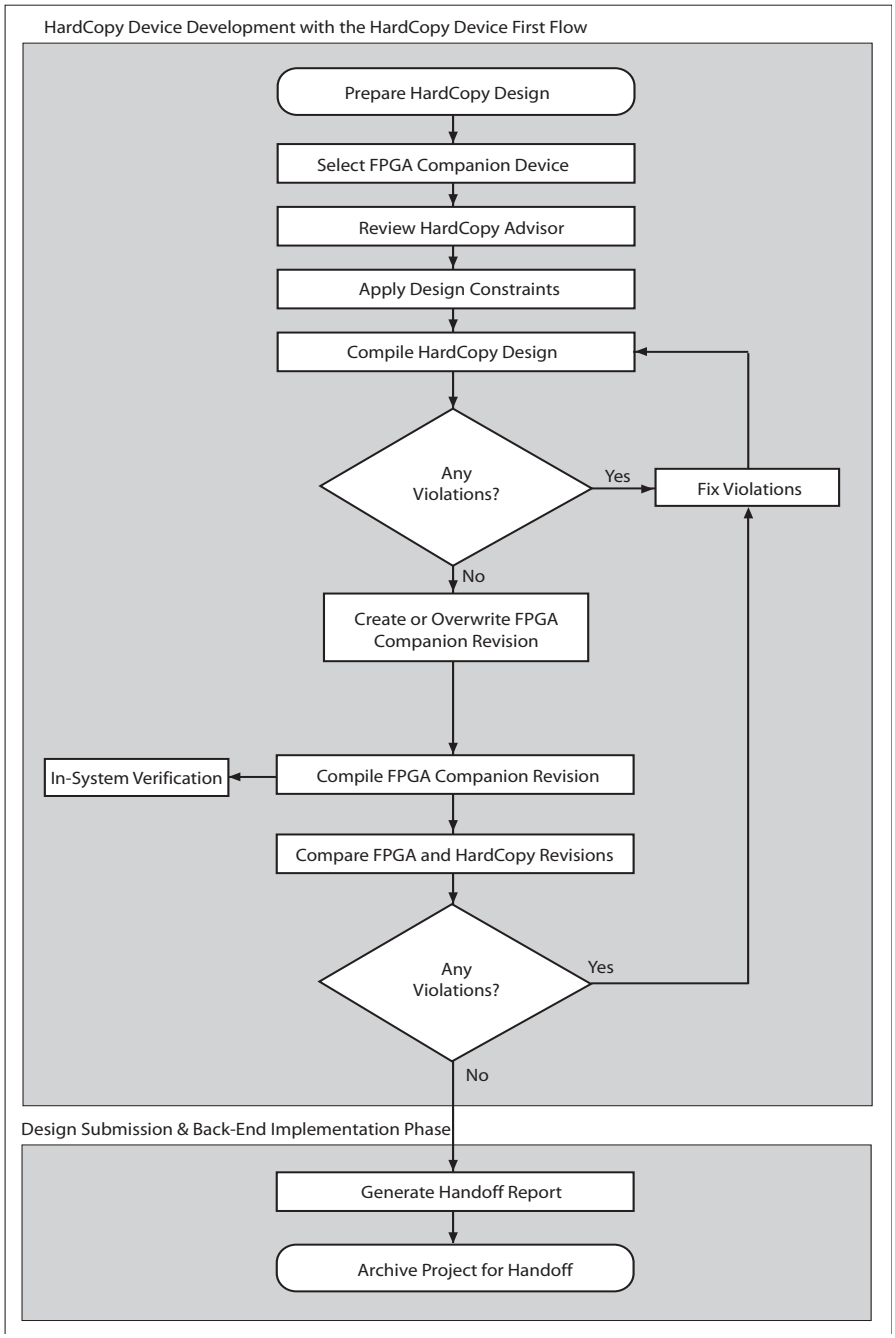
- Review the HardCopy Advisor for required and recommended tasks
- Enable the Design Assistant to run during compilation
- Add timing and location assignments
- Compile your FPGA design
- Create your HardCopy companion revision
- Compile your design for the HardCopy companion device
- Compare the HardCopy 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 the Altera 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, which is available on the Literature page of the Altera website at www.altera.com.

Designing the HardCopy Device First

After you select an initial HardCopy ASIC device, you can design your HardCopy device first, and then create your FPGA prototype second. This approach is preferred when using the HardCopy device 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 FPGA first flow development flow, but you begin the design with a different initial device family instead. The remaining tasks to complete your design for both the FPGA and HardCopy devices roughly follow the same process (Figure 3–3). 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 3–3. Designing HardCopy Device First Flow



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 overwrite 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 the Altera HardCopy Design Center
- Start the HardCopy Design Readiness Check feature
- Track your design progress using the HardCopy Advisor

Each **HardCopy Utilities** feature is summarized in Table 3–1. The process for using each of these features is explained in the following sections.

Table 3–1. HardCopy Utilities Menu Options (Part 1 of 2)

Menu	Description	Applicable Design Revision	Restrictions
Create/Overwrite HardCopy Companion Revision	Creates a new companion revision or overwrites an existing companion revision for your FPGA and HardCopy design	FPGA prototype design and HardCopy companion revision	■ The **Auto device selected by the Fitter** option must be turned off ■ An FPGA device and a HardCopy companion device must be set
Set Current HardCopy Companion Revision	Specifies the companion revision to associate with the current design revision	FPGA prototype design and HardCopy companion revision	A 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	Both revisions must be compiled
Generate HardCopy Handoff Report	Generates a report containing important design information files and messages generated by the Quartus II Compiler	FPGA prototype design and HardCopy companion revision	■ Both revisions must be compiled ■ The **Compare HardCopy Companion Revisions** command must be successfully run
Archive HardCopy Handoff Files	Generates a Quartus II Archive File (**.qar**) specifically for submitting the design to the Altera HardCopy Design Center	HardCopy companion revision	■ Both revisions must be compiled ■ The **Compare HardCopy Companion Revisions** command must be successfully run ■ The **Generate HardCopy Handoff Report** command must be successfully run

Table 3–1. HardCopy Utilities Menu Options (Part 2 of 2)

Menu	Description	Applicable Design Revision	Restrictions
Start 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
HardCopy Advisor	Opens the HardCopy Advisor, which walks you through the process of creating a HardCopy project	FPGA prototype design and HardCopy companion revision	None

Companion Revisions

You can create multiple design revisions for both the FPGA and the HardCopy device. For example, if your initial FPGA revision is named *top* and the corresponding HardCopy revision is named *top_hc*, you could create another FPGA revision, named *top_fpga*, and the corresponding HardCopy revision would be named *top_fpga_hc*. The Quartus II software creates specific HardCopy design revisions of the project in conjunction with the primary project revisions. These parallel design revisions for HardCopy devices are called companion revisions.

Although you can create multiple design revisions, Altera recommends that you maintain only one FPGA revision once you create the HardCopy companion revision.

Once you have successfully compiled your FPGA prototype, you can create and compile the HardCopy companion revision of your design.

You can associate only one FPGA revision to one HardCopy companion revision. If you create more than one revision or companion revision, set the current companion for the revision you are working on.

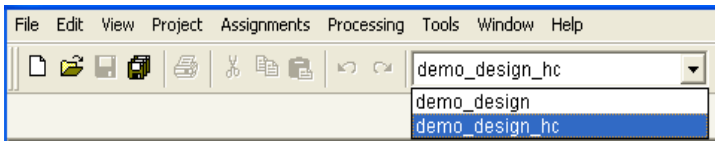
For more information about creating or setting a companion revision in the Quartus II software, refer to *Migrating a Design to a HardCopy or FPGA Device* in Quartus II Help.

Compiling the HardCopy Companion Revision

You can compile your HardCopy design with preliminary timing information in the Quartus II software. 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 the degree of 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 the Altera 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 3–4) or from the **Revisions** list. Compile the HardCopy companion revision. After you compile your design in the Quartus II software, you can perform a comparison check of the HardCopy companion revision to the FPGA prototype revision.

Figure 3–4. 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 that is 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.

The Comparison Revision Summary in the Compilation report identifies where assignments were changed between revisions or if there is a change in the logic resource count due to different compilation settings.

For more information about comparing companion revisions in the Quartus II software, refer to *Migrating a Design to a HardCopy or FPGA Device* in Quartus II Help.

Generating a HardCopy Handoff Report

To submit a design to the Altera HardCopy Design Center, you must generate a HardCopy Handoff report, which provides important information about the design that you want the Altera HardCopy Design Center to review.

After you generate the HardCopy Handoff report, you can archive the design using the **Archive HardCopy Handoff Files** command, which is described in "Archiving HardCopy Handoff Files".

For more information about the **Generate HardCopy Handoff Report** command in the Quartus II software, refer to *Generate HardCopy Handoff Report Command* in Quartus II Help.

Archiving HardCopy Handoff Files

The last step in the HardCopy design methodology is to archive the HardCopy project for submission to the Altera HardCopy Design Center for HardCopy back-end implementation. The **Archive HardCopy Handoff** command creates a unique **.qar** file, which is different 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 the Altera HardCopy Design Center.

For more information about the **Archive HardCopy Handoff Files** command in the Quartus II software, refer to *Archive HardCopy Handoff Files Command* in Quartus II Help.

HardCopy Advisor

The HardCopy Advisor provides a list of tasks to help guide you through the development of your FPGA prototype and your HardCopy design. The following tasks highlight the checkpoints that the HardCopy Advisor reviews, including 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 them to Altera.

When your design uses the FPGA as your starting point, Altera recommends following the HardCopy 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. Then follow the HardCopy Advisor steps shown in that revision until completion and then submit the design to Altera for the HardCopy back-end implementation process.

For more information about the HardCopy Advisor in the Quartus II software, refer to *About the HardCopy Advisor* in Quartus II Help.

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 select your HardCopy companion device, on the Assignments menu, click **Device** 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.

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 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 according to 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. Refer to Table 3–2 for an explanation of the color codes.

Use this report to determine which HardCopy device is a potential candidate for your design. The HardCopy and FPGA device package must be compatible. 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 specific HardCopy device.

Table 3–2. HardCopy Device Resource Guide Color Legend

Color	Package Resource *(1)*	Device Resources
Green (High)	The design can map to the HardCopy package and 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 still must compile the HardCopy revision to ensure 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 3–2:

(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 the Altera HardCopy Design Center after the HardCopy back-end implementation process is complete.

For more information about the HardCopy device resources, refer to the respective HardCopy series device handbook, which is available on the Literature page of the Altera website at www.altera.com.

The report example in Figure 3–5 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 3–5 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 3–5. HardCopy Device Resource Guide with Target Migration Enabled

HardCopy Device Resource Guide

Color Legend:
-- Green:
-- Package Resource: The HardCopy device package can be migrated from the selected FPGA device 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 Recommended Settings in the Quartus II Software

The HardCopy development flow involves additional planning and preparation in the Quartus II software when compared to a standard FPGA design. Additional planning and preparation is required because you are developing your design for implementation 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.

Limit DSP and RAM to HardCopy Device Resources

The **Limit DSP & RAM to HardCopy device resources** option in the **Device** dialog box maintains compatibility between the FPGA and HardCopy devices by ensuring that your design does not use resources in the FPGA device that are not available in the selected HardCopy device or vice versa.

For more information about the **Limit DSP & RAM to HardCopy device resources** option in the Quartus II software, refer to *Device Dialog Box* in Quartus II Help.

Enabling Design Assistant to Run During Compile

You must use the Design Assistant in the Quartus II software to check all HardCopy designs for design rule violations before submitting the designs to the Altera 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 review the violations you must fix or waive.

For more information about the Design Assistant and its rules in the Quartus II software, refer to *About the Design Assistant* in Quartus II Help.

Timing Settings

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. The TimeQuest Timing Analyzer is the required timing analysis tool for all designs. The Classic Timing Analyzer is no longer supported, and the Altera 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.

For more information about the TimeQuest Timing Analyzer, refer to the *TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* and *About TimeQuest Timing Analysis* in Quartus II Help.

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 *TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

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 of the clock uncertainties. 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 is 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] [-add]
\[-overwrite]
```

where the arguments are listed in Table 3–3:

Table 3–3. Arguments for derive_clock_uncertainty

Option	Description
`-h | -help`	Short help
`-long_help`	Long help with examples and possible return values
`-add`	Adds results user-defined clock uncertainty assignments
`-overwrite`	Overwrites previously performed clock uncertainty assignments

When the `derive_clock_uncertainty` command 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 *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. The Altera HardCopy Design Center does not accept designs that do not have clock uncertainty constraints 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

The physical synthesis optimizations performed in the FPGA device are passed to 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.

The **Effort level** on the **Physical Synthesis Optimizations** page for HardCopy III and HardCopy IV devices is limited to **Fast** only because the performance gain achieved compared to the compile time is very limited.

For more information about setting physical synthesis optimizations for the FPGA revision of the designs in the Quartus II software, refer to *Setting up and Running the Fitter* in Quartus II Help.

LogicLock Regions

LogicLock regions are flexible floorplan location constraints that help you place logic on the target device. You can use LogicLock regions in FPGA designs targeted to HardCopy devices, which are also passed on to the HardCopy companion revision. LogicLock regions in HardCopy devices cannot have their properties set to **Auto Size**. Although, floating LogicLock regions are supported, Altera recommends that you do not use floating LogicLock regions for HardCopy devices, because floating LogicLock regions may affect the design's ability to meet timing closure. 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 create in the HardCopy device before compiling the design.

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*.

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 *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Incremental Compilation

Quartus II incremental compilation within a single Quartus II project is supported for the base family in HardCopy migration flows for both the FPGA first and HardCopy first flows. Exporting and importing partitions is not supported in HardCopy ASIC or FPGA device compilations when there is a migration device setting.

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* and *About Incremental Compilation* in Quartus II Help.

HardCopy Design Readiness Check

The HardCopy Design Readiness Check (HCDRC) is available as one of the processing steps in the default compilation of either the FPGA first or the HardCopy first flow. This feature checks issues that must be addressed prior to handing off the HardCopy design to the Altera HardCopy Design Center for the HardCopy back-end process. This is different from the user-driven approach in the HardCopy Advisor, in which you must manually open the Advisor to check for any violations.

The checks performed in the HCDRC for the Quartus II software include I/Os, PLL, RAM, ALTGX, and settings checks (**Global Setting**, **Instance Setting**, and **Operating Setting**).

Turning the HardCopy Design Readiness Check On and Off

The HCDRC is turned on by default, but can be turned on or off 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 in the **More Compilation Process Settings** dialog box.

Setting Check

The Setting Check report lists the results of the setting checks from the Handoff report. The Setting Check report consists of the following sections.

Summary

The Summary section displays the number of settings that do not meet 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 Global Setting section displays recommendations for global settings only. Global settings that currently have a different value than the recommended value are highlighted in red.

Instance Setting

The Instance Setting section is identical to the Global Setting section, but checks only for instances assignments.

Operating Setting

The Operating Setting section reports checks related to the recommended operating settings for the FPGA and the HardCopy device.

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 core voltage.

The Setting Check reports also include checking for illegal assignments in the HardCopy design flow. The illegal assignments checks are shown in Example 3–1.

Example 3–1. 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 3–1:

(1) Refer to the section "RAM Usage Check" on page 3–19.
(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 device, 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 the HardCopy device. 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. Pins that are missing location assignments are listed in the Missing Pin Location Assignment report.

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

The PLL Usage Check report lists PLL usage requirements and violations checks.

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.
```

PLL elements that do not have PLL reconfiguration are listed in a table.

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 lists all PLLs that are in No Compensation operating mode. This setting is not recommended for a design migrating to a HardCopy device because of 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 is set either in Normal Compensation mode or Source Synchronous mode, a warning message is issued 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, RAM blocks power up uninitialized. During the RAM Usage Check, the HCDRC tool checks for RAMs that are initialized using a Memory Initialization File (**.mif**). Any RAM with 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

The Assembler, which is the module in the Compiler, allows you to write an FPGA programming file with an initialized checkerboard pattern for memory contents of M4K memory blocks for the FPGA revision. You should not use this option in a FPGA revision used to migrate to the HardCopy revision, because it creates irreconcilable revision differences between the FPGA and HardCopy designs since the HardCopy handoff cannot physically have any initialized memory content. Use this option only on a parallel copy of your compiled FPGA design that you wish to test on your board.

To create a programming file with an initialized checkerboard pattern, perform the following steps:

1. Compile your completed FPGA design revision to use for prototype testing. This is the revision you should eventually use to create your HardCopy companion revision.
2. Create the HardCopy companion revision.

3. Compile, compare, and generate the hand off archive files for your design.
4. Switch back to your FPGA revision, and on the Project menu, click **Revisions**, and then click **Create** in the **Revisions** dialog box.
5. In the **Create Revision** dialog box, type a a revision name in the **Revision name** box and turn on **Copy database** and **Set as current revision**. This step copies your FPGA revision and sets the new revision as the current open revision in the Quartus II software.
6. On the Assignments menu, click **Settings**, and then click **Assembler** in the **Category** list. Turn on **Use checkered pattern as uninitialized RAM content** on the **Assembler** page, or edit the revision **.qsf** file and add the following line:

   ```
   set_global_assignment -name
   USE_CHECKERED_PATTERN_AS_UNINITIALIZED_RAM_CONTENT ON
   ```

7. Run the Assembler in the FPGA revision to generate a new programming file for your FPGA.
8. 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 the checkerboard pattern is used for testing, the patterns written into the RAM blocks for the new programming file may not detect all cases of zero-initialized RAM content dependencies. Some designs may detect only one bit as zero (for example, the LSB of a memory word), so this method may not detect all cases. This checkerboard pattern test will detect when a full RAM word line is expected as zeros at startup.

ALTGX Usage Check

Beginning in the Quartus II software version 10.0, the ALTGX Usage Check has been added to the HCDRC to perform checks on ALTGX instance usage for designs targeting Stratix IV GX and HardCopy IV GX devices.

The HCDRC tool checks all the ALTGX instances that are initialized in the design for connectivity with the ALTGX_RECONFIG instance. For ALTGX instances that do not connect to an ALTGX_RECONFIG instance, the following warning message is generated with the respective instance HSSI_CMU atom name.

```
ALTGX megafunctions do not have ALTGX_RECONFIG megafunctions
connected. Altera recommends connecting ALTGX_RECONFIG
megafunction to each ALTGX megafunction when migrating your
designs to HardCopy devices.
```

Performing ECOs with Quartus II Engineering Change Management with the Chip Planner

As designs grow 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 process may become difficult to manage, because ECOs are often implemented as last minute changes to your design.

With the Altera Chip Planner, 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 types of migration scenarios:

- One-to-one changes, which are changes that can be implemented on each architecture—FPGA and HardCopy.
- Changes that must be implemented differently on the two architectures to achieve the same result.
- 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 include 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 types of changes can be implemented identically on both architectures.

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)

If such changes are exported to Tcl, directly reapplying the generated Tcl script (with a minor text edit) on the companion revision implements the appropriate changes as described in the following steps:

1. Export the changes from the Change Manager to Tcl.
2. Open the generated Tcl script, change the line `project_open <project> -revision <revision>` to refer to the appropriate companion revision.
3. Apply the Tcl script to the companion revision.

Migrating Changes that Must be Implemented Differently

Some changes must be implemented differently on the two architectures, such as changes affecting the logic of the design. Examples include LUTMASK changes, LC_COMB/HSADDER creation and deletion, connectivity changes not covered in the previous section, and 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 3–4 summarizes suggested implementation of various changes.

Table 3–4. Implementation Suggestions for Changes that Must Be Implemented Differently

Change Type	Suggested Implementation
LUTMASK changes	Because a single FPGA atom can require multiple HardCopy atoms to implement, you may need to change multiple HardCopy atoms to implement the change, including adding or modifying connectivity.
Make/Delete LC_COMB	If you are using an 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 the HardCopy device. Additionally, the placement of the LC_COMB cell has no meaning in the companion revision because the underlying resources are different.
Make/Delete LC_FF	Basic creation and deletion is the same on both architectures; however, similar to LC_COMB creation and deletion, the location of an LC_FF in a HardCopy and FPGA revision do not translate.
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 are incompatible and cannot be implemented in both architectures. The best example of this incompatibility occurs when moving logic in a design; because the logic fabric is different between the two architectures, locations in the FPGA are not compatible 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 comparison such that the design can be submitted to the Altera HardCopy Design Center.

Preparing the Revisions

The general procedure for migrating changes between devices is the same, whether migrating from the FPGA to HardCopy device or vice versa. The major steps are described below:

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. Run the **Compare HardCopy Companion Revisions** command. Both revisions should pass the revision comparison.

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 described below:

1. Make changes in one revision using the Chip Planner tools (Chip Planner, Resource Property Editor, and Change Manager), and then verify and export these changes by following these steps:
   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 Revisions** 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 the 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, use the **Compare HardCopy Revisions** command in the Quartus II software 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 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 compilation of your FPGA revision, perform the following steps:

1. On the Assignments menu, click **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 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, you can run the **Compare HardCopy Revisions** command to ensure that the HardCopy implementation matches the FPGA.

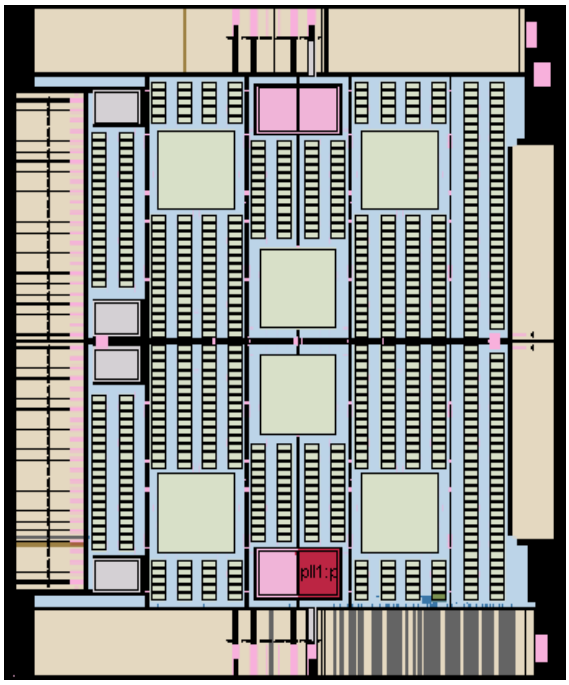
For more information about using the Cadence Encounter Conformal verification software, refer to the *Cadence Encounter Conformal Support* chapter in volume 3 of the *Quartus II Handbook*.

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 (from the View menu of the Chip Planner). This is useful in analyzing densely packed areas of your floorplan that can reduce the peak performance of your design. The Altera HardCopy Design Center verifies final HCell macro timing and placement to guarantee timing closure is achieved.

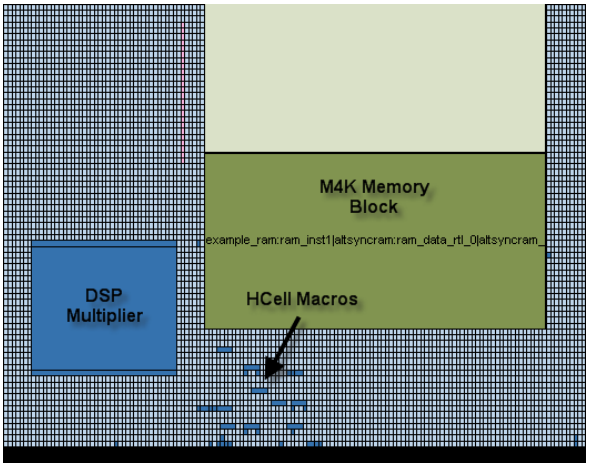
Figure 3–6 shows an example of the HC230F1020 device floorplan.

Figure 3–6. 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 3–7.

Figure 3–7. Close-Up View of Floorplan



The Altera 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 the Altera HardCopy Design Center process, refer to the respective HardCopy series device handbook, which is available on the Literature page of the Altera website at www.altera.com.

Document Revision History

Table 3–5 shows the revision history for this chapter.

Table 3–5. Document Revision History

Date	Version	Changes Made
July 2010	v10.0.0	■ Added new section"ALTGX Usage Check" on page 3–20 ■ Updated "LogicLock Regions" on page 3–15 to reflect updated companion revision support ■ Updated "Incremental Compilation" on page 3–16 to reflect updated companion revision support ■ Linked sections throughout the chapter to Quartus II Help ■ Removed "Referenced Documents"
November 2009	v9.1.0	■ Removed HardCopy Stratix legacy support information ■ Updated "Physical Synthesis Optimization" on page 3–15 ■ Updated "Quartus II Software Features Supported for HardCopy Designs" on page 3–15 ■ Updated "Referenced Documents" ■ Updated the tables and figures for HardCopy Series devices
March 2009	v9.0.0	■ Updated "RAM Usage Check" on page 3–19 ■ Updated "Referenced Documents"
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"
May 2008	v8.0.0	■ Updated "RAM Usage Check" on page 3–19 ■ Updated "Referenced Documents"

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

Take an online survey to provide feedback about this handbook chapter.

4. Quartus II Design Separation Flow

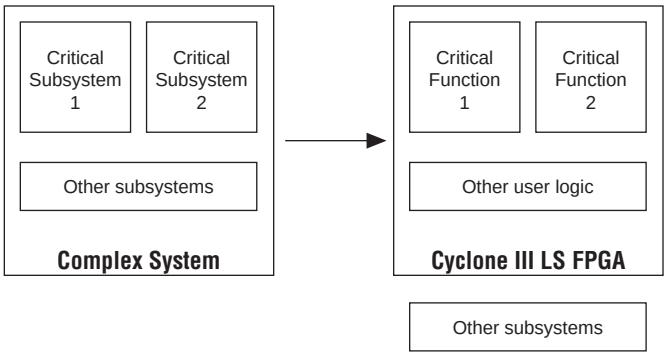
QII51019-10.0.0

This chapter contains rules and guidelines for creating a floorplan with the design separation flow, and assumes familiarity with the Quartus® II incremental compilation flow and floorplanning with the LogicLock™ feature.

The basic principle of a secure and reliable system is that critical subsystems in the design have physical and functional independence. Systems with redundancy require physical independence to ensure fault isolation—that a failure or corruption of any single subsystem will not adversely affect any other part of the system. Furthermore, should errors occur, physical independence simplifies analysis by allowing developers to evaluate each subsystem separately.

Traditionally, systems that require redundancy implement critical IP structures using multiple devices. The Quartus II design separation flow, used in Cyclone® III LS devices, provides the ability to design physically independent structures on a single device. This functionality allows system designers to achieve a higher level of integration on a single FPGA, and alleviates increasingly strict Size Weight and Power (SWaP) requirements. Figure 4–1 illustrates this concept.

Figure 4–1. Achieving Higher Level Integration on a Single Cyclone III LS Device



The Quartus II design separation flow introduces a secured region to the compilation flow. When implemented in Cyclone III LS devices, a secured region provides physical independence through controlled routing and a boundary of unused resources. Multiple secured regions can be created on a single Cyclone III LS device without unwanted mutual interference. By restricting routing resources and providing a physical guard band of unused logic array blocks (LABs), faults or unintended signals originating in one secured region are prevented from adversely affecting any other design block on the device.

The Quartus II design separation flow incorporates the incremental compilation and LogicLock features to floorplan a design by adding the secured region constraint. The following three chapters in the *Quartus II Handbook* are companion references to this chapter:

- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design*—Describes the Quartus II incremental compilation flow

- *Best Practices for Incremental Compilation Partitions and Floorplan Assignments*—Contains guidelines for using the incremental compilation flow and creating a design floorplan
- *Analyzing and Optimizing the Design Floorplan*—Describes various attributes associated with LogicLock location constraints and introduces the Chip Planner for creating and modifying a floorplan

Design Flow Overview

The Quartus II design separation flow isolates design partitions into physically independent regions, which facilitates fault isolation in the design. Each physically independent region is called a secured region in the design separation flow.

The design separation flow is similar to the incremental compilation flow; plus additional steps required to implement design separation flow features. This section provides an overview of the steps to implement the design separation flow.

First, to prepare your design for implementation of the design separation flow, you set up your design hierarchy for secure partitioning, and then run analysis and elaboration.

Then, for each design entity that requires physical independence, you create a logical design partition for each design entity and assign the design partition to a LogicLock location constraint. You can partition logic using guidelines from the incremental compilation flow and floorplan secured regions using LogicLock region location assignments.

Next, during the floorplanning stage, you create secured regions by confining all routing to the LogicLock region by applying a security attribute constraint. When you apply this attribute, a boundary consisting of unused LABs, known as a fence, is created around the LogicLock region.

You then create security routing interfaces by applying an attribute to the LogicLock region. Signals may only enter or exit a secured region through a security routing interface.

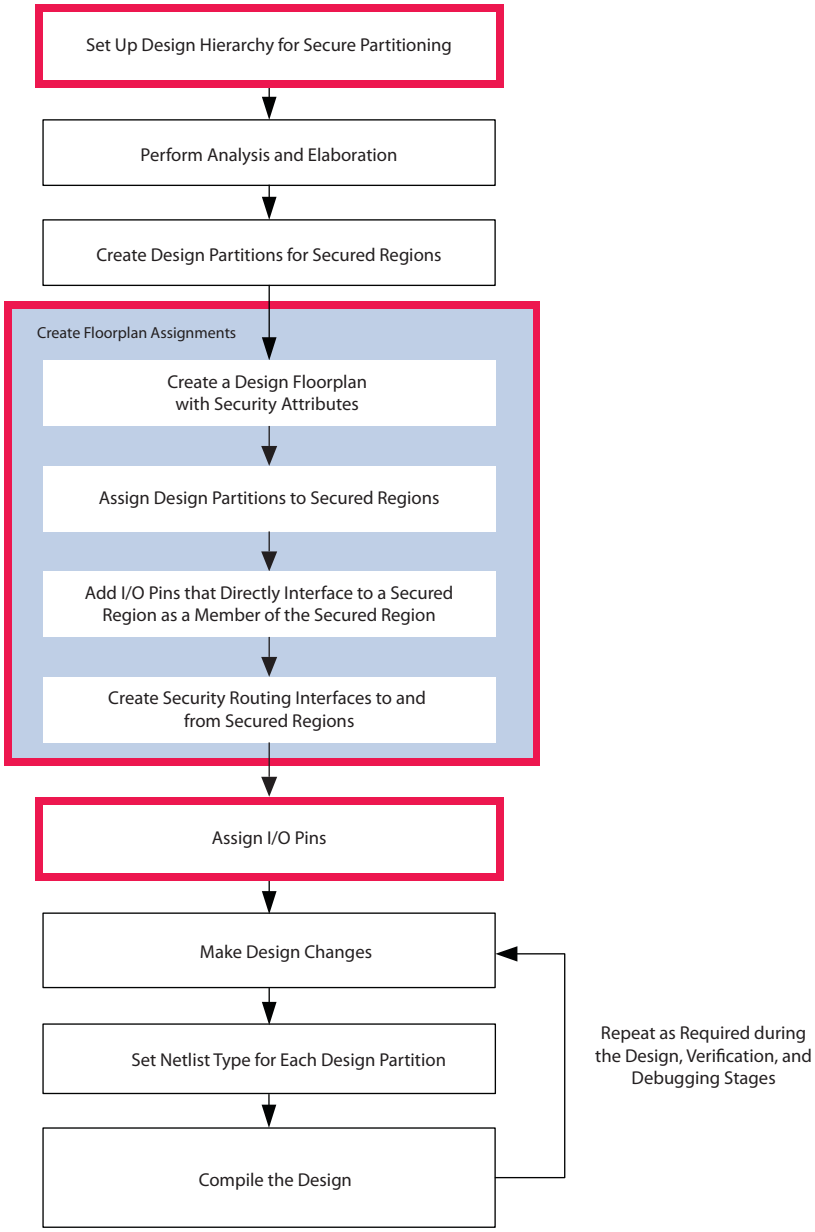
After creating secured regions and security routing interfaces, if the secured regions contain I/O pins as members, you must assign I/O pins so that the entire I/O bank is usable only by the secured region that sinks or sources the I/O pin.

After you make necessary I/O pin assignments, you complete the design separation flow-specific steps, and you can start the iterative process of making design changes, setting the netlist type for each design partition, and then compiling your design until you achieve a floorplan that meets your design requirements.

Rapid recompilation is not supported in the design separation flow. Rapid recompilation is used in place of, or together with, creating design partitions, and allows the Quartus II software to reuse placement and routing resources of compatible nodes.

Figure 4–2 shows a flow chart of the incremental compilation flow with modified floorplan constraints that support the design separation flow. Red boxes in the flow chart highlight steps that are specific to a design separation flow design. A brief description is given for each step in the flow chart below and serves as a quick-start guide for the design separation flow.

Figure 4–2. Design Separation Compilation Flow



1. **Set up design hierarchy for secure partitioning**—Because each secured region must contain only a single design partition, you must set up your design hierarchy to support partitioning along logical hierarchical boundaries. If necessary, create wrapper files to create logical boundaries in the design hierarchy to support the design entities that you must separate from the remainder of the design.
2. **Perform analysis and elaboration**—Run analysis and elaboration to identify the hierarchy in your design.
3. **Create design partitions for secured regions**—Create design partitions so that each secured region will contain only a single partition. Refer to "Creating Design Partitions for the Design Separation Flow" on page 4–5 for more information.

4. **Create a design floorplan with security attributes**—Create LogicLock location assignments, and create a floorplan, at minimum, for all the entities to be secured in your design. Use the security assignments in the LogicLock Regions window to specify the security level of each LogicLock region. Fencing regions are automatically created in your floorplan to isolate the secured LogicLock regions. Refer to "Creating a Design Floorplan with Secured Regions" on page 4–6 for more information.
5. **Assign design partitions to secured regions**—Assign design partitions to secured LogicLock regions to separate them from each other and from all other hierarchy blocks. Refer to "Using Secured Regions" on page 4–9 for more information.
6. **Add I/O pins that directly interface with a secured region as a member of the secured region**—If a secured region interfaces with one or more I/O pins, make the I/O pins members of the secured region. If a secured region has I/O pins as members, that region must overlap the I/O pads. Refer to "Adding I/O Pins as Members of Secured Regions" on page 4–10 for more information.
7. **Create security routing interfaces to and from secured regions**—Create security routing interfaces by applying the security routing interface attribute to LogicLock regions.

   Each security routing interface is a LogicLock region with the security routing interface attribute applied to it. No logic can be placed into a security routing interface; only routing resources can be used within a security routing interface. Each security routing interface must abut one or two secured regions. After you create an interface region for each secured region, assign signals that enter or exit the secured region to a routing interface.

   For signals routing between secured regions with different security attributes or between a secured region and an unsecured region, you must lower the security attribute for the signal exiting the stricter security region. Refer to "Making Signal Security Assignments" on page 4–20 for more information.
8. **Assign I/O pins**—Assign I/O pins to meet design separation flow requirements. For example, I/O banks cannot be shared between secured regions. If a secured region contains I/O pins as members, the entire I/O bank is usable only by the secured region that sinks or sources the I/O pin. Refer to "Assigning I/O Pins" on page 4–26 for more information.
9. **Make design changes, set the netlist type for each design partition, and compile the design**—After you perform a full compilation of your initial project, modify your floorplan and assignments as necessary to achieve a floorplan that meets your design requirements. You can set each partition to a post-fit netlist type, with a preservation level of placement or a netlist type with a lower level of preservation.

The design separation flow-specific steps, step 1 and steps 4 through 8, are described in further detail in subsequent sections in this chapter.

Creating Design Partitions for the Design Separation Flow

After setting up your design to support secure partitioning and running analysis and elaboration, you can create design partitions.

Each secured region floorplan assignment uses a single design partition in the incremental compilation flow to identify the functional elements belonging to a secured region. Design partition assignments are made along entity boundaries in the hierarchy of your RTL design.

Because only a single design partition may be used in a secured region, you must plan your design entities such that logic that requires physical isolation from the rest of the design is packed into a single design entity. Additionally, you should create wrapper files where necessary to reorganize your hierarchy, so that all your secured regions are contained within a single entity or module in your RTL. The incremental compilation feature allows functional independence of each design partition.

You must turn on the **Full incremental compilation** option when you use the design separation flow. Design partitions in the incremental compilation flow are synthesized and placed independently with minimal dependencies on the rest of the design.

Most of the rules, guidelines, and tools for creating design partitions used in the incremental compilation flow are applicable in the design separation flow. You can use the Incremental Compilation Advisor, the Design Partition Planner, and the Chip Planner features in the Quartus II software to help you create design partition assignments.

The design guidelines for creating design partitions when using incremental compilation also apply when creating design partitions in the design separation flow. The following considerations are important:

- Register the inputs and outputs of a design partition to avoid cross-boundary logic optimizations and to maintain timing performance along the signal path.
- Minimize the number of I/O paths that cross between partition boundaries to keep logic paths within a single partition for optimization. Minimizing the number of cross-boundary I/O paths makes partitions more independent for both logic and placement optimization.
- Avoid logic that requires cross-boundary logic optimizations.

For more details about design guidelines for creating design partitions, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.

When creating your design in the design separation flow, you must be aware of some restrictions and special considerations that differ from the incremental compilation flow. These considerations are discussed in the following "Merging PLL Resources" and "Avoiding Child Partitions" sections.

Merging PLL Resources

In the Quartus II incremental compilation flow, the Fitter can use the same PLL resource on the device when multiple design partitions instantiate a PLL with the same parameters. This resource merge occurs even if optimization across design partitions is required. When the design separation flow is enabled, PLL merging across design partitions is disabled, which helps maintain the physical separation between design partitions. PLL merging is disabled regardless of the security levels of the region, even if LogicLock regions in a Cyclone III LS design contain no security attributes. For partitions that require shared PLL resources, the PLL must be instantiated outside of the design partitions.

Avoiding Child Partitions

Because the presence of child partitions creates multiple design partitions in a secured region, they are not allowed in the design separation flow. Each secured region, which you designate after creating design partitions, must contain only a single design partition. Child partitions are design partitions created from a subentity of an existing design partition.

Creating a Design Floorplan with Secured Regions

After you create design partitions, you can create a design floorplan with secured regions using the Chip Planner and security attributes in the LogicLock Regions window.

The Quartus II software uses LogicLock location assignments to map logic in your design hierarchy to physical resources on the device. The Chip Planner provides a visual floorplan of the entire device and allows you to move and resize your LogicLock location constraints on the floorplan of the device. The design separation flow adds an additional security attribute constraint to each LogicLock region to further constrain routing to achieve physical isolation between LogicLock regions. Signals that require connectivity between two secured regions or between a secured region and unsecured logic are assigned to a special LogicLock region known as a security routing interface region. The security routing interface is a controlled region and limits the routing of the contained signals to only the destination this region abuts.

To create fault isolation between secured regions, the design separation flow selectively shuts off routing around the periphery of a secured region. Because signal connectivity at the boundary of the secured region is unused, any faults that occur within the secured region are prevented from adversely affecting neighboring regions. Fault isolation when using the design separation flow is possible because no physical connection exists to propagate the fault outside of the region.

Cyclone III LS devices use a MultiTrack interconnect architecture consisting of row and column interconnects that span fixed distances to achieve signal connectivity between LABs. In the horizontal direction, row interconnects use wire resources that span 1-LAB, 4-LABs, and 24-LABs. These row-routing resources are direct link interconnects, R4 interconnects, and R24 interconnects, respectively. In the vertical direction, routing resources span distances of 1-LAB, 4-LABs, and 16-LABs. These column routing resources are register chain interconnects, C4 interconnects, and C16

interconnects, respectively. In the design separation flow, LogicLock region routing wires (C4, C16, R4, and R16) that span outside the border of a boundary are turned off. Each secured region uses an unused boundary (or a fence) of LABs to guard band against the faults from wire resources spanning a length of one-LAB (direct link and register chain routing resources) from affecting a neighboring region.

The rules and guidelines for floorplanning in the design separation flow are similar to those in a typical compilation flow. However, there are some special considerations for the relative placement of secured regions in your design floorplan. Because each secured region is a keep-out region for routing resources from other LogicLock regions, ensure that a routing path with valid communication interfaces exists between secured regions. Furthermore, the routing path (encapsulated in a security routing interface) should not follow a circuitous path and must be simple enough to meet your timing requirements.

A Fitter-generated floorplan is not possible while a security attribute is applied to a LogicLock region; that is, the size attribute cannot be **Auto**, and the state attribute cannot be **Floating** for a secure LogicLock region.

You can leverage a Fitter-generated floorplan, created without security attributes, as a starting point to create a final floorplan for the design separation flow.

To use a Fitter-generated floorplan as an initial floorplan, apply **Reserved** attributes to LogicLock regions that must be physically isolated from the rest of the design. A Fitter-generated floorplan with **Reserved** attributes generates non-overlapping LogicLock regions. You can modify the initial floorplan by adjusting the relative placement for each secured region, taking into account the connectivity requirements for each region.

Subsequent sections further detail the rules and guidelines for floorplanning that are specific to the design separation flow.

For more information about using the Chip Planner settings and options, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Using Security Attributes

When you use the design separation flow, the **Security Attributes** column in the LogicLock Regions window and the **Security** tab in the **LogicLock Regions Properties** dialog box are present. Setting the security attribute applies a constraint to a LogicLock region, making the region either a secured region or a security routing interface, from where signals enter or exit a secured region. The design separation flow security features are highlighted in the LogicLock Regions window and the **LogicLock Regions Properties** dialog box in Figure 4–3 and Figure 4–4, respectively.

Figure 4–3. Security Attribute Column Available in the Design Separation Flow

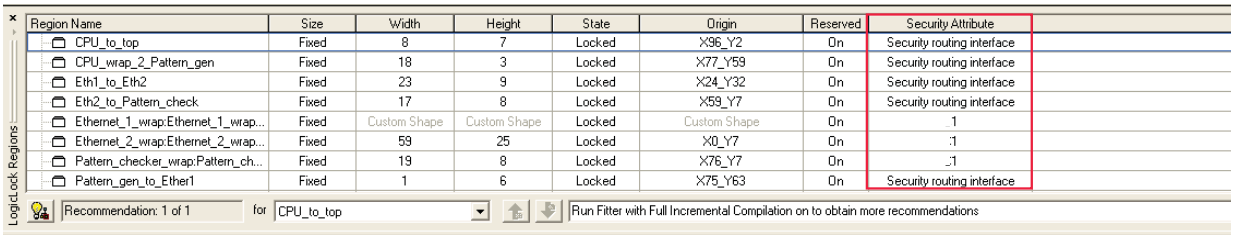
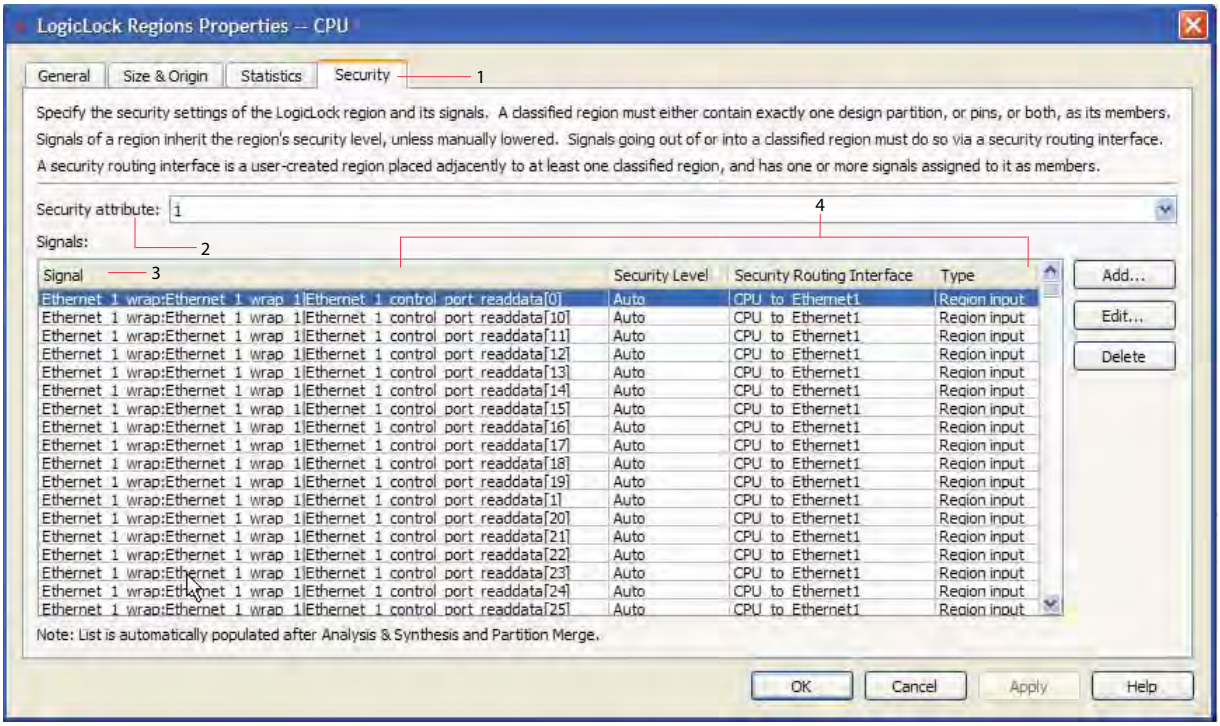


Figure 4–4. Security Tab Available in the Design Separation Flow



Notes to Figure 4–4:

(1) **Security** tab

(2) **Security attribute** applies routing constraints to LogicLock regions

(3) **Signals** list is prepopulated after analysis and synthesis with inputs and outputs of secured region

(4) Columns in the **Signals** list describe the **Security Level**, the security routing interface the signal is assigned to, and whether the signal is an output or input to the region

Table 4–1 lists a summary of the **Security attributes** available for the design separation flow.

Table 4–1. Security Attributes for LogicLock Regions

Security Attribute	Description
Unsecured	Disables the constraint for physical isolation.
1	Creates a secured region. Physically isolates the LogicLock region by restricting routing resources from leaving the region. Creates a one-lab width border of unused LABs around the LogicLock region. Applying this attribute to a LogicLock region sets the global assignment LL_REGION_SECURITY_LEVEL 1 to the LogicLock region.
2	Creates a secured region. Security attribute 2 represents a stricter level of fault isolation than security attribute 1. For Cyclone III LS devices, implementation of security attribute 2 is the same as security attribute 1; however, this may not be true in future architectures supporting the design separation flow. When selected, for the Cyclone III LS family, the Quartus II software creates a one-lab width border of unused around the LogicLock region. Applying this attribute to a LogicLock region sets the global assignment LL_REGION_SECURITY_LEVEL 2 to the LogicLock region.
Security Routing Interface	Creates a routing interface for signals entering or exiting a secured region. Only routing resources (no logic) may be used within a security routing interface. Applying this attribute to a LogicLock region sets the global assignment LL_SECURITY_ROUTING_INTERFACE ON to this LogicLock region.

Using Secured Regions

When you apply a secured region attribute (1 or 2) to an existing LogicLock region, the LogicLock region must have a fixed size with a locked origin. Each secured region must have a minimum size of eight-LABs in both the horizontal and vertical dimensions. A region smaller than 8 × 8 LABs may be non-routable when using the design separation flow.

Child regions are not allowed when creating a secured region. In the non-secured compilation flow, child regions are useful primarily for the following two reasons:

- To ensure that logic in a child partition is physically contained inside the LogicLock region of the parent partition
- To make non-rectangular LogicLock regions by using the child region as a keep-out region for logic contained within the parent region (using the limited attribute)

The first case is unnecessary because a secured region contains only a single partition. The second cause is unnecessary because a simpler flow to create non-rectangular LogicLock regions is available for the design separation flow.

For more information about creating non-rectangular LogicLock regions with the design separation flow, refer to "Creating Non-Rectangular Regions" on page 4–14.

Adding I/O Pins as Members of Secured Regions

A secured region must contain all physical device resources required to complete compilation. I/O pads that are members of a secured region must be contained within the boundaries of the secured region that sources or sinks it. That is, a secured region must overlap the I/O pads that are members of the region. If the logic in the secured region instantiates a PLL or a clock block, those physical device resources must also be overlapped by the region.

You can add I/O pins as members of a secured region using the **LogicLock Region Properties** dialog box.

Using Security Routing Interfaces

A LogicLock region with the security routing interface security attribute creates a routing channel for signals to and from a secured region. No logic may be placed in a security routing interface. Each security routing interface can connect two secured regions, or a secured region with one or more unsecured regions. If you are connecting two secured regions, a fence is automatically placed around the interface region. You can assign each signal entering or leaving a secured region to a security routing interface on the **Security** tab in the **LogicLock Regions Properties** dialog box.

For information about assigning signals to a security routing interface, refer to "Making Signal Security Assignments" on page 4–20.

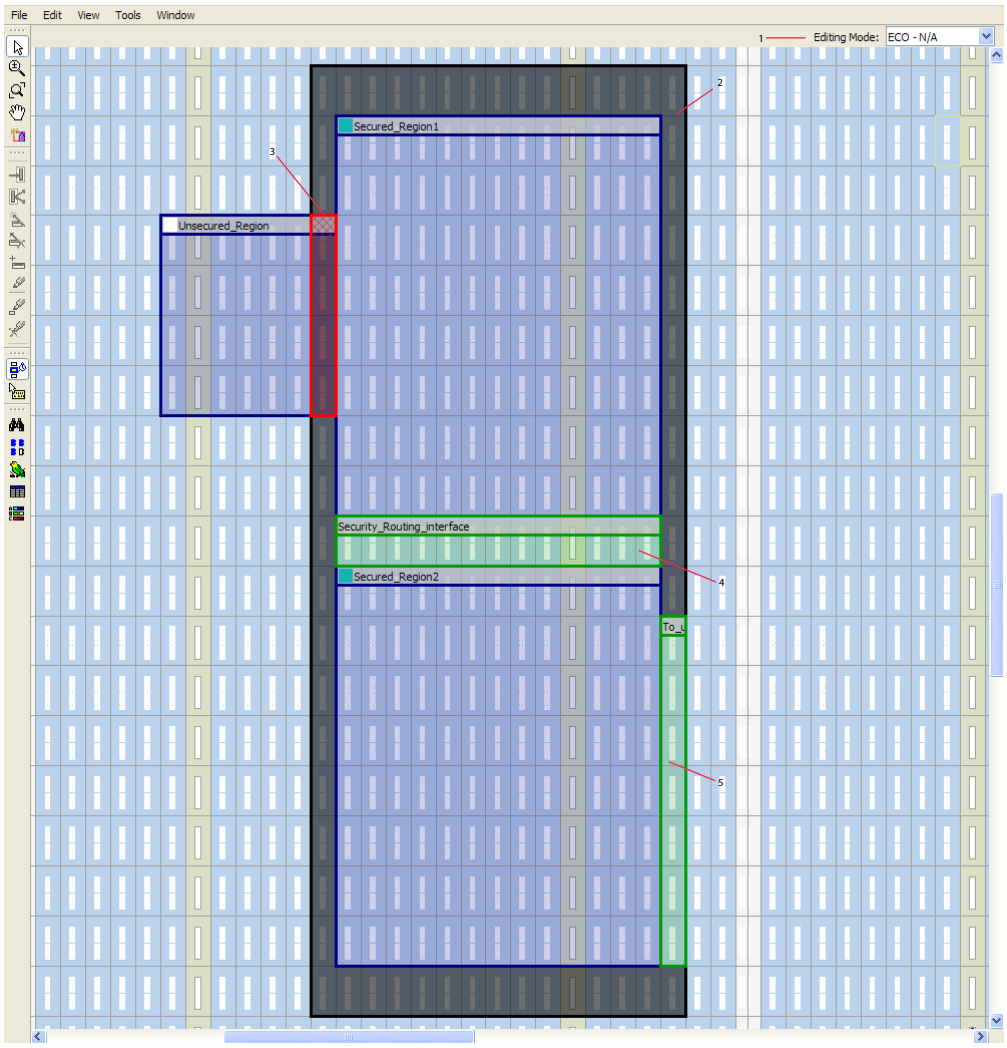
For information about the number of signals that can be contained in a security routing interface, refer to "Routing Restrictions" on page 4–27.

Making Design Separation Flow Location Assignments in the Chip Planner

The Chip Planner allows you to visually modify the size and location of LogicLock regions. This section describes the attributes of LogicLock regions within the context of the design separation flow.

When the design separation flow is enabled, the fencing region around each secured region in the Chip Planner is shaded brown. Security routing interfaces are shaded green. Illegal placements that violate secured region boundaries are highlighted in red at the location where the violation occurs. Figure 4–5 shows the LogicLock regions with security attributes in the Chip Planner.

Figure 4–5. LogicLock Regions Using Security Attributes



Notes to Figure 4–5:

(1) Floorplan Editing Task
(2) Unused fence around a secured region
(3) Security violation, created from a LogicLock region being placed within a fence boundary of a secured region
(4) Security routing interface region connecting two secured regions
(5) Security routing interface region connecting secured region and unsecure logic

Understanding Fencing Regions

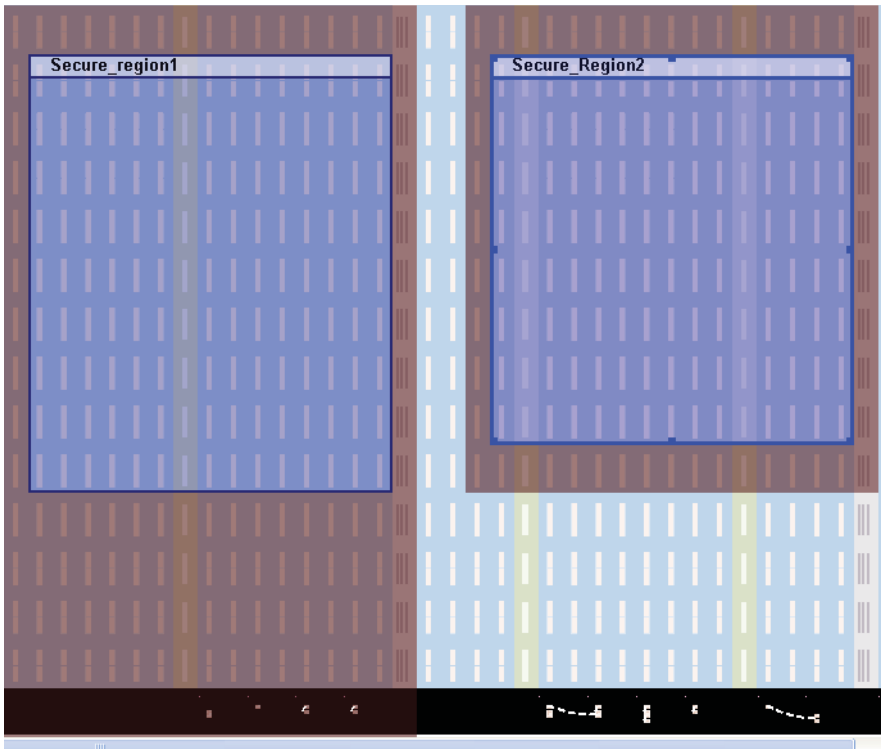
The Quartus II software automatically adds a fencing region, which is a border of unused LABs, when you apply a security attribute 1 or security attribute 2 to a LogicLock region. No logic may be placed into a fencing region. The Fitter does not use any routing wires that go outside of the fence boundary that are within a secured region. Because direct drive and carry chains can be used at the edge of a secured region, the fencing region prevents signals being driven on length one wires (in the horizontal and vertical directions) from leaving the secured region.

The fence size around a secured region is generally one-LAB in the horizontal direction and one-LAB in the vertical direction. There are two regions that have a special consideration for the fencing region:

- Vertical I/O regions
- Areas around the configuration engine

I/O banks along the top and bottom of the chip use only vertical routing wires to and from the I/O Elements (IOEs). The heavy use of C4 wires from IOEs creates a four-LAB fence between the vertical I/O banks and a secured region. Secured regions requiring a connection to I/O in the top or bottom banks of the device optimally use resources if you add the I/O signals as members of the secured region and overlap the corresponding I/O pads in the floorplan. Figure 4–6 compares the fence sizes of two secured regions: one is five LABs away from the bottom of the device and the other is four LABs away from the bottom of the device.

Figure 4–6. Vertical Fence Size Near I/O Banks



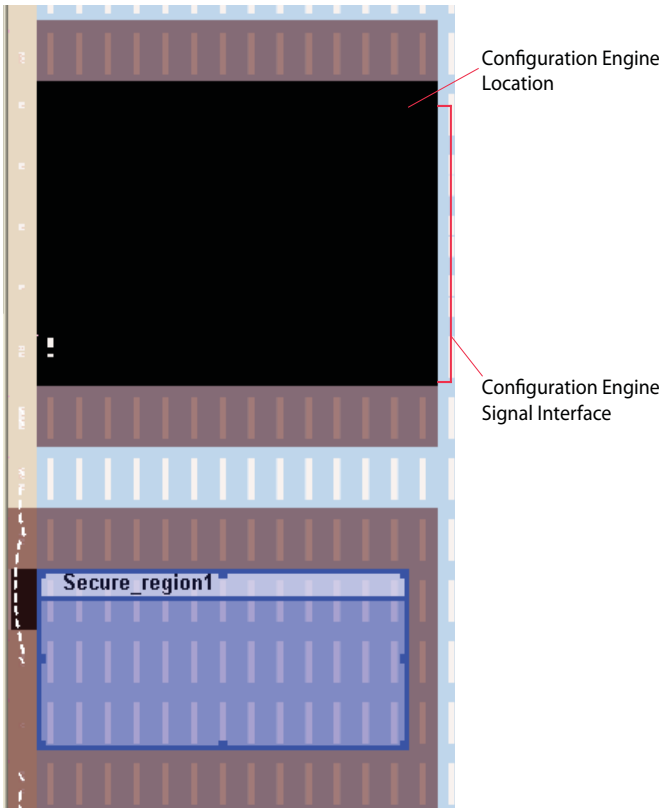
The configuration engine is a hard IP block that manages the configuration of the device. In addition to configuration services, the configuration engine routes the control signals for the CRC detection circuit and the internal oscillator into the core logic on the device. In the design separation flow, a one-LAB fence is automatically added around the configuration engine whenever a secured region occupies the same LAB column as the configuration engine. The configuration engine is a region notched out of the left side and halfway down the device.

All control signals to and from the configuration engine route from its right edge. If you use an instantiated WYSIWYG that uses any control signals to and from the configuration engine, the signals must either interface to unsecured logic or they must interface with a secured region through a security routing interface.

If your design requires signals to and from the configuration engine, a secured region that directly abuts the configuration engine signal interface (along the right side of the configuration engine) causes a Fitter error.

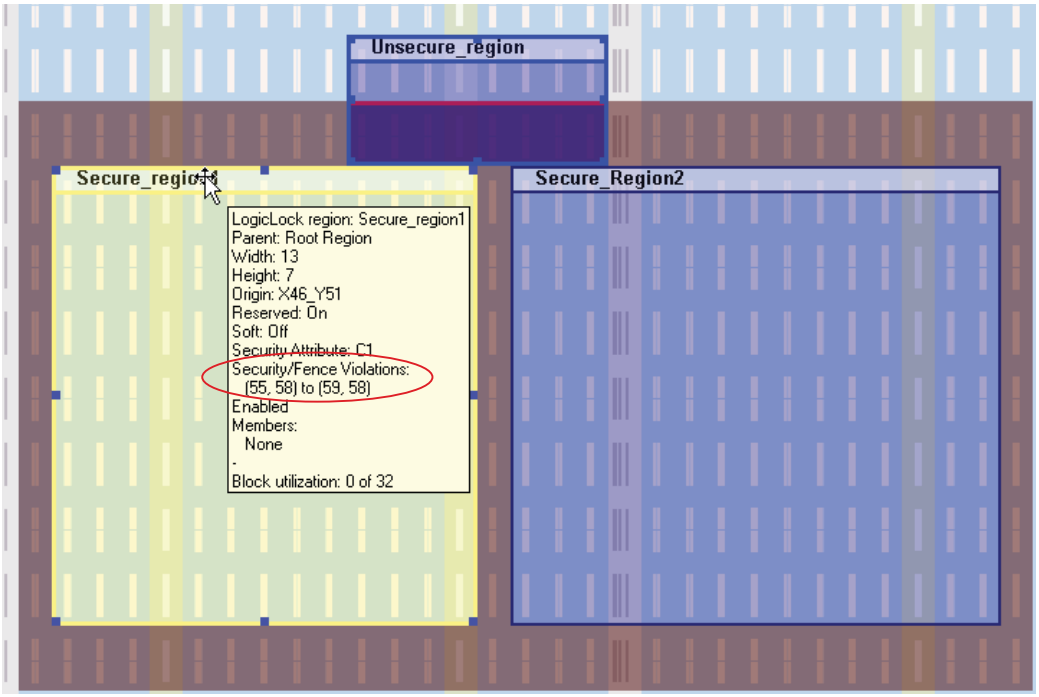
Figure 4–7 shows a configuration engine with a fencing region in the floorplan for a Cyclone III LS device.

Figure 4–7. Configuration Engine in the Chip Planner



Fencing regions between two secured regions are allowed to overlap. That is, two adjacent secured regions can be separated by a one-LAB fence. The Chip Planner issues a security violation warning if a LogicLock region is placed within the boundary of a secured region. Security violations are highlighted in red and the tooltip of a secured region indicates the locations of all security violations. You may receive an error if you try to compile a design with a security violation. Figure 4–8 shows two regions with overlapping fences and a security violation from an unsecured region.

Figure 4–8. Overlapping Regions *(Note 1)*, *(2)*



Notes to Figure 4–8:

(1) One-LAB fence can be shared between regions

(2) Tooltips list all locations of a security violation

Creating Non-Rectangular Regions

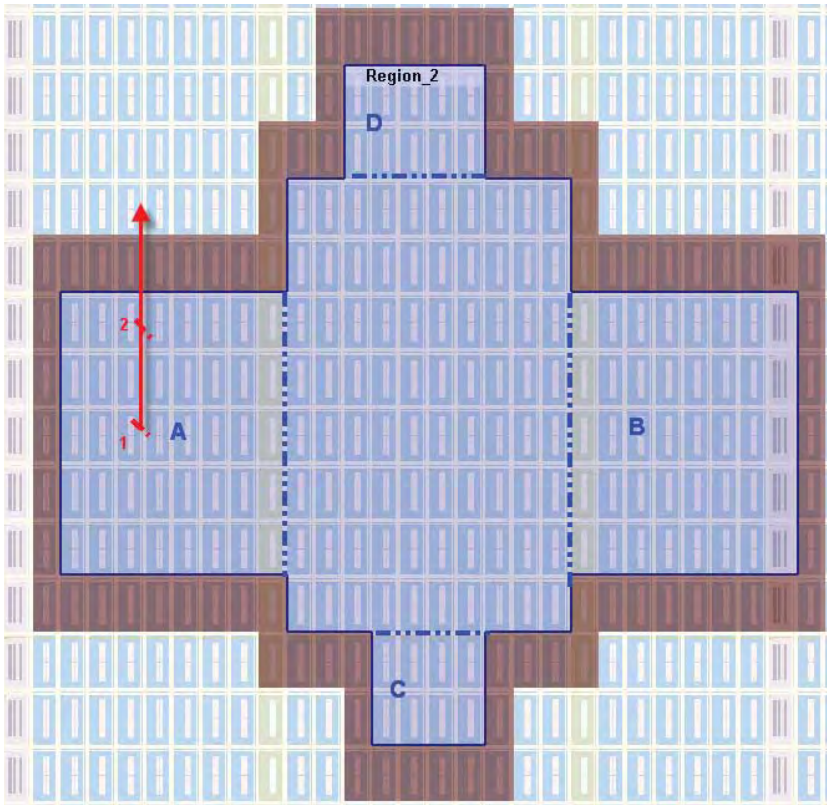
You can create non-rectangular regions by creating multiple rectangular regions and then merging them.

For more information about creating non-rectangular regions in the Chip Planner in the Quartus II software, refer to *Creating and Manipulating LogicLock Regions* in Quartus II Help.

Non-rectangular LogicLock regions in the design separation flow make circuitous routes more likely. As such, non-rectangular regions can have an adverse affect on performance when used with the design separation flow.

If a secured non-rectangular region contains a subregion that is less than 8 × 8 LABs, the chances of a non-routable situation occurring increases. Subregions that deterministically require the use of certain routing resources may not fit successfully if violation of the secured region is required. As a general guideline, each subregion should be 8 × 8 LABs or larger, to ensure that routing resources of length four are readily available. In Figure 4–9, each subregion of region 2 (labeled A, B, C, and D) are less than 8 × 8 LABS in dimension. These subregions can potentially cause a no-fit situation. Depending on the placement and connectivity of LABs, certain routes may be difficult to achieve. For example, the Fitter would not be able to route a connection from LAB 1 to LAB 2 in region A directly. While another path may be possible, a series of hops that do not leave the LogicLock region may not be available and may not satisfy the timing requirements of the route.

Figure 4–9. Non-Rectangular LogicLock Regions



Guidelines for the Relative Placement of Secure LogicLock Regions

Because each secured region is a keep-out region for placement and routing of any logic that is not a member of the secured region, there are a few guidelines to consider as you lay out your floorplan. Placement that does not account for the connectivity requirements between LogicLock regions may cause poor performance or a non-routable design. The guidelines for floorplanning when using the design separation flow include:

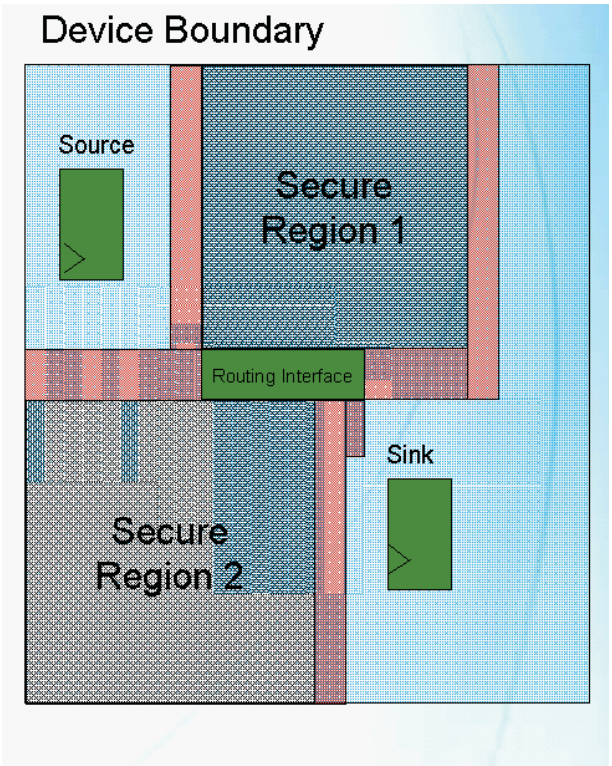
- Create a complete floorplan, including location assignments for unsecure logic.
- Create a non-circuitous route between secured regions requiring a routing region. Generally, routing regions between secured regions should be rectangular.
- Create security routing interfaces between secured regions that do not intersect with other routing regions; secured regions and their routing edges must fit on a single plane. A secured region must overlap any physical resources (such as I/Os, PLLs, and CLKCTRL) that are instantiated by the design partition contained in the secured region.
- Abut the secured region to the edge of the device whenever possible.

Creating a Complete Floorplan

You must allocate a region for each partition in your design. If you have a large secured region that divides the device into multiple disjointed regions, and you have unsecured logic that is not floorplanned, the design may be non-routable.

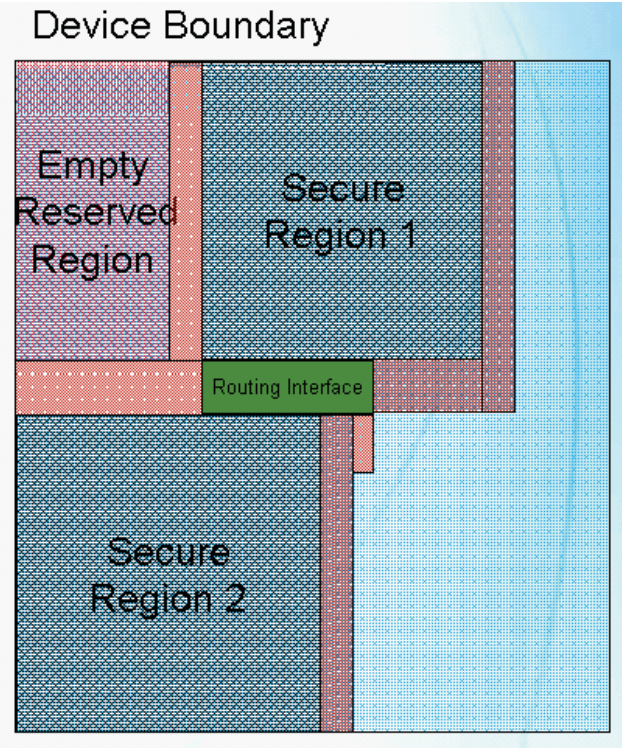
If there is an unsecure partition and it does not contain any location assignments, the placement algorithms may use any unallocated space on the device to make logic assignments. In the floorplan shown in Figure 4–10, the source and sink registers do not have a valid path through the device, because all routing channels are occupied by Secure Region 1 and Secure Region 2.

Figure 4–10. Non-Routable Placement Example



If a complete floorplan is not possible for all partitions in your design, you can use empty LogicLock regions with the **Reserved** attribute to prevent the Fitter from placing any logic in a region that can potentially cause a no-fit. For the example provided in Figure 4–10, an empty region can be placed in the upper-left corner of the device to prevent any logic that has not been floorplanned from being placed there, which is then shown in Figure 4–11.

Figure 4–11. Empty Reserved Region Preventing Fitter From Placing Logic

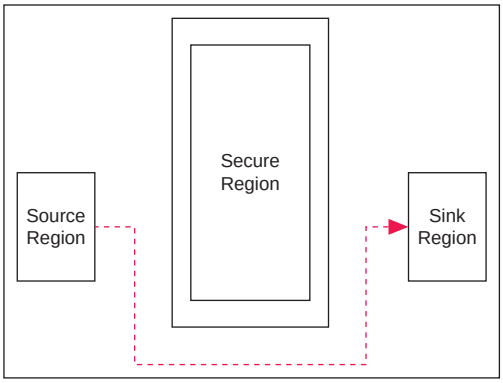


Ensuring Routability Between Regions

The Quartus II software cannot create auto-generated location constraints for any region with a security attribute. If you use a Fitter-generated placement as a starting point for a floorplan for the design separation flow, an optimal floorplan in a design without separation may not work in the same design with security attributes applied to a region. In a floorplan without secured regions, only the placement of logic is restricted. All routing resources on the device are available for the Fitter to use, and a route may go through a region. Secured regions reserve all routing resources within the LogicLock boundary to the design partition contained in the region.

Having a circuitous route between two regions degrades performance and may cause a non-routable design. Modify any regions that have signal connectivity and must route around a secured region to achieve a connection. Figure 4–12 shows a floorplan that does not contain disjointed parts. However, the source region must route around a secured region to connect to the sink region.

Figure 4–12. Relative Placement of Regions Containing a Circuitous Path *(Note 1)*



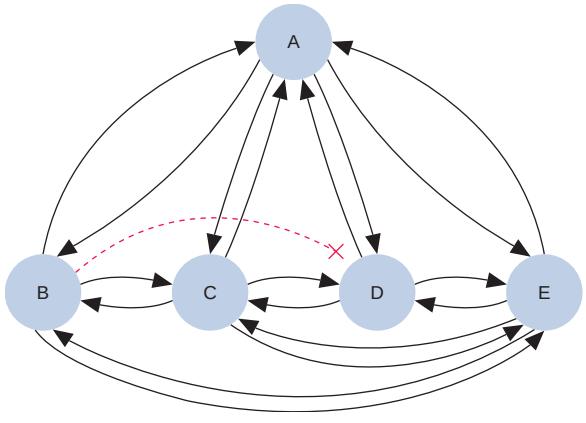
Note to Figure 4–12:

(1) Initial placement from the Fitter may cause routability issues in the same design with secured regions.

Ensuring Planarity

A fence is automatically created around a security routing interface connecting two secured regions. Because no other routing resources may pass through a security routing interface connecting two secured regions, you must model all secured regions as nodes in a routing graph and all security routing interfaces as the edges, and all nodes and their edges must fit on a planar graph (that is, none of the edges can intersect). If you have five or more secured regions on the device, and each secured region contains signals that fan-out to multiple secured regions, a planar floorplan may not be possible. Figure 4–13 shows a routing graph with five nodes. A complete graph having each pair of distinct vertices connected by an edge is not possible without having any of the edges cross. If the topology of your floorplan contains such a non-routable arrangement, your design hierarchy must be rearranged to collapse related design partitions into a single design partition.

Figure 4–13. Non-Planar Routing Graph: Connection BD Not Possible



If your secured regions and security routing interfaces can be modeled as a planar graph, but have a high degree of connectivity between the components, you may have to rearrange the shape, size, or location of the secured regions to generate a routable floorplan. For instance, the hypothetical floorplan shown in Figure 4–14 does not have a valid routing path BD (between region B and region D). The modified floorplan in Figure 4–15 shows how all of the required connections can be achieved on a planar surface.

Figure 4–14. Non-Routable Connection BD

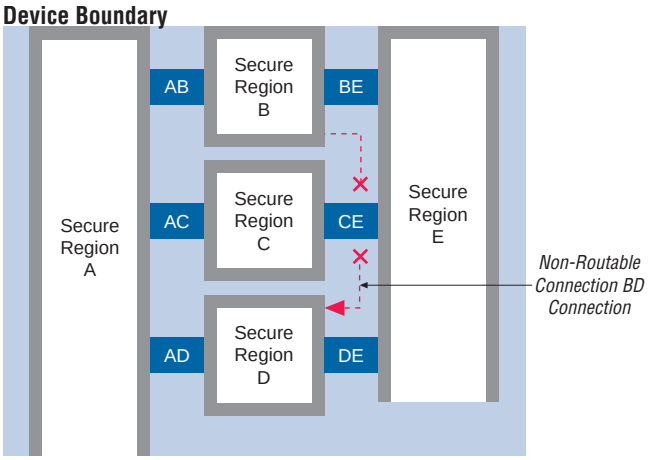
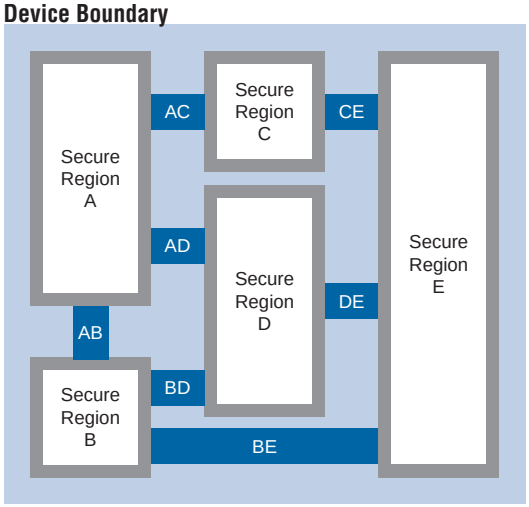
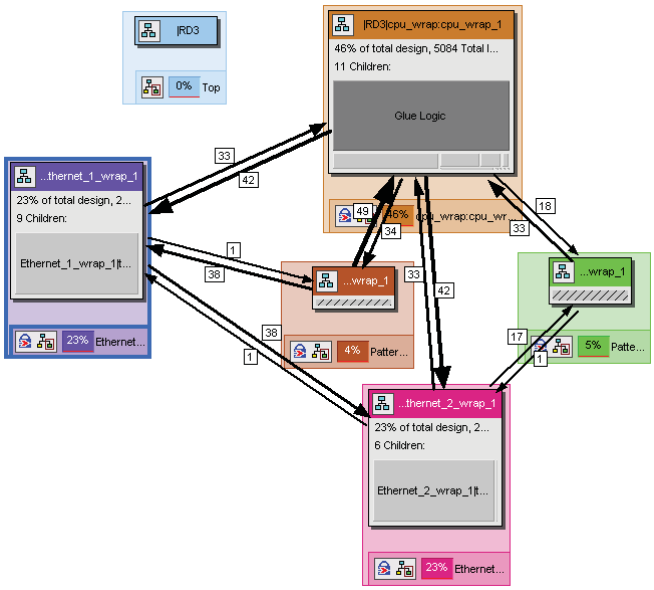


Figure 4–15. Floorplan Arranged to Accommodate BD



You can use the Design Partition Planner for a visual representation of the connectivity between design partitions. This tool helps you determine if the secured regions in the design can be arranged on a planar floorplan. Figure 4–16 shows the Design Partition Planner.

Figure 4–16. Design Partition Planner



Placing Physical Resources

All physical resources that are required by the secured region must be contained inside the boundary of the secured region, including the following:

- I/O pins connected to the secured region
- Primitives that have been instantiated within the secured region, such as PLLs and clock control blocks

Making Signal Security Assignments

Each signal that enters or exits a secured region must contain a security level attribute and be explicitly assigned to a security routing interface. The security level for each signal is automatically assigned a default value and matches the secured region that sources the signal. The possible values of the security level of a signal are **Auto**, **Unsecured**, **1**, and **2**. An assignment of **Auto** sets the default security level for the signal.

A signal with a security attribute may connect to a region with an equivalent or higher security level. For example, a signal with a security level of **Unsecured** can drive logic in a region set to **Unsecured**, **1**, or **2** and a signal with a security level of **1** can drive logic in a region set to **1** or **2**. A signal originating from a secured region may not drive logic in a region with a lower security level. If you have a signal from a higher security level that must drive logic in a lower security level, you can direct the Fitter to honor the connection by explicitly lowering the security level of the signal.

At most, each security routing interface connects two regions. If a signal fans out to multiple regions, assign the signal to multiple security routing interface regions; one interface region per destination.

Assignment of signals to security routing interfaces and the security level of signals can be modified using the **Security** tab in the **LogicLock Region Properties** dialog box, shown in Figure 4–4.

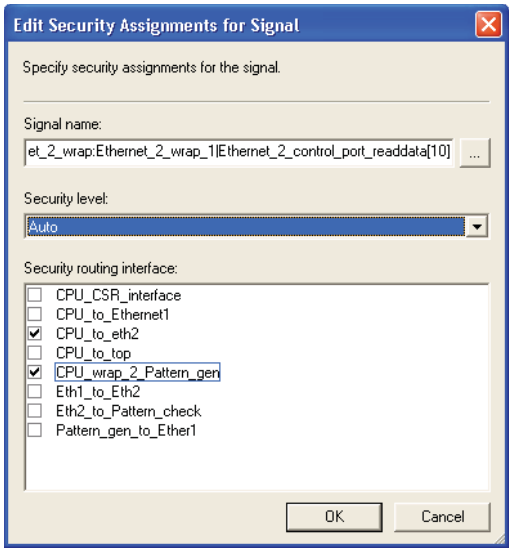
To assign a signal to a security routing interface, follow these steps:

1. On the **Security** tab of the **LogicLock Regions Properties** dialog box, select a signal name in the **Signals** list, and then click **Edit**. The **Edit Security Assignments for Signal** dialog box appears, as shown in Figure 4–17.

   The **Signals** list is prepopulated with the names of signals entering and exiting the secured region after analysis and synthesis and a partition merge have been run successfully.

Alternatively, you can select multiple names in the **Signal** list by holding down the **Ctrl** key, clicking multiple names, and then clicking **Edit**.

Figure 4–17. Edit Security Assignments for Signal Dialog Box



2. If necessary, lower the security level of the signal by specifying the **Security level**.
3. Select the security routing interface for signal(s) assignment. Signals that fan-out or fan-in to multiple regions can be assigned to multiple security routing interfaces.

Understanding Signal Names

The list of signals entering and leaving a secured region are signal names from the post-map netlist. Signal outputs from a secured region are derived from the output port name, as specified in the top-level RTL entity contained in the secured region. Signal inputs to a secured region are derived from the name of the register or LUT that drives the port. In the design separation flow, all output port names are preserved through the compilation process.

The post-map region output signals listed in the signal list coincide with the signal name in the post-fit netlist. However, combinational signal names from unsecure or unpartitioned logic that feed a secured region may change through the compilation process. Many of the RTL signals are optimized during the process of synthesis and place-and-route. Frequently, RTL signal names may not appear in the post-fit netlist after optimization. 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. Registered signals tend to maintain the same signal names in both the post-map and post-fit netlists. When possible, use registered signals as inputs into a secured region, and register the output signals from a secure design partition.

Working with Global Signals

Global signals are low-skew routing lines that drive throughout the device. Global signals do not require an interface region to drive into a secured region. In Cyclone III LS devices, there are 20 global routing resources for use with high fan-out signals, such as clocks or control signals. Each global signal is accessed by a clock control block, located on the periphery boundary of the device. Each clock control block can be driven directly by external clock pins, PLL outputs, or a signal generated from internal logic.

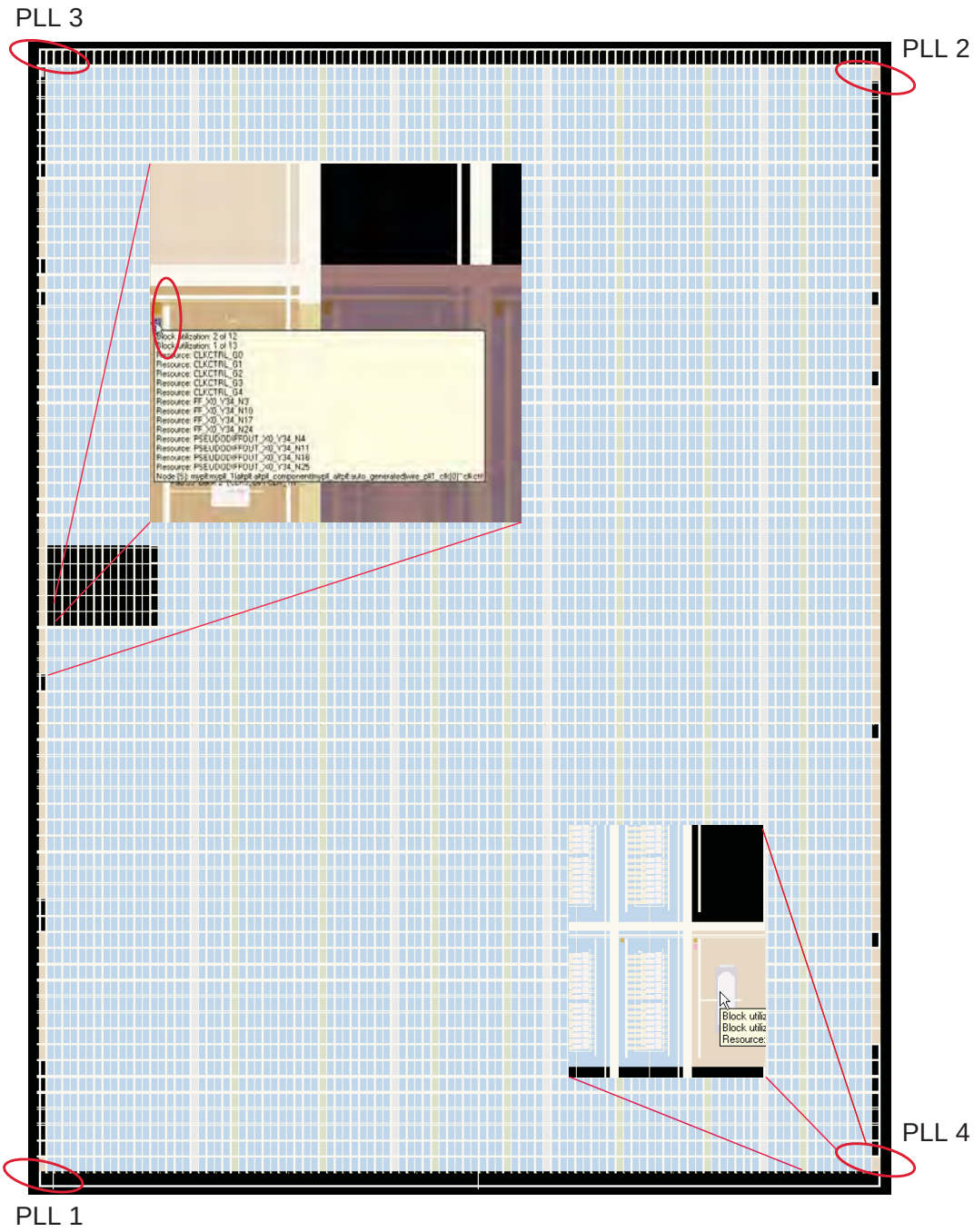
For more information about the clock networks in Cyclone III LS devices, refer to the *Clock Networks and PLLs in Cyclone III LS Devices* chapter in volume 1 of the *Cyclone III Device Handbook*.

In a typical compilation flow, signals with a high fan-out (such as clock pins and control signals) are automatically promoted to use global clock resources. In the design separation flow, automatic global promotion is not turned on. Signals with high fan-out requiring global routing resources must be automatically promoted to drive a clock control block.

Signals cannot be promoted onto a global routing resource through a global signal assignment from within a secured region. The Fitter only allows a clock promotion assignment to a signal if the signal is in an unsecured region. If you have a signal inside of a secured region that must use a global routing resource, you must first route the signal outside of the secured region before applying a global promotion assignment. The signal must be assigned to a security routing interface and the security level of the signal must be lowered.

For a global promotion assignment to be honored, there must be an available clock control block that is not overlapped by a secured region, and an available routing path to the clock control block. The clock control blocks are located on each side of the device, along the horizontal and vertical axes that run through the center of the device. There are five clock control blocks on each side of the device. Figure 4–18 shows the location of the clock control blocks and the PLLs for a 3CLS70 device in the Chip Planner floorplan.

Figure 4–18. PLL and Clock Control Block Location on a EPC3SL70 Device



PLLs and clock control blocks can be manually instantiated in the design partition of a secured region using the ALTPLL and ALTCLKCTRL megafunctions, respectively. Instantiation of the ALTCLKCTRL megafunction within a secure partition forces the global promotion of the signal driving the clock control block. To generate a valid placement when you instantiate PLLs or a clock control block, the secured region containing the physical resource must overlap a free PLL, a free clock control block, or both.

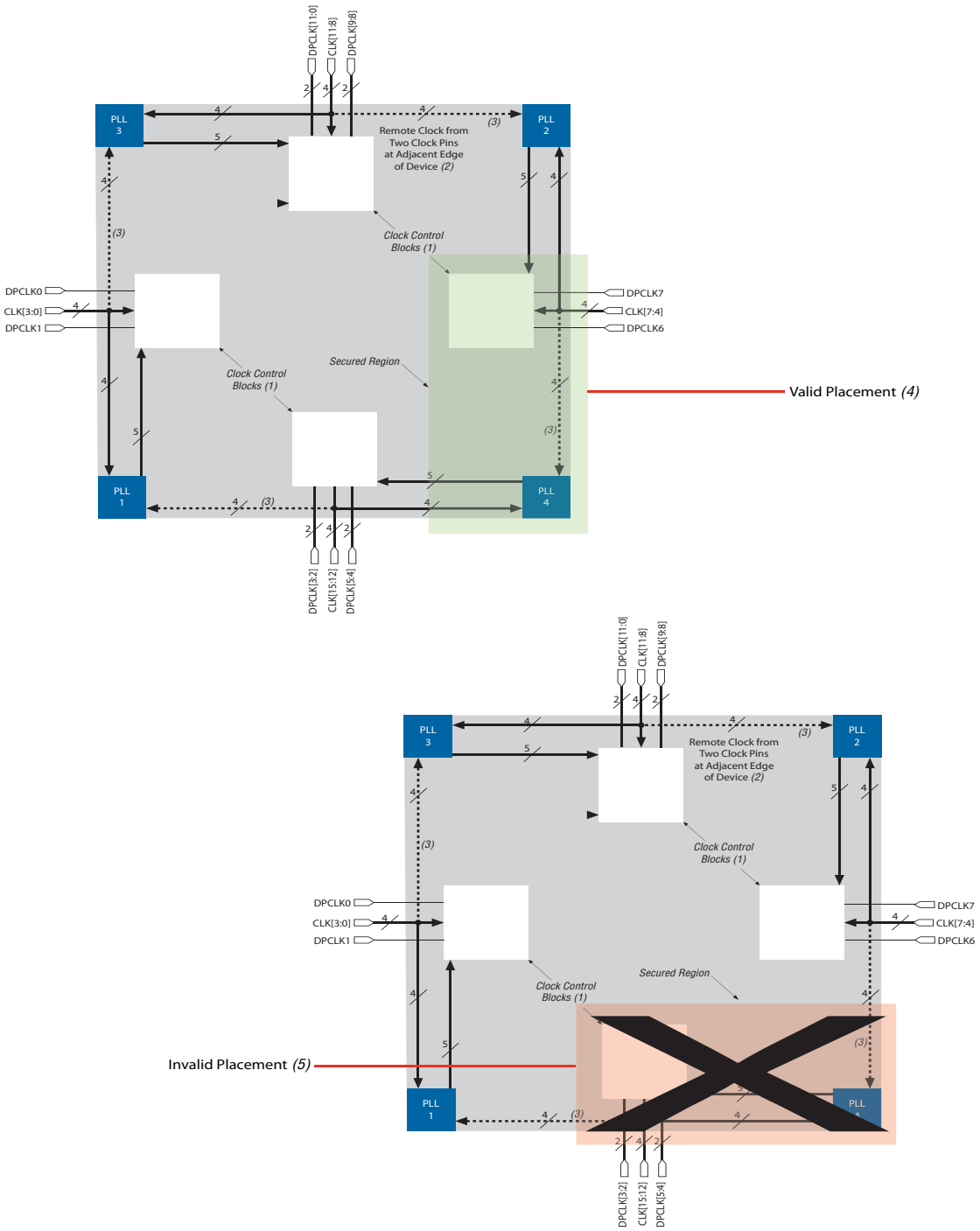
There are certain restrictions you should be aware of when you instantiate a PLL within a secured region. Secured regions with a PLL that are fed by an external clock pin must contain the PLL and a valid clock pin that can drive the PLL. Each PLL has a set of dedicated clock control blocks that it can access, located to the right (clockwise) of the PLL in the device floorplan.

Because automatic promotion of signals onto a global resource is not allowed, a PLL and the clock control block it drives must not be located in the same secure region. If you have a PLL inside of a secure region, you must assign the PLL output to a security routing interface and then lower the security level of the PLL output.

The clock control block associated with the PLL must not be covered by any secure LogicLock region. There are two sets of dedicated clock pins that can drive a PLL input. The pads for the clock input pins are co-located with the clock control blocks. If you use the clock input pin that is co-located with the clock control block associated with the PLL, the clock pin cannot be added as a member of the secured region. Instead, you must assign the clock pin to a security routing interface that is connected with the secured region.

Figure 4–19 shows examples of valid placement and invalid placement of secured regions that instantiate PLLs.

Figure 4–19. Location of Valid and Invalid PLL, Clock Pin, and Clock Control Block Placement in a Cyclone III LS Device



Notes to Figure 4–19:

(1) There are five clock control blocks on each side.
(2) Remote clocks cannot be used to feed the PLLs.
(3) Dedicated clock paths can feed into this PLL. However, these are not fully-compensated paths.
(4) This secured region contains a PLL that is fed by an external clock pin, whose outputs drive the clock control block through an unsecured region.
(5) This secured region contains a PLL whose output drives an clock control block within the same secured region. This placement is invalid.

Assigning I/O Pins

After ensuring that signals that enter or exit a secured region contain a security level attribute and are explicitly assigned to a security routing interface, you must also ensure that I/O pin assignments adhere to design separation flow guidelines.

Consider the following three rules, in addition to the typical pin assignment rules, when assigning I/O pins with the design separation flow enabled:

- I/O pins that are connected to a secured region must be assigned as a member of that secured region or assigned to a security routing interface region that abuts the secured region.
- Secured regions with I/O pins as members cannot share the I/O banks with any other region.
- I/O pins associated with different secured regions or with different security levels may not use adjacent pins.

I/O pins may be added as members of a secured region, typically when directly connected to the secured region. To add I/O pins as members of a secured region, in the **LogicLock Regions Properties** dialog box, under the **General** tab, click **Add node**. If an I/O pin is a member of a secured region, the I/O pad must be physically contained within the region, and the secured region must overlap the I/O resource.

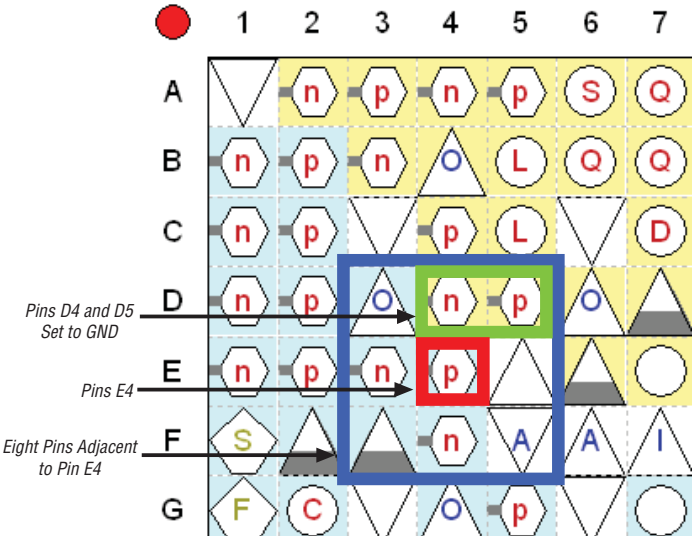
If you do not add the I/O pin as a member of the secured region, you must assign the I/O signal to a security routing interface that abuts the secured region. This security routing interface must connect the secured region to the root region or another unsecured region. Explicitly lower the security level of any output signals from the secured region that are connected to I/O pins.

CAUTION

I/O signals that are routed out to an unsecured logic are no longer guaranteed to be physically isolated from other signals in the design.

Each I/O pin is adjacent to eight other pins: four along the horizontal and vertical axes, and four in the two diagonal axes, as shown in Figure 4–20.

Figure 4–20. Pin Adjacency



Pins from different I/O banks may not share an adjacent I/O pin if one of the I/O banks contains pins that are members of a secured region. User I/O pins that are adjacent to a signal in a secured region, which belong to a different I/O bank than the secured signal, should be assigned to GND in the Quartus II software. For example, in Figure 4–20, pin E4 is assigned a signal from a secured region, and I/O banks 1 and 7 belong to different LogicLock regions. Pins D4 and D5 are assigned to GND to ensure that no signal adjacencies exist between the I/O banks.

As a general rule, all unused I/O pins should be assigned to GND in the Quartus II software and assigned to a ground plane on the PCB. By default, the Quartus II software assigns unused pins to GND. You can configure this option in the **Unused Pins** page of the **Device and Pin Options** dialog box.

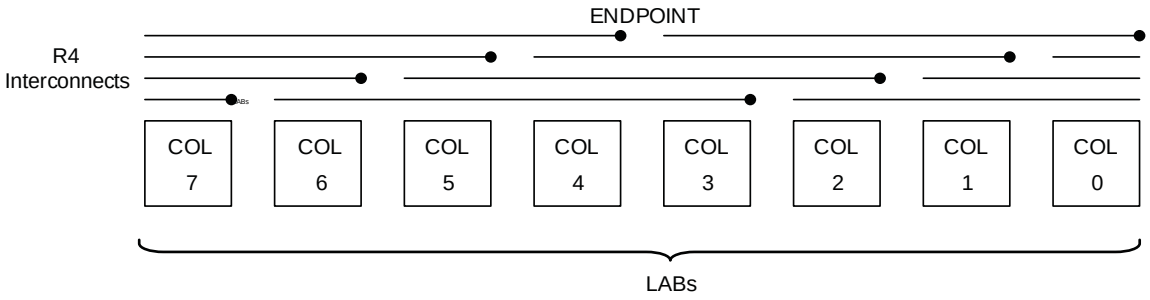
For information about the pinouts and pin adjacencies for Cyclone III LS devices, refer to the *Cyclone III Device Pin-Out* tables. For additional information and guidance about I/O assignments, refer to the *Cyclone III Device Family Pin Connection Guidelines* for Cyclone III LS devices and the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Routing Restrictions

During the overall planning of your design, you should be aware of specific design separation flow routing restrictions, especially during the floorplanning stages. These routing restrictions are discussed in this section.

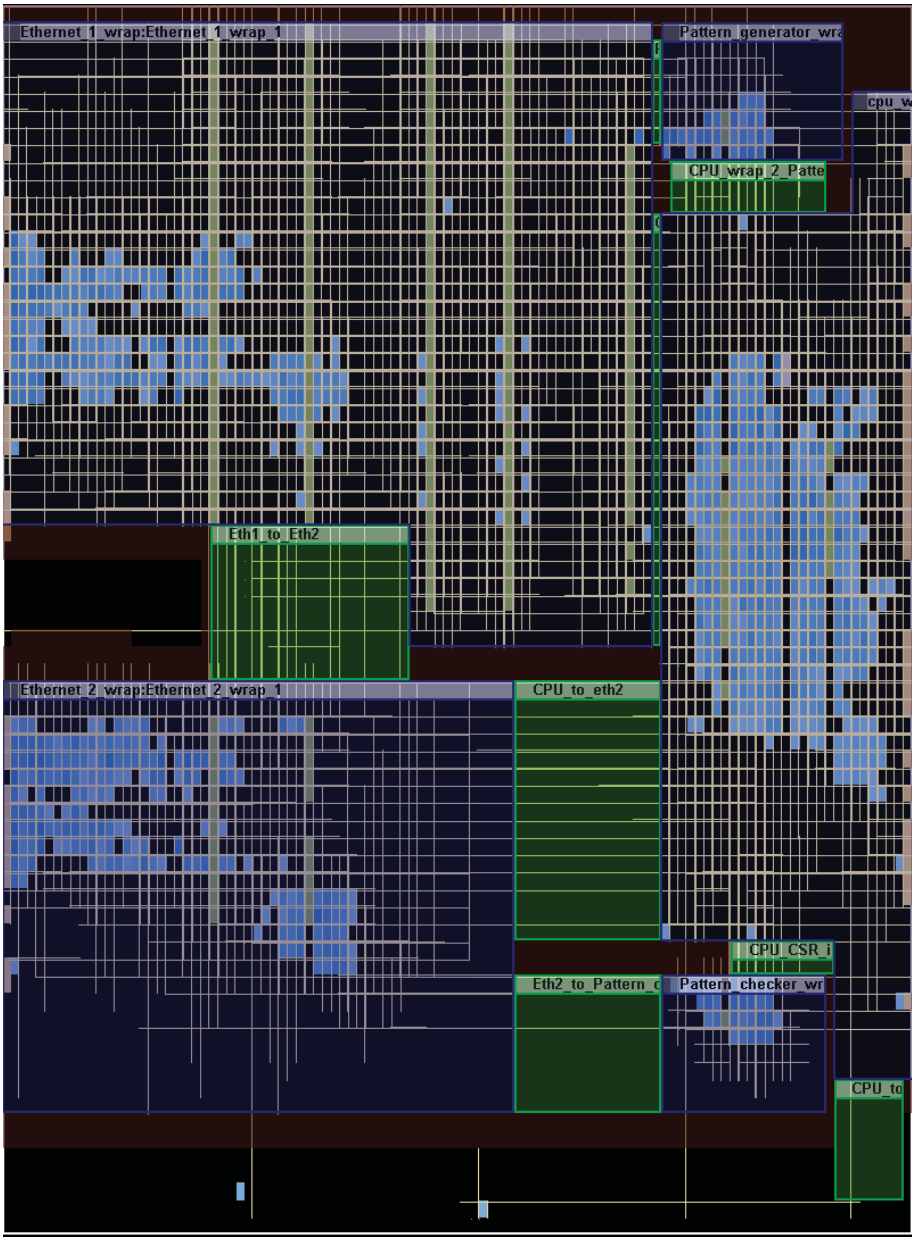
Column and row interconnect routing resources on Cyclone III LS devices are staggered, with a group of routing elements starting at each LAB location. Each routing element is driven by the LAB location where the wire starts and can reach any LAB destination along the length of the routing element. Figure 4–21 shows a set of staggered R4 interconnects.

Figure 4–21. Staggered R4 Interconnects



The Fitter disables routing wires near the edge of a secured region, where routing is confined within the region. Figure 4–22 shows the used routing elements on a design with secured regions with the Chip Planner.

Figure 4–22. Chip Planner View of Used Resources *(Note 1)*, *(2)*



Notes to Figure 4–22:

(1) No routing resources reach outside of LogicLock region boundaries, except for global routing signals and signals through interface regions.

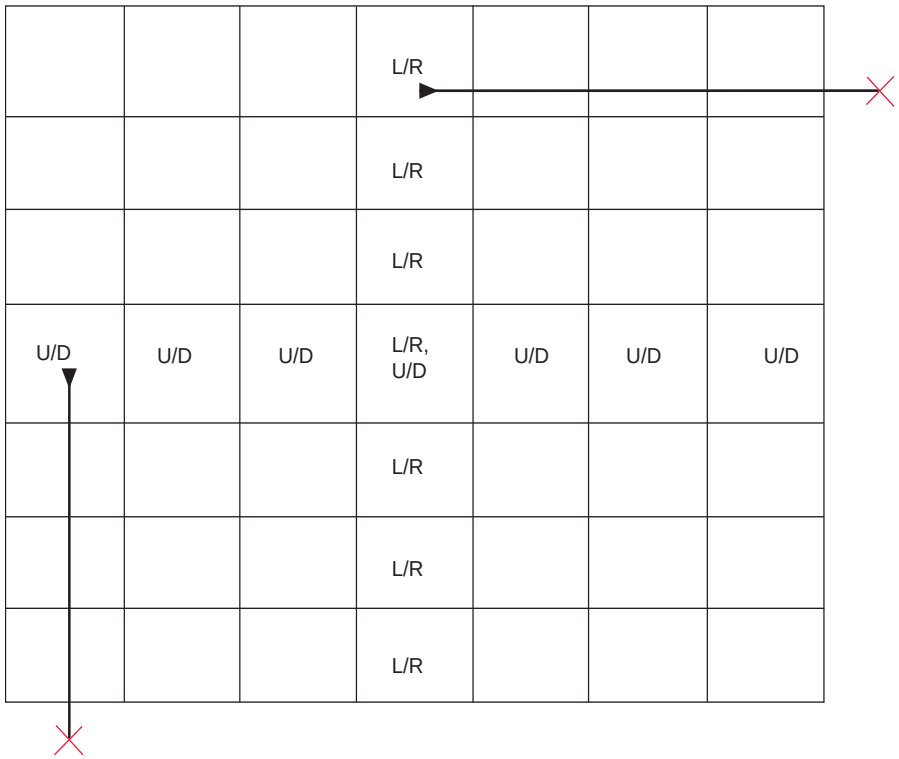
(2) Chip Planner view, configured using Layer settings and using the background color map I/O banks, with only the **Global Routing** and **Used Resources** options selected.

Most long wire connections are not used in secured regions. However, all wire bundles terminate along the edges of the device. Therefore, there are a number of stubbed wires driving toward each edge of the device. If a secured region abuts the device boundary, you can often attain an increase in routability, because all of the routing interconnects that start inside the region and drive toward the edge of the device can be used.

I/O pads along the top and bottom of the device can only use column interconnects to drive into the device fabric. The shortest routing element from the I/O to core logic is a C4 routing wire. I/O pads on the left and right sides of the device can use both C4 and R4 routing elements to reach their LAB destinations. Because column I/Os are restricted to using C4 interconnects going into the device, a four-LAB fence is created around secured regions when the boundary of the secured region is within four-LABs of the top and bottom I/O pads.

Secured regions should be sized at a minimum of 8 × 8 LABs. If a region is smaller than 8 × 8 LABs, a connection between two LABs that violates the secured region boundary may occur. For example, in Figure 4–23, any elements along the middle axis of the 7 × 7 LAB array cannot use any C4 or R4 routing elements, because a C4 routing element would reach outside the secured region.

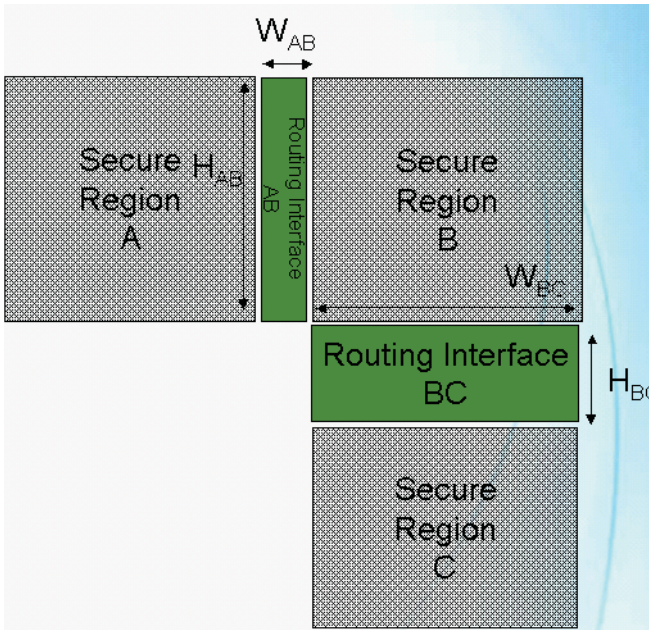
Figure 4–23. 7x7 LAB Array



Number of Signals in Routing Interfaces

In Cyclone III LS devices, every LAB has 68 routing elements (R4) driving horizontally in each direction and 48 routing elements (C4) driving vertically in each direction. The number of connections that can be directly driven by a LAB is 17 connections in the horizontal direction and 12 in the vertical direction. To guarantee routability, Altera recommends that you have a routing interface height of at least one-LAB for every 17 signals routing either left or right, and a routing interface width of one-LAB for every 12 signals routing either up or down. Figure 4–24, Table 4–2, and Table 4–3 illustrate this concept. Figure 4–24 shows three secured regions with two security routing regions; one routing signals horizontally and the other routing signals vertically. Table 4–2 and Table 4–3 list the maximum and recommended number of signals crossing each security region.

Figure 4–24. Signals Crossing a Routing Interface *(Note 1)*, *(2)*, *(3)*



Notes to Figure 4–24:

(1) H_{AB} is the *smaller* height in the region and the height of the routing interface.

(2) Minimum W_{AB} is 1. Changing W_{AB} does not affect Table 4–3. W_{BC} is the *smaller* width in the region and the width of the routing interface.

(3) Minimum H_{BC} is 1. Changing H_{BC} does not affect Table 4–3.

Table 4–2. Maximum Number of Signals Assigned in an Interface Region

To	From		
	A	B	C
A	—	$68 \times H_{AB}$	—
B	$68 \times H_{AB}$	—	$48 \times W_{BC}$
C	—	$48 \times W_{BC}$	—

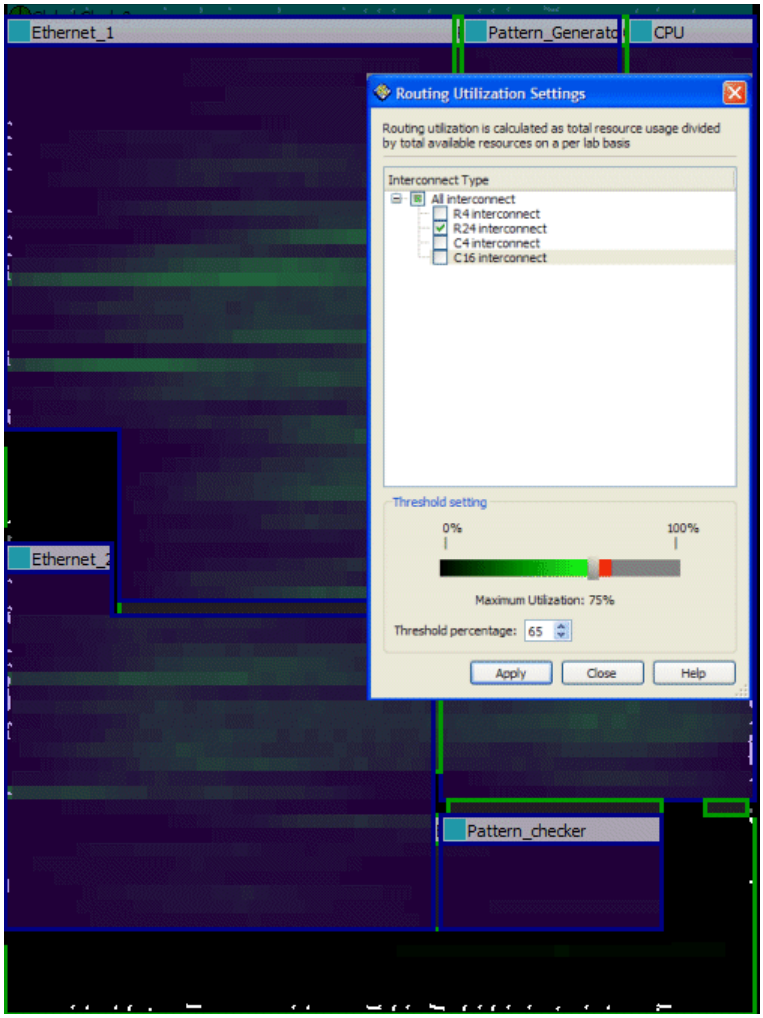
Table 4–3. Recommended Number of Signals to Ensure Routability

To	From		
	A	B	C
A	—	$17 \times H_{AB}$	—
B	$17 \times H_{AB}$	—	$12 \times W_{BC}$
C	—	$12 \times W_{BC}$	—

As a general guideline, keep the security routing interface channel length between the two connecting secured regions as short as possible and the width of the channel as wide as possible. Having a short and wide channel increases the number of routing resources available between two secured regions.

You can use the **Routing Congestion** task in the Chip Planner for a visual representation of the routing utilization between secured regions. Routing resources are filtered by type. Utilization of each routing resource type is highlighted on a color gradient over the range that you specify. This tool can help you adjust region sizes and security routing interface channel widths to help you achieve an optimal floorplan. A design with the **Routing Congestion** task in the Chip Planner and R24 routing utilization is shown in Figure 4–25.

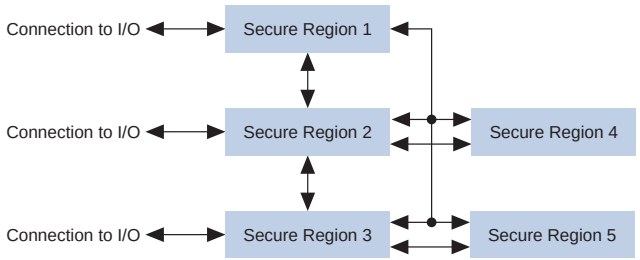
Figure 4–25. Routing Congestion



Application Example: Modify a Fitter-Generated Floorplan for the Design Separation Flow

In this example, the design contains five partitions that must be packed into secured regions. Figure 4–26 shows a block diagram of the design, the entities of the design, and the connectivity between the five secure partitions.

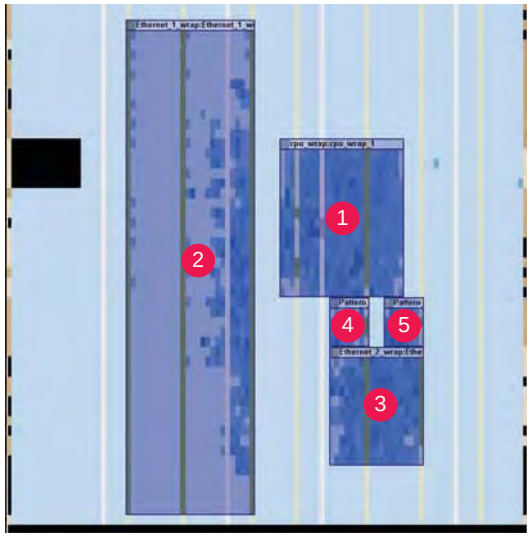
Figure 4–26. Connectivity between Five Secure Partitions



The following procedure outlines a recommended design flow for creating a floorplan for this design:

1. Create a LogicLock region for each partition that must be packed into a secured region.
2. Set each LogicLock region with the attributes of **Auto**, **Floating**, **On**, and **Unsecured** for the **Size**, **State**, **Reserved**, and **Security Attributes** columns, respectively. Running an initial placement with these settings generates non-overlapping LogicLock regions that can be used as an initial floorplan.
3. On the Processing menu, point to **Start** and click **Start Early Timing Estimate** to run an initial place and route. The initial place and route approximates the size of each region and a general idea of how the LogicLock regions must be placed relative to other LogicLock regions to achieve timing closure. The floorplan generated by the early timing estimate is shown in Figure 4–27.

Figure 4–27. Initial Fitter-Generated Floorplan

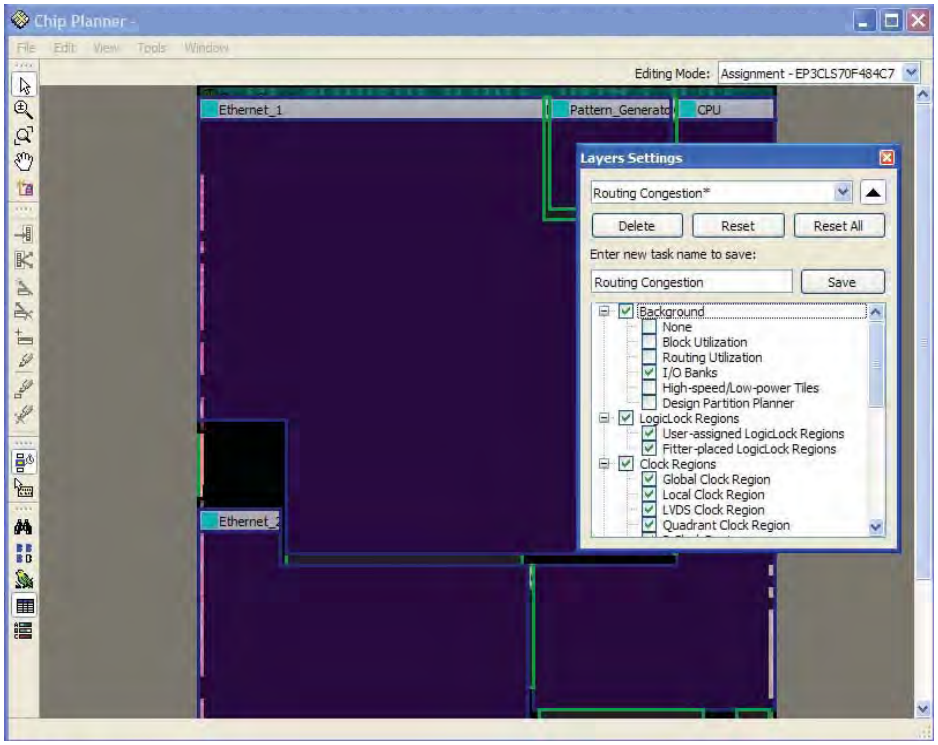


4. In the LogicLock Regions window, select the LogicLock regions, right-click, and then click **Set Size and Origin to Previous Fitter Results**.
5. Use the Design Partition Planner to view the connectivity between the different regions. You can experiment with different placement of the blocks relative to one another by dragging and dropping each design partition. The wire bundles between design partitions help you to determine a placement that has non-overlapping routing channels.

   ☞ You must also consider the connectivity to the I/O banks when arranging your floorplan. You can toggle the display of the connections between the partitions and the I/O banks within the Design Partition Planner to help you properly allocate I/O resources, as well as avoid conflicts between I/O connections and inter-partition signals. To display routing between partitions and the I/O banks, turn on **Display connections to I/O banks** in the **Bundle Configuration** dialog box.

6. Set each LogicLock region to the desired security attribute.
7. In the Chip Planner, adjust the size and placement of each LogicLock region using the relative placement you created with the Design Partition Planner. Note the following considerations when modifying the floorplan:
   a. The floorplan must be complete. If there is unsecured logic that is non-contiguous due to a the placement of a secure region, use an empty reserved LogicLock region to prevent a non-routable placement.
   b. Each secured region must be a minimum of 8 × 8 LABs.
   c. Each region that has I/O pins added as members of the LogicLock region should overlap the I/O bank to which it is connected. You can use the I/O bank background color map to visualize the boundaries between the I/O banks (Figure 4–28).
   d. All global resources (such as clock pins, and PLLs) that are required by unsecured logic must not be covered by a secured region.

Figure 4–28. I/O Banks Layers Setting for Viewing Connectivity of LogicLock Regions to I/O Banks



8. Create security routing interfaces between each of the secured regions. Assign all signals entering or exiting a region to a security routing interface.

   The final floorplan result for this example is shown in Figure 4–29.

Figure 4–29. Final Floorplan

Report Panels

After the Fitter successfully places and routes a design with secured regions, the Quartus II software generates security reports. Use the security reports to review the secured regions, their associated routing interfaces, all inputs and outputs from each secured region, and the I/O bank usage for each secured region. The security reports are located in the Fitter section of the Compilation reports.

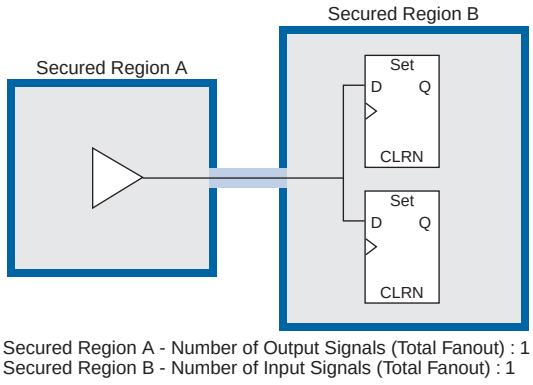
Secured LogicLock Region Summary

This report provides a summary of all secured regions in your design. Table 4–4 describes each column in the Secured LogicLock Region Summary report.

Table 4–4. Secured Logic Region Report

Column Name	Description
Secured LogicLock Region	Lists all secured LogicLock regions in the design.
Security Attribute	Lists the security attribute (unsecured,1, 2, or security routing interface) of the LogicLock region.
Partition Assigned	Lists the design partition assigned to the secured region.
Number of Input Signals (Total Fanout)	Number of inputs and fan-outs into a region. The input counts the number of unique drivers that feed a secured region. Fan-out counts the total number of unique destinations being fed by the input signals into the secured region. Refer to Figure 4–30 for an illustration of input signals and fan-outs to a region.
Number of Output Signals (Total Fanout)	Number of outputs and fan-outs from a region. The output counts the number of unique drivers sourcing a signal from the secured region. Fan-out counts the total number of unique destinations fed by the output signal.

Figure 4–30. Input Signals and Fan-Outs to a Region



Security Routing Interfaces

This report summarizes the security routing interfaces. Table 4–5 describes each column in the Security Routing Interfaces report.

Table 4–5. Security Routing Interface Report

Column Name	Description
Interface Name	Lists all security routing interfaces in the design.
Abutting Region A	First region that the security routing interface abuts (touches the blue border of the secured region).
Abutting Region B	Second region that the security routing interface abuts (touches the blue border of the secured region).
Number of Signals A to B (Total Fanout in B)	Lists the number of signal connections between region A and region B. The counts are shown as signals and fan-outs. Signals list the number of unique drivers from region A. Fan-out lists the number of unique destinations in region B that are fed by region A.
Number of Signals B to A (Total Fanout in A)	Lists the number of signal connections between region B and region A. The counts are shown as signals and fan-outs. Signals list the number of unique drivers from region B. Fan-out lists the number of unique destinations in region A that are fed by region B.

Secured LogicLock Region Inputs and Outputs

This set of reports provides a detailed list of every signal that enters or exits a secured region. There is one report per secured region.

Security I/O Bank Usage

This report displays the secured LogicLock region associated with each I/O bank, lists the number of pins within each region, and lists the number of pins in use. Table 4–6 describes each column in the Secured LogicLock Region Inputs and Outputs report.

Table 4–6. Secured LogicLock Region Input and Output Report

Column Name	Description
I/O Bank	Lists all available I/O banks on the device.
Associated Region	An I/O bank becomes associated with a secured LogicLock region if any portion of the I/O bank is covered by the region. If no secured region covers an I/O bank, “Unsecured Logic” is displayed, and all pins of the I/O bank are available for unsecured use.
Pin Locations Used / Pin Locations Covered by Region	Displays the ratio of pins with a signal assignment in the I/O bank to the number of possible I/O pin assignments.

Quartus Settings File Syntax

This section contains the syntax description for each Quartus Settings File (**.qsf**) assignment in the design separation flow.

LL_SECURITY_ROUTING_INTERFACE

This command changes a LogicLock region assignment to a security routing interface.

Type: Boolean; (ON/OFF - Defaults to OFF)

Syntax:

```
set_global_assignment -name LL_SECURITY_ROUTING_INTERFACE <value> \ -
section_id <section_identifier>LL_REGION_SECURITY_ LEVEL
```

This command identifies the security level of a LogicLock region.

Type: Enumeration—defaults to UNSECURED

- 1
- 2
- UNSECURED

Syntax:

```
set_global_assignment -name LL_REGION_SECURITY_LEVEL <value> \
-section_id <section_identifier>
```

LL_MEMBER_OF_SECURITY_ROUTING_INTERFACE

This command assigns an I/O pin from a secured region to a security routing interface. Both value and section ID denote the name of the routing interface region. <*to*> specifies the name of the signal.

Type: String

Syntax:

```
set_instance_assignment -name \
LL_MEMBER_OF_SECURITY_ROUTING_INTERFACE <value> -to <to> \
-section_id <section_id>
```

LL_SIGNAL_SECURITY_LEVEL

This command sets the security level of a signal. The default value is the security level of the region that generates the signal. This assignment may be used only to lower a security level.

Type: Enumeration

- UNSECURED
- 1
- 2

Syntax:

```
set_instance_assignment -name LL_SIGNAL_SECURITY_LEVEL <value> \
-to <to> -section_id <section_id>
```

Document Revision History

Table 4–7 shows the revision history for this chapter.

Table 4–7. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	Initial release. Content originated from *AN 567: Quartus II Design Separation Flow*.

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

Take an online survey to provide feedback about this handbook chapter.

Section II. Design Guidelines

When designing for large and complex FPGAs, your design and coding styles can impact your quality of results significantly. Designs reflecting synchronous design practices behave predictably reliably, 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 in 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 use of clock and register-control features in device architecture.

- Chapter 6, Recommended HDL Coding Styles

  This chapter discusses Altera megafunctions and provides specific Verilog HDL and VHDL coding examples to insure the Quartus II software infers 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 information.

- Chapter 7, Managing Metastability with the Quartus II Software

  This chapter describes ways you can use the Quartus II software to analyze the average mean time between failures (MTBF) due to metastability caused by synchronization of asynchronous signals, and optimize the design to improve the metastability MTBF.

- 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.

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

QII51006-10.0.0

This chapter provides design recommendations for Altera® devices and describes the Quartus® II Design Assistant, which helps you check your design for violations of design recommendations. 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 the timing performance, logic utilization, and system reliability of your device. Well-coded designs behave in a predictable and reliable manner even when retargeted 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, you should 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 contains the following sections:

- "Synchronous FPGA Design Practices"
- "Design Guidelines" on page 5–3
- "Checking Design Violations With the Design Assistant" on page 5–13
- "Targeting Clock and Register-Control Architectural Features" on page 5–22
- "Targeting Embedded RAM Architectural Features" on page 5–26

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 more information about 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*.

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*.

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 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 you ensure that all the timing requirements of the registers 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 a HardCopy device or if you are prototyping an ASIC.

Fundamentals of Synchronous Design

In a synchronous design, the clock signal controls the activities of all inputs and outputs. 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 several 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 you meet the following timing requirements:

- Before an active clock edge, you must ensure that the data input has been stable for at least the setup time of the register.
- After an active clock edge, you must ensure that 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 in your device.

To meet setup and hold time requirements on all input pins, any inputs to combinational logic that feed 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 information about timing requirements and analysis in the Quartus II software, refer to *About Classic Timing Analysis* or *About TimeQuest Timing Analysis* in Quartus II Help.

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, Altera recommends that you focus on design practices that help you meet design goals consistently than to save device resources with 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 5–3. 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 several 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, you should 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.

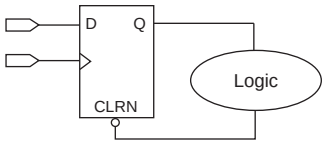
Combinational Logic Structures

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, with either logic elements (LEs) or adaptive logic modules (ALMs). For some cases in which combinational logic feeds registers, the register control signals can 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 TimeQuest Timing Analyzer, refer to *Specifying Timing Constraints and Exceptions (TimeQuest Timing Analyzer)* in Quartus II Help for details about how the TimeQuest analyzer performs recovery and removal analysis.

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. You can implement latches directly with primitives with `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 cannot identify these safe applications.

The Quartus II software **Analyze latches as Synchronous Elements** setting allows you to treat latches as having nontransparent start and end points. Be aware 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, you should not use latches when using formal verification.

Altera recommends that 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 you use two or more consecutive nodes with a single fan-in and a single fan-out 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.

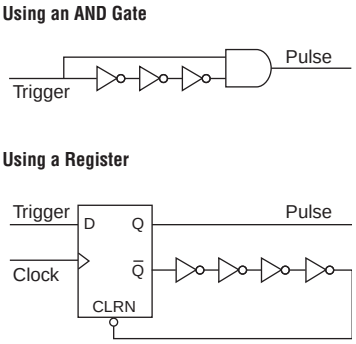
Delays in PLD designs can change with each placement and routing cycle. Effects such as rise and fall time differences and on-chip variation mean that delay chains, especially those placed on clock paths, can cause significant problems in your design. Refer to "Hazards of Asynchronous Design" on page 5–3 for examples of the kinds of problems that delay chains can cause. Avoid using delay chains to prevent these kinds 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

You can use delay chains 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 must 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 feed the gate directly and the path 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.

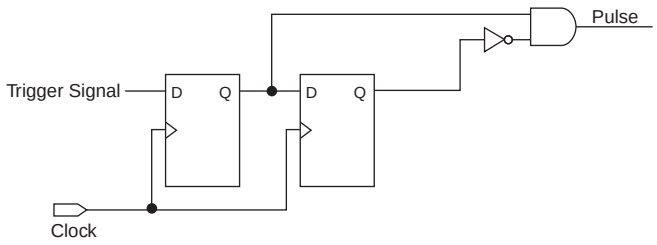
In "Using a Register" (Figure 5–2), a register's output drives the same register's asynchronous reset signal through a delay chain. The register resets itself asynchronously after a certain delay.

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 more information, refer to "Clock Network Resources" on page 5–22.

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.

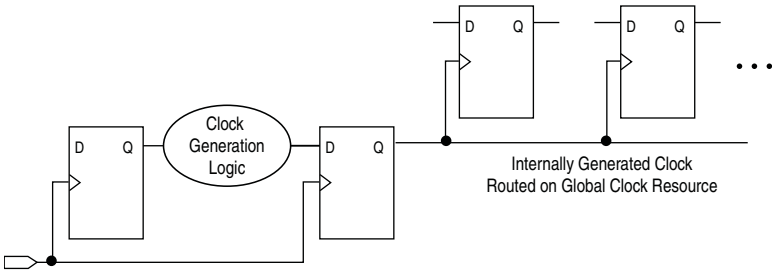
The following sections provide some specific examples and recommendations for avoiding these problems.

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 might 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", 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

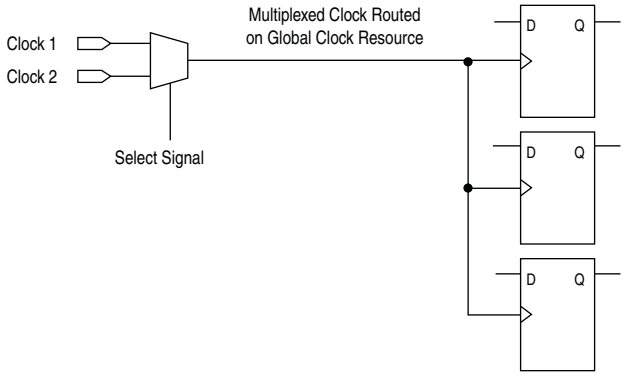
To simplify verification, Altera recommends that you avoid ripple counters in your design. In the past, FPGA designers implemented ripple counters to divide clocks by a power of two because the counters are easy to design and may use fewer gates than their synchronous counterparts. Ripple counters use 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 must be handled properly during timing analysis, which can be difficult and may require you to make complicated timing assignments in your synthesis and placement and routing 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.

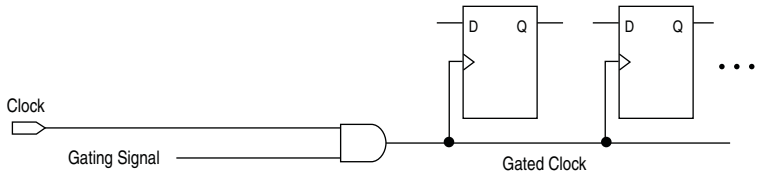
Altera recommends using dedicated hardware to perform clock multiplexing when it is available, instead of using multiplexing logic. For example, you can use the Clock Switchover feature or Clock Control Block available in certain Altera devices. These dedicated hardware blocks ensure that you use global low-skew routing lines and avoid any possible hold time problems on the device due to logic delay on the clock line.

For device-specific information about clocking structures, refer to the appropriate device data sheet or handbook on the Literature page of the Altera website.

Gated Clocks

Gated clocks turn a clock signal on and off using an enable signal that controls 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.

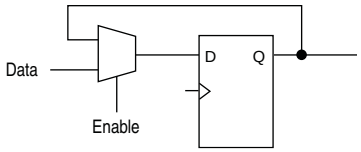
For device-specific information about clocking structures, refer to the appropriate device data sheet or handbook on the Literature page of the Altera website.

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

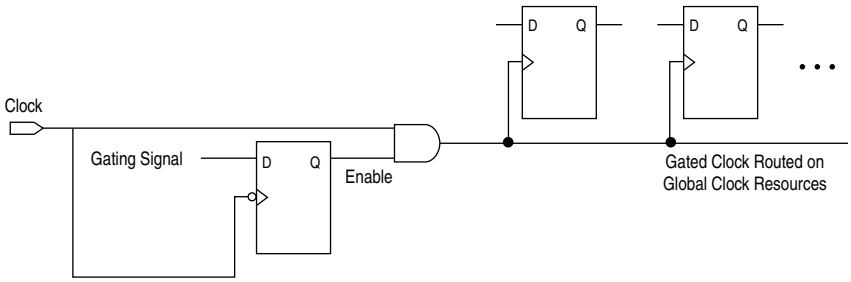


Recommended Clock-Gating Methods

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 might 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 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*.

Power Optimization

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 placement and routing.

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 suite of power analysis and optimization tools to aid you during the design process by delivering fast and accurate estimations of power consumption. For information about the Quartus II PowerPlay suite of power analysis and optimization tools, refer to *About Power Estimation and Analysis* in Quartus II Help.

Metastability

Metastability in Altera designs can be caused by the synchronization of asynchronous signals. You can use the Quartus II software to analyze the mean time between failures (MTBF) due to metastability, thus optimizing the design to improve the metastability MTBF. A high metastability MTBF indicates a more robust design.

For more information about how to ensure complete and accurate metastability analysis, refer to the *Managing Metastability With the Quartus II Software* chapter in volume 1 of the *Quartus II Handbook*.

Incremental Compilation

The incremental compilation feature in the Quartus II software allows you to partition your design, separately compile partitions, and reuse the results for unchanged partitions. Incremental compilation flows require more up-front planning than flat compilations, and generally require you to be more rigorous about following good design practices than flat compilations.

For more information about incremental compilation and floorplan assignments, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.

Checking Design Violations With 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.

This section provides an introduction to the Quartus II design flow with the 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. The Design Assistant supports all Altera devices supported by the Quartus II software.

Quartus II Design Flow with the Design Assistant

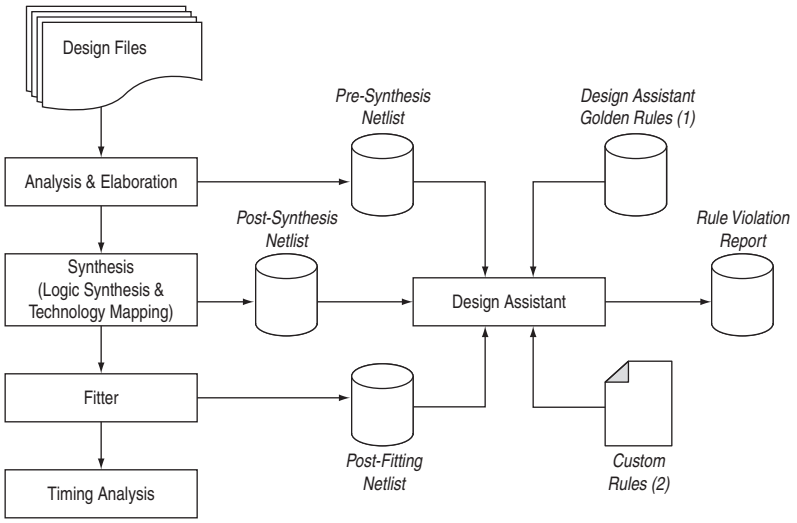
You can run the Design Assistant after Analysis and Elaboration, Analysis and Synthesis, fitting, or a full compilation. To run the Design Assistant, on the Processing menu, point to **Start** and click **Start Design Assistant**.

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.

For more information on enabling or disabling specific Design Assistant rules, refer to *About the Design Assistant* in Quartus II Help.

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 5–15.

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 might 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.

The exact operation of the Design Assistant depends on when you run it:

- 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.

For detailed instructions on Design Assistant settings, refer to *About the Design Assistant* and *Design Assistant Page (Settings Dialog Box)* in Quartus II Help.

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 with a Tcl command.

For instructions on enabling or disabling Design Assistant rules, refer to *Enabling Design Assistant Rules on Nodes, Entities, or Instances*, or *Disabling Design Assistant Rules on Nodes, Entities, or Instances* in Quartus II Help.

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.

For information on the contents of the reports generated by the Design Assistant, refer to *Design Assistant Reports* in Quartus II Help.

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) with 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 5–17.

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 in 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 in <DA_RULE> and </DA_RULE> tags. Table 5–1 describes the attributes of the Rule Attribute section.

Table 5–1. Attributes of 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 5–27.
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 About the Design Assistant in Quartus II Help.
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 in 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 in the Rule Definition section that you can use to declare the properties and conditions:

- <DECLARE>—This subsection contains global nodes that are used in the file. 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. The declarations can be performed with the node declaration command directly without being enclosed in 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 use when checking for violations, refer to *Design Assistant Page (Settings Dialog Box)* in Quartus II Help.

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>
```

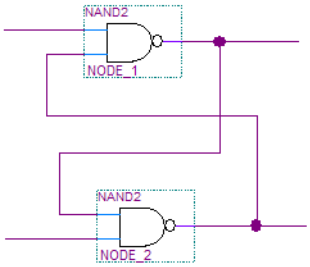
In Example 5–2, the possible SR latch structures are specified in the rule definition section. Codes defined in the <AND></AND> block are tied together, meaning that each statement in the block must be true for the block to be fulfilled (AND gate similarity). In the <OR></OR> block, as long as one statement 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 5–10 and Figure 5–11):

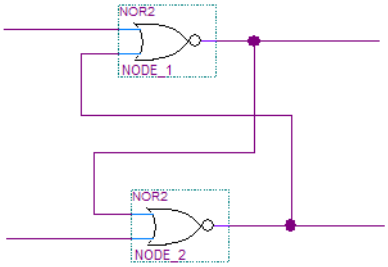
```
<AND>
  <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NAND" TO_NAME="NODE_2"
TO_TYPE="NAND" />
  <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NAND" TO_NAME="NODE_1"
TO_TYPE="NAND" />
</AND>
```

Figure 5–10. 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 5–11. 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 with 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
- There is no 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" />
```

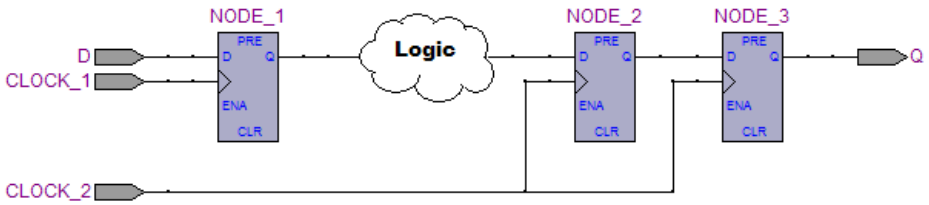
The <FORBID></FORBID> section contains the undesirable condition for the design, which in this case is the undesired configuration of the synchronizer. If the condition is fulfilled, the Design Assistant highlights a rule violation.

The following examples are the undesired conditions from Example 5–3 with their equivalent block diagrams (Figure 5–12 and Figure 5–13):

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" />
```

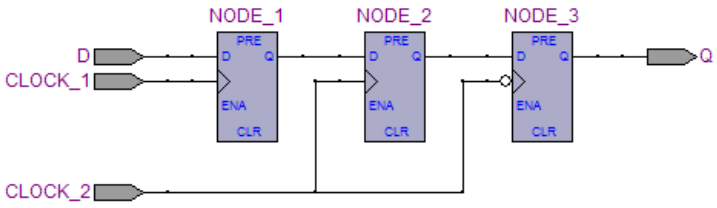
Figure 5–12. Undesired Condition 3



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 5–13. Undesired Condition 4



Targeting Clock and Register-Control Architectural Features

In addition to following general design guidelines, you must 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 with 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 that you limit 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 organized into a hierarchical clock structure that allows many clocks in each device region with low skew and delay. There are typically several dedicated clock pins to drive either global or regional clock networks, and both PLL outputs and internal clocks can drive various clock networks.

To reduce clock skew in a given clock domain and ensure that hold times are met in 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.

Three types of resets are used in synchronous circuits:

- Synchronous Reset
- Asynchronous Reset
- Synchronized Asynchronous Reset—this type is preferred when designing an FPGA circuit

Synchronous Reset

The synchronous reset ensures that the circuit is fully synchronous. You can easily time it with static timing analyzer tools, such as the Quartus II TimeQuest analyzer. The synchronous reset is easier to use with cycle-based simulators.

However, the synchronous reset might require pulse stretchers to guarantee a reset pulse width wide enough to ensure that reset is present during an active edge of the clock. The synchronous reset requires a clock to reset a circuit. If the clock fails to launch, the resulting circuit is not reset.

Asynchronous Reset

The asynchronous reset is the most common form of reset used in circuit designs. Typically, you can insert the asynchronous reset into the device, turn on the global buffer, and connect to the asynchronous reset pin of every register in the device. This method is only advantageous under certain circumstances—you do not need to always reset the register. Unlike the synchronous reset, the asynchronous reset is not inserted in the data path, and does not negatively impact the data arrival times between registers. Reset takes effect immediately, and as soon as the registers receive the reset pulse, the registers are reset. The asynchronous reset is not dependent on the clock.

However, when the reset is deasserted and does not pass the recovery (μt_{SU}) or removal (μt_{H}) time check (both times are checked by the TimeQuest analyzer Recovery and Removal Analysis), the edge is said to have fallen in the metastability zone. Additional time is required to determine the correct state, and the delay can cause the setup time to fail to register downstream, leading to system failure.

The asynchronous reset is susceptible to noise, and a noisy asynchronous reset can cause a spurious reset. You must ensure that the asynchronous reset is debounced and filtered. You can easily enter into a reset asynchronously, but releasing a reset asynchronously can lead to potential problems (also referred to as “reset removal”) with metastability, including the hazards of unwanted situations with synchronous circuits involving feedback.

Synchronized Asynchronous Reset

To avoid potential problems associated with purely synchronous resets and purely asynchronous resets, you can use synchronized asynchronous resets. Synchronized asynchronous resets combine the advantages of synchronous and asynchronous resets. These resets are asynchronously asserted and synchronously deasserted. This takes effect almost instantaneously, and ensures that no data path for speed is involved, and that the circuit is synchronous for timing analysis and is resistant to noise.

Figure 5–14 shows a method for implementing the synchronized asynchronous reset. Altera recommends using synchronizer registers in a similar manner as synchronous resets. However, the asynchronous reset input is gated directly to the CLRN pin of the synchronizer registers and immediately asserts the resulting reset. When the reset is deasserted, logic "1" is clocked through the synchronizers to synchronously deassert the resulting reset.

Figure 5–14. Schematic of Synchronized Asynchronous Reset

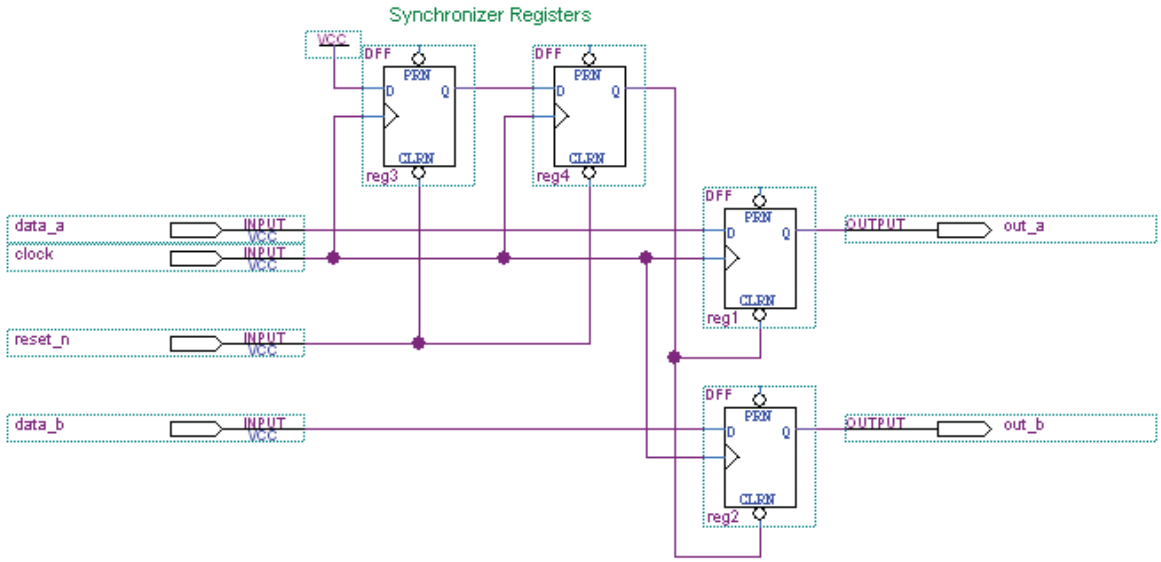


Figure 5–15 shows the equivalent Verilog code. The active edge of the reset is used in the sensitivity list for the blocks in Figure 5–14.

Figure 5–15. Verilog Code for Synchronized Asynchronous Reset

```
module sync_async_reset (
        input   clock,
        input   reset_n,
        input   data_a,
        input   data_b,
        output  out_a,
        output  out_b
        );

reg     reg1, reg2;
reg     reg3, reg4;

assign  out_a   = reg1;
assign  out_b   = reg2;
assign  rst_n   = reg4;

always @(posedge clock, negedge reset_n)
begin
    if (!reset_n)
    begin
        reg3    <= 1'b0;
        reg4    <= 1'b0;
    end
    else
    begin
        reg3    <= 1'b1;
        reg4    <= reg3;
    end
end

always @(posedge clock, negedge rst_n)
begin
    if (!rst_n)
    begin
        reg1    <= 1'b0;
        reg2    <= 1'b0;
    end
    else
    begin
        reg1    <= data_a;
        reg2    <= data_b;
    end
end

endmodule   // sync_async_reset
```

To minimize the metastability effect between the two synchronization registers, and to increase the MTBF, the registers should be located as close as possible in the device to minimize routing delay. Altera recommends you locate the registers in the same logic array block (LAB). The input reset signal (`reset_n`) must be cut with a `set_false_path` command, so the reset that comes from the synchronization register (`rst_n`) can be timed in the TimeQuest analyzer with Recovery and Removal Analysis.

For more information about the TimeQuest analyzer, refer to the *Best Practices for the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

The circuit in Figure 5–14 ensures that the synchronized asynchronous reset is at least one full clock period in length. To extend this time to `n` clock periods, you must increase the number of synchronizer registers to `n + 1`. You must connect the asynchronous input reset (`reset_n`) to the `CLRN` pin of all the synchronizer registers to maintain the asynchronous assertion of the synchronized asynchronous reset.

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 these 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 placement and routing 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 necessary 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 with the MegaWizard™ Plug-In Manager or with 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 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 different 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 that describes the read returns either the old data stored at the memory location or the new data 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 Cyclone and Stratix 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.

Document Revision History

Table 5–2 shows the revision history for this chapter.

Table 5–2. Document Revision History (Part 1 of 2)

Date	Version	Changes Made
July 2010	10.0.0	■ Removed duplicated content and added references to Quartus II Help for Design Assistant settings, Design Assistant rules, Enabling and Disabling Design Assistant Rules, and Viewing Design Assistant reports. ■ Removed information from "Combinational Logic Structures" on page 5–4 ■ Changed heading from "Design Techniques to Save Power" to "Power Optimization" on page 5–12 ■ Added new "Metastability" section ■ Added new "Incremental Compilation" section ■ Added information to "Reset Resources" on page 5–23 ■ Removed "Referenced Documents" section
November 2009	9.1.0	Removed documentation of obsolete rules.
March 2009	9.0.0	No change to content.

Table 5–2. Document Revision History (Part 2 of 2)

Date	Version	Changes Made
November 2008	8.1.0	■ Changed to 8-1/2 x 11 page size ■ Added new section "Custom Rules Coding Examples" on page 5–18 ■ Added paragraph to "Recommended Clock-Gating Methods" on page 5–11 ■ Added new section: "Design Techniques to Save Power" on page 5–12
May 2008	8.0.0	■ 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

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

Take an online survey to provide feedback about this handbook chapter.

6. Recommended HDL Coding Styles

QII51007-10.0.0

This chapter provides Hardware Description Language (HDL) coding style recommendations to ensure optimal synthesis results when targeting Altera® devices.

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, synthesis tools have no information about the purpose or intent of the design. The best optimizations require conscious interaction by the you, the designer.

This chapter includes 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 6–5
- "Inferring Memory Functions from HDL Code" on page 6–12
- "Coding Guidelines for Registers and Latches" on page 6–41
- "General Coding Guidelines" on page 6–51
- "Designing with Low-Level Primitives" on page 6–72

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 handcrafted 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*.

The Altera website also provides design examples for other types of functions and to target specific applications. For more information about design examples, refer to the Design Examples page and the Reference Designs page on the Altera website.

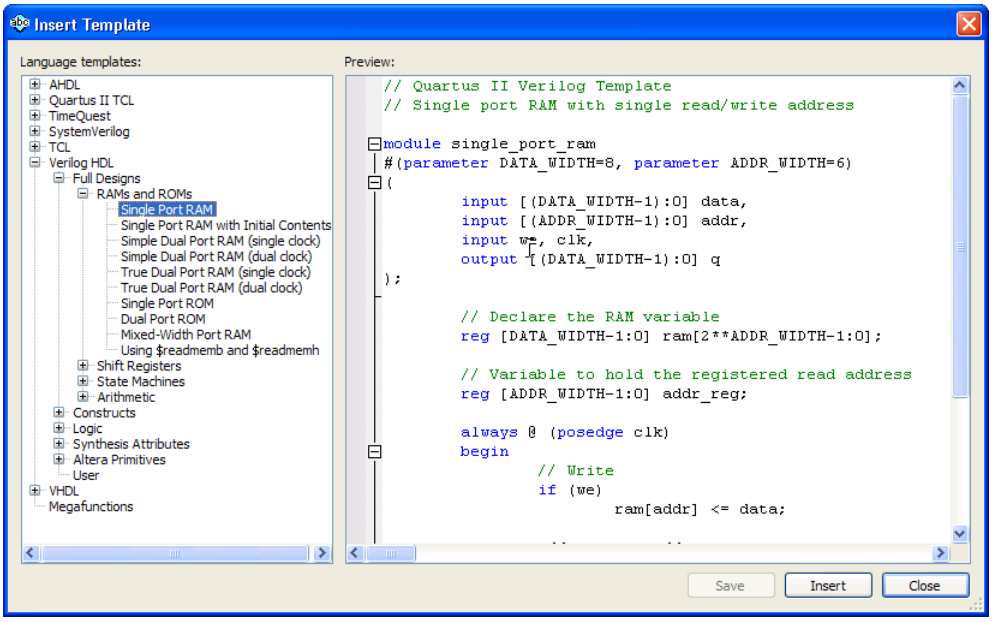
For style recommendations, options, or HDL attributes specific to your synthesis tool (including Quartus® II integrated synthesis and other EDA tools), refer to the tool vendor's documentation or the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Quartus II Language Templates

Many of the Verilog HDL and VHDL examples in this document correspond with examples in the Full Designs section of the Quartus II Templates. You can easily insert examples into your HDL source code using the **Insert Template** dialog box in the Quartus II software user interface, in Figure 6–1.

To open the **Insert Template** dialog box when you have a file open in the Text Editor of the Quartus II software, on the Edit menu, click **Insert Template**. Alternatively, you can right-click in the Text Editor window and click **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 specify 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.

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 device specific dedicated architecture blocks instead of instantiating a megafunction. Synthesis tools, including Quartus II integrated synthesis, recognize certain types of HDL code and automatically infer the appropriate megafunction or map directly to device atoms. 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 specific function, follow the guidelines and coding examples in "Inferring Multiplier and DSP Functions from HDL Code" on page 6–5 and "Inferring Memory Functions from HDL Code" on page 6–12 to ensure your HDL code infers the appropriate function.

You can infer or instantiate megafunctions to target some Altera device-specific architecture features such as memory and DSP blocks. You must instantiate megafunctions to target certain other device and high-speed features, such as LVDS drivers, phase-locked loops (PLLs), transceivers, and double-data rate input/output (DDIO) circuitry.

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 software 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, either **_inst.v**, or **_inst.vhd**, or **_inst.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 other EDA synthesis tool (**_bb.v**).
- If you enable the option to generate a synthesis timing and resource 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 6–4 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

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.
<*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 timing and resource estimation netlist—Megafunction netlist may be used by other EDA synthesis tools to improve timing and resource estimations.

Notes to Table 6–1:

(1) The MegaWizard Plug-In Manager generates either the **.v**, **.vhd**, or **.tdf** file, 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 other EDA synthesis tools (that is, tools other than Quartus II integrated synthesis), you can optionally create a netlist for timing and resource 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 other EDA synthesis tools to better report timing and resource 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 netlist** under **Timing and resource estimation** 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, either <*output file*>**.v** or <*output file*>**.vhd,** for placement and routing in the project created with the Quartus II software.

Because your synthesis tool may call the Quartus II software in the background to generate this netlist, turning on this option might not be required.

For information about support for timing and resource 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.

For a list of the megafunction ports and parameters, refer to the specific megafunction in the Quartus II Help. You can also refer to the IP and Megafunction page on the Altera website.

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 6–3.

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:

- "Inferring Multipliers from HDL Code"
- "Inferring Multiply-Accumulators and Multiply-Adders from HDL Code" on page 6–7

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 the Altera website.

Inferring Multipliers 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 the Altera 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 a megafunction or DSP block atoms. Each example fits into one DSP block 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
```

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;

ARCHITECTURE rtl OF signed_mult IS
BEGIN
   result <= a * b;
END rtl;
```

Inferring Multiply-Accumulators and Multiply-Adders 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 in Example 6–5 through Example 6–8 on pages 6–9 through 6–12 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 three. You can remove the registers in your design to reduce the latency.

Example 6–5. Verilog HDL Unsigned Multiply-Accumulator

```
module unsig_altmult_accum (dataout, dataa, datab, clk, aclr, clken);
   input [7:0] dataa, datab;
   input clk, aclr, clken;
   output reg[16:0] dataout;

   reg [7:0] dataa_reg, datab_reg;
   reg [15:0] multa_reg;
   wire [15:0] multa;
   wire [16:0] 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 <= 17'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;
           mult1_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:

- "Inferring RAM functions from HDL Code" on page 6–13
- "Inferring ROM Functions from HDL Code" on page 6–33
- "Shift Registers—Inferring the ALTSHIFT_TAPS Megafunction from HDL Code" on page 6–37

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.

Inferring RAM functions 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 multi-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.

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 (two read ports and two write ports) RAM blocks that map to the memory blocks in certain Altera devices.

Some tools (such as the Quartus II software) also infer memory blocks for array variables and signals that are referenced (read/written) by two indices, to recognize mixed-width and byte-enabled RAMs for certain coding styles.

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.

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 situation occurs. If the entity or module contains any additional logic outside the RAM block, this logic cannot be verified because it also must be treated as a black box for formal verification.

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. Because memory blocks in the newest devices from Altera are synchronous, 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 memory offers several advantages over asynchronous memory, including higher frequencies and thus higher memory bandwidth, increased reliability, and less standby power. In many designs with asynchronous memory, the memory interfaces with synchronous logic so that the conversion to synchronous memory design is straightforward. To convert asynchronous memory you can move registers from the data path into the memory block.

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. The recommended coding style for synchronous memories is to create your design with a registered read output.
- 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, or may require external bypass logic, depending on the target device architecture.

The synchronous memory structures in Altera devices can differ from the structures in other vendors' devices. For best results, match your design to the target device architecture.

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. Altera recommends against 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 6–31 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, reset, 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. You can use the address stall feature as a read address clock enable in Stratix II, Cyclone® II, Arria® GX, and other newer devices to avoid this limitation. Check the documentation for 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 behavior 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 in the following examples. 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 synchronous 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 synchronous memory block, the device functionality and gate-level simulation results would not match the HDL description or functional 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 matches 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 certain device 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` set to `"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. In some cases, this attribute prevents the synthesis tool from using extra logic to implement the memory block, or can allow memory inference when it would otherwise be impossible.

Synchronous RAM blocks require a synchronous read, so Quartus II integrated synthesis packs either data output registers or read address registers into the RAM block. When the read address registers are packed into the RAM block, the read address signals connected to the RAM block contain the next value of the read address signals indexing the HDL variable, which impacts which clock cycle the read and the write occur and changes the read-during-write conditions. Therefore, bypass logic may still be added to the design to preserve the read-during-write behavior, even if the "`no_rw_check`" attribute is set.

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 next subsection describes how you control the logic implementation in the Altera device, and 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.

Controlling Inference and Implementation in Device RAM Blocks

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 RAM blocks for all sizes with the **Allow Any RAM Size for Recognition** option in the **More Analysis & Synthesis Settings** dialog box.

Some synthesis tools provide options to control the implementation of inferred RAM blocks for Altera devices with synchronous 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 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*.

If you want to control the implementation after the RAM function is inferred during synthesis, you can set the `ram_block_type` parameter of the ALTSYNCRAM megafunction. In the Assignment Editor, select **Parameters** in the **Categories** list. You can use the **Node Finder** or drag the appropriate instance from the Project Navigator window to enter the RAM hierarchical instance name. Type `ram_block_type` as the **Parameter Name** and type one of the following memory types supported by your target device family in the **Value** field: "`M-RAM`", "`M4K`", "`M20K`", "`M512`", "`M9K`", "`M144K`", or "`MLAB`".

You can also specify the maximum depth of memory blocks used to infer RAM or ROM in your design. Apply the `max_depth` synthesis attribute to the declaration of a variable that represents a RAM or ROM in your design file. For example:

```
// Limit the depth of the memory blocks implement "ram" to 512
// This forces the software to use two M512 blocks instead of one M4K
block to implement this RAM

(* max_depth = 512 *) reg [7:0] ram[0:1023];
```

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 the Old Data Read-During-Write coding style for most RAM blocks as long as your design does not require the RAM location's new value when you perform a simultaneous read and write 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 in Example 6–11 and Example 6–12 map directly into Altera synchronous 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 6–18.

The simple dual-port RAM 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 address, and same 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 example. By itself, the code 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. 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;
```

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 using 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 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 synchronous 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 synchronous 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 that there is 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 occurs on the same port for the same address, the read operation may behave as follows:

- **Read new data**—This mode matches the behavior of synchronous memory blocks.
- **Read old data**—This mode is supported only by synchronous memory blocks in Arria II GX, Cyclone III, Stratix III, and newer device families. This behavior is not possible in memory blocks of older families.

When a read and write operation occurs 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 synchronous memory block.
- **Read old data**—This behavior is supported by synchronous memory blocks.

The Verilog HDL single-clock code sample in Example 6–17 maps directly into Altera synchronous 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. 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];
    end
    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 operation and write operation occurs at the same time for the same address on the same port or mixed ports. This behavior is supported only in the memory blocks of Arria II GX, Cyclone III, Stratix III, and newer device families, and is not inferred as memory for older 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 in Example 6–18 maps directly into Altera synchronous 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. Simultaneous write operations 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 6–18. 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 array((2**ADDR_WIDTH - 1) 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;
```

Mixed-Width Dual-Port RAM

The RAM code examples in Example 6–19 through Example 6–22 show SystemVerilog and VHDL code that infers RAM with data ports with different widths. This type of logic is not supported in Verilog-1995 or Verilog-2001 because of the requirement for a multi-dimensional array to model the different read width, write width, or both. Different synthesis tools may differ in their support for these memories. This section describes the inference rules for Quartus II integrated synthesis.

The first dimension of the multi-dimensional packed array represents the ratio of the wider port to the narrower port, and the second dimension represents the narrower port width. The read and write port widths must specify a read or write ratio supported by the memory blocks in the target device, or the synthesis tool does not infer a RAM.

Refer to the Quartus II Templates for parameterized examples that you can use for supported combinations of read and write widths, and true dual port RAM examples with two read ports and two write ports for mixed-width writes and reads.

Example 6–19. SystemVerilog Mixed-Width RAM with Read Width Smaller than Write Width

```
module mixed_width_ram    // 256x32 write and 1024x8 read
(
       input [7:0] waddr,
       input [31:0] wdata,
       input we, clk,
       input [9:0] raddr,
       output [7:0] q
);
    logic [3:0][7:0] ram[0:255];
    always_ff@(posedge clk)
        begin
           if(we) ram[waddr] <= wdata;
           q <= ram[raddr / 4][raddr % 4];
        end
endmodule : mixed_width_ram
```

Example 6–20. SystemVerilog Mixed-Width RAM with Read Width Larger than Write Width

```
module mixed_width_ram     // 1024x8 write and 256x32 read
(
       input [9:0] waddr,
       input [31:0] wdata,
       input we, clk,
       input [7:0] raddr,
       output [9:0] q
);
    logic [3:0][7:0] ram[0:255];
    always_ff@(posedge clk)
         begin
           if(we) ram[waddr / 4][waddr % 4] <= wdata;
           q <= ram[raddr];
         end
endmodule : mixed_width_ram
```

Example 6–21. VHDL Mixed-Width RAM with Read Width Smaller than Write Width

```
library ieee;
use ieee.std_logic_1164.all;

package ram_types is
   type word_t is array (0 to 3) of std_logic_vector(7 downto 0);
   type ram_t is array (0 to 255) of word_t;
end ram_types;

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.ram_types.all;

entity mixed_width_ram is
   port (
      we, clk : in  std_logic;
      waddr   : in  integer range 0 to 255;
      wdata   : in  word_t;
      raddr   : in  integer range 0 to 1023;
      q       : out std_logic_vector(7 downto 0));
end mixed_width_ram;

architecture rtl of mixed_width_ram is
   signal ram : ram_t;
begin  -- rtl
   process(clk, we)
   begin
      if(rising_edge(clk)) then
         if(we = '1') then
            ram(waddr) <= wdata;
         end if;
         q <= ram(raddr / 4 )(raddr mod 4);
      end if;
   end process;
end rtl;
```

Example 6–22. VHDL Mixed-Width RAM with Read Width Larger than Write Width

```
library ieee;
use ieee.std_logic_1164.all;

package ram_types is
   type word_t is array (0 to 3) of std_logic_vector(7 downto 0);
   type ram_t is array (0 to 255) of word_t;
end ram_types;

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.ram_types.all;

entity mixed_width_ram is
   port (
      we, clk : in  std_logic;
      waddr   : in  integer range 0 to 1023;
      wdata   : in  std_logic_vector(7 downto 0);
      raddr   : in  integer range 0 to 255;
      q       : out word_t);
end mixed_width_ram;

architecture rtl of mixed_width_ram is
   signal ram : ram_t;
begin  -- rtl
   process(clk, we)
   begin
      if(rising_edge(clk)) then
         if(we = '1') then
            ram(waddr / 4)(waddr mod 4) <= wdata;
         end if;
         q <= ram(raddr);
      end if;
   end process;
end rtl;
```

RAM with Byte-Enable Signals

The RAM code examples in Example 6–23 and Example 6–24 show SystemVerilog and VHDL code that infers RAM with controls for writing single bytes into the memory word, or byte-enable signals. Byte enables are modeled by creating write expressions with two indices and writing part of a RAM "word." With these implementations, you can also write more than one byte at once by enabling the appropriate byte enables.

This type of logic is not supported in Verilog-1995 or Verilog-2001 because of the requirement for a multidimensional array. Different synthesis tools may differ in their support for these memories. This section describes the inference rules for Quartus II integrated synthesis.

Refer to the Quartus II Templates for parameterized examples that you can use for different address widths, and true dual port RAM examples with two read ports and two write ports.

Example 6–23. SystemVerilog Simple Dual-Port Synchronous RAM with Byte Enable

```
module byte_enabled_simple_dual_port_ram
(
    input we, clk,
    input [5:0] waddr, raddr, // address width = 6
    input [3:0] be,     // 4 bytes per word
    input [31:0] wdata,  // byte width = 8, 4 bytes per word
    output reg [31:0] q  // byte width = 8, 4 bytes per word
);
   // use a multi-dimensional packed array
   //to model individual bytes within the word
   logic [3:0][7:0] ram[0:63];// # words = 1 << address width

   always_ff@(posedge clk)
   begin
       if(we) begin
           if(be[0]) ram[waddr][0] <= wdata[7:0];
           if(be[1]) ram[waddr][1] <= wdata[15:8];
           if(be[2]) ram[waddr][2] <= wdata[23:16];
      if(be[3]) ram[waddr][3] <= wdata[31:24];
       end
   q <= ram[raddr];
   end
endmodule
```

Example 6–24. VHDL Simple Dual-Port Synchronous RAM with Byte Enable

```
library ieee;
use ieee.std_logic_1164.all;
library work;

entity byte_enabled_simple_dual_port_ram is
port (
   we, clk : in  std_logic;
   waddr, raddr : in  integer range 0 to 63 ;     -- address width = 6
   be      : in  std_logic_vector (3 downto 0);   -- 4 bytes per word
   wdata   : in  std_logic_vector(31 downto 0);    -- byte width = 8
   q       : out std_logic_vector(31 downto 0) ); -- byte width = 8
end byte_enabled_simple_dual_port_ram;

architecture rtl of byte_enabled_simple_dual_port_ram is
  --  build up 2D array to hold the memory
   type word_t is array (0 to 3) of std_logic_vector(7 downto 0);
   type ram_t is array (0 to 63) of word_t;

   signal ram : ram_t;
   signal q_local : word_t;

   begin  -- Re-organize the read data from the RAM to match the output
      unpack: for i in 0 to 3 generate
         q(8*(i+1) - 1 downto 8*i) <= q_local(i);
   end generate unpack;

   process(clk)
   begin
      if(rising_edge(clk)) then
         if(we = '1') then
            if(be(0) = '1') then
               ram(waddr)(0) <= wdata(7 downto 0);
            end if;
            if be(1) = '1' then
               ram(waddr)(1) <= wdata(15 downto 8);
            end if;
            if be(2) = '1' then
               ram(waddr)(2) <= wdata(23 downto 16);
            end if;
            if be(3) = '1' then
               ram(waddr)(3) <= wdata(31 downto 24);
            end if;
         end if;
         q_local <= ram(raddr);
      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 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, the power-up output of `0` is maintained until the first valid read cycle. The 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 is seen, regardless of the enable status. 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** file for the inferred RAM. Example 6–25 shows Verilog HDL code that infers a simple dual-port RAM block and corresponding **.mif** file.

Example 6–25. 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 initialization and ROM initialization work identically in synthesis and simulation. Example 6–26 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 or the example in the Templates for the Quartus II software for details about the format of the **ram.txt** file.

Example 6–26. 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 6–27 shows VHDL code that infers a simple dual-port RAM block and corresponding **.mif** file.

Example 6–27. 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;
```

Inferring ROM Functions 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, and 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 in the **More Analysis & Synthesis Settings** dialog box.

Some synthesis tools provide options to control the implementation of inferred ROM blocks for Altera devices with synchronous 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*.

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. When you are using a formal verification flow, Altera recommends that you instantiate ROM megafunction blocks in separate entities or modules that contain only the ROM logic, because you may need to treat the entity or module as a black box during formal verification.

The Verilog HDL and VHDL code samples in Example 6–28 through Example 6–31 on pages 6–35 through 6–37 infer synchronous ROM blocks. Depending on the device family's dedicated RAM architecture, the ROM logic may have to be synchronous; refer to the device family handbook for details.

For device architectures with synchronous RAM blocks, such as the Arria series, Cyclone series, or Stratix series devices and newer device families, either the address or the output must be registered for synthesis software to infer a ROM block. When your design uses output registers, the synthesis software implements registers from the input registers of the RAM block without affecting the functionality of the ROM. 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.

The ROM code examples in Example 6–28 through Example 6–31 on pages 6–35 through 6–37 map directly to the Altera memory architecture.

Example 6–28. 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 6–29. 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 6–30. 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 6–31. 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 you might 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 design on the **EDA Tool Settings** page of the **Settings** dialog box in the Quartus II software.

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.

Quartus II integrated synthesis uses the following guidelines which are common in other EDA tools. The Quartus II 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 **Auto Shift Register Recognition** setting is set to **Auto**, Quartus II integrated synthesis uses the **Optimization Technique** setting, logic and RAM utilization information about the design, and timing information from **Timing-Driven Synthesis** to determine which shift registers are implemented in RAM blocks for logic.

- 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 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 in Example 6–32 and Example 6–33 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 and maps it to RAM in supported devices, which may be placed in dedicated RAM blocks or MLAB memory. If the length of the register is less than 64 bits, the software implements the shift register in logic.

Example 6–32. 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 6–33. 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 in Example 6–34 and Example 6–35 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, which is allowed placement in dedicated RAM blocks or MLAB memory.

Example 6–34. 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 6–35. 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 6–43
- "Latches" on page 6–47

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, 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, and 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.

When an asynchronous load (`aload`) signal is available in the device registers, 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 a 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 on the Altera website.

Designers typically use an explicit reset signal for the design, which forces all registers into their appropriate values after reset. Altera recommends this practice to reset the device after power-up to restore the proper state if there is any doubt about the power-up conditions of the device.

You can make 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, you can 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.

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 signal inference 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. When the Quartus II software reads the default values, the synthesized behavior matches the power-up state of the HDL code during a functional simulation.

For example, the code samples in Example 6–36 and Example 6–37 both infer a register for `q` and set its power-up level to high.

Example 6–36. Verilog Register with High Power-Up Value

```
reg q = 1'b1; //q has a default value of '1'

always @ (posedge clk)
begin
   q <= d;
end
```

Example 6–37. VHDL Register with High Power-Up Level

```
SIGNAL q : STD_LOGIC := '1'; -- q has a default value of '1'

PROCESS (clk, reset)
BEGIN
   IF (rising_edge(clk)) THEN
      q <= d;
   END IF;
END PROCESS;
```

If the target device architecture does not support two asynchronous control signals, such as `aclr` and `aload`, you cannot set a different power-up state and reset state. If the NOT-gate push-back algorithm creates logic to set a register to 1, that register will power-up high. If you set a different power-up condition through a synthesis assignment or initial value, the power-up level is ignored during synthesis.

Secondary Register Control Signals Such as Clear and Clock Enable

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 achieving 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, 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. Therefore, 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 6–38) does not have `adata` on the sensitivity list, but the VHDL example (Example 6–39) 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 6–38. 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 6–39. 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, because Stratix II devices have more flexibility than Stratix devices with respect to secondary control signals, 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 in Example 6–38 (after adding the control signals to the module declaration).

Example 6–40. 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 in Example 6–39 on page 6–45 (after adding the control signals to the entity declaration).

Example 6–41. 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 6–48.

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 code sample in Example 6–42 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 6–42. 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 required by your design, 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 both a cover term to ensure there is no glitching and 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, such as 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 optimizations performed by the Quartus II software, such as physical synthesis netlist optimizations in the Fitter, maintain the hazard-free performance.

To ensure hazard-free behavior, only one control input can 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 in Example 6–43 infers a S-R latch correctly in the Quartus II software.

Example 6–43. 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 in Example 6–44 infers a D-type latch correctly in the Quartus II software.

Example 6–44. 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:

```
assign latch_out = (~en & latch_out) | (en & data);
```

The behavior of the assignment is similar to a latch, but it may not function correctly as a latch, and its timing is not analyzed as a latch.

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, 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 6–52. This section provides recommendations for multiplexing clock signals.
- "Adder Trees" on page 6–55. 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 6–58. This section helps ensure the best results when you use state machines.
- "Multiplexers" on page 6–65. 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 6–70. 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 in Example 6–45 and Example 6–46 show Verilog HDL and VHDL code that creates tri-state bidirectional signals.

Example 6–45. Verilog HDL Tri-State Signal

```
module tristate (myinput, myenable, mybidir);
   input myinput, myenable;
   inout mybidir;
   assign mybidir = (myenable ? myinput : 1'bZ);
endmodule
```

Example 6–46. 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.

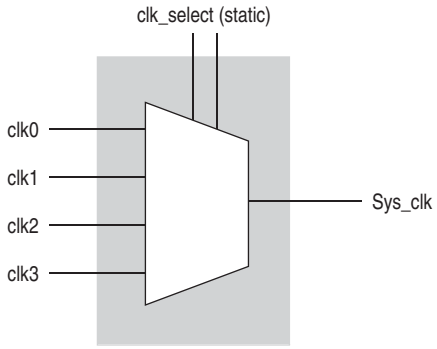
Altera recommends using dedicated hardware to perform clock multiplexing when it is available, instead of using multiplexing logic. For example, you can use the Clock Switchover feature or the Clock Control Block available in certain Altera devices. These dedicated hardware blocks avoid glitches, ensure 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.

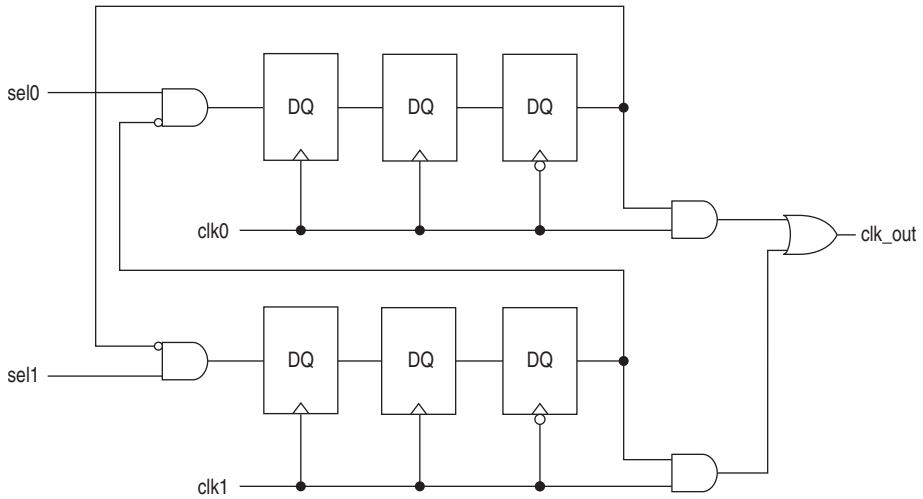
Figure 6–2. Simple Clock Multiplexer in a 6-Input LUT



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 in Figure 6–3.

Figure 6–3. Glitch-Free Clock Multiplexer Structure



This structure can be generalized for any number of clock channels. Example 6–47 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.

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 6–47. 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 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 6–48 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 6–48. 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 6–48 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 6–49 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 6–49. 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 by forcing them into the reset state. All other registers in the design are not protected this way. If the design has asynchronous inputs, Altera recommends using a synchronization register chain instead of relying on the safe state machine option.

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 6–63, 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. For more information, refer too "SystemVerilog State Machine Coding Example" on page 6–62.
- Represent the states in a state machine with the parameter data types in Verilog-1995 and Verilog-2001, and use the parameters to make state assignments. For more information, refer too"Verilog-2001 State Machine Coding Example" on page 6–60. This parameter 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 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 6–50).

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 6–50. 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`, for 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` in Example 6–51 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 in Example 6–51. 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 6–51. 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 6–52).

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 6–52. 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.

For more information, refer to the *Advanced Synthesis Cookbook: A Design Guide for Stratix II, Stratix III, and Stratix IV Devices*.

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. The default setting **Auto** for this option uses the optimization when it is most likely to benefit the optimization targets for your design. You can turn the option on or off specifically to have more control over its use.

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 toward optimizing multiplexer structures for best results.

Binary Multiplexers

Binary multiplexers select inputs based on binary-encoded selection bits. Example 6–53 shows Verilog HDL code for two ways to describe a simple 4:1 binary multiplexer.

Example 6–53. Verilog HDL Binary-Encoded Multiplexers

```
case (sel)
   2'b00: z = a;
   2'b01: z = b;
   2'b10: z = c;
   2'b11: z = d;
endcase
```

The Stratix series features 6-input look up tables (LUTs) which are perfectly suited for 4:1 multiplexer building blocks (4 data and 2 select inputs). The extended input mode facilitates implementing 8:1 blocks, and the fractured mode handles residual 2:1 multiplexer pairs. For device families using 4-input LUTs, such as the Cyclone series and Stratix devices, the 4:1 binary multiplexer is efficiently implemented by using two 4-input LUTs. Larger binary multiplexers are decomposed by the synthesis tool into 4:1 multiplexer blocks, possibly with a residual 2:1 multiplexer at the head.

Selector Multiplexers

Selector multiplexers have a separate select line for each data input. The select lines for the multiplexer are one-hot encoded. Example 6–54 shows a simple Verilog HDL code example describing a one-hot selector multiplexer.

Example 6–54. 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. An N-input selector multiplexer of this structure is slightly less efficient in implementation than a binary multiplexer. However, in many cases the select signal is the output of a decoder, in which case Quartus II Synthesis will try to combine the selector and decoder into a binary multiplexer.

Priority Multiplexers

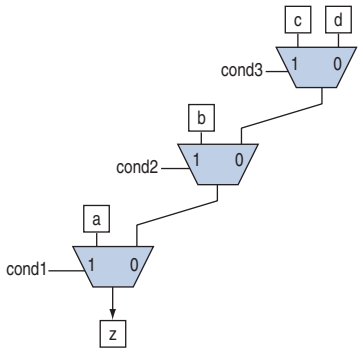
In priority multiplexers, the select logic implies a priority. The options to select the correct item must be checked in a specific order based on signal priority. These structures commonly are created from `IF`, `ELSE`, `WHEN`, `SELECT`, and `?:` statements in VHDL or Verilog HDL. The example VHDL code in Example 6–55 probably results in the schematic implementation illustrated in Figure 6–4.

Example 6–55. 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 in Figure 6–4 form a chain, evaluating each condition or select bit sequentially.

Figure 6–4. Priority Multiplexer Implementation of an IF Statement



Depending on the number of multiplexers in the chain, the timing delay through this chain can become large, especially for device families with 4-input LUTs.

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.

Implicit Defaults in If Statements

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.

There are several ways you can simplify multiplexed logic and remove unneeded 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 reduce default cases and flatten the multiplexer. Examine whether the default "ELSE IF" conditions are don't care cases. You may be able to create a default ELSE statement to make the behavior explicit. Avoid unnecessary default conditions in your multiplexer logic to reduce the complexity and logic utilization required to implement your design.

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 specify 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.

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.

Cyclic Redundancy Check Functions

CRC computations are used heavily by communications protocols and storage devices to detect any corruption of 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 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 that 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, and 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 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 6–43. You can check the register equations in 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, which is 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):

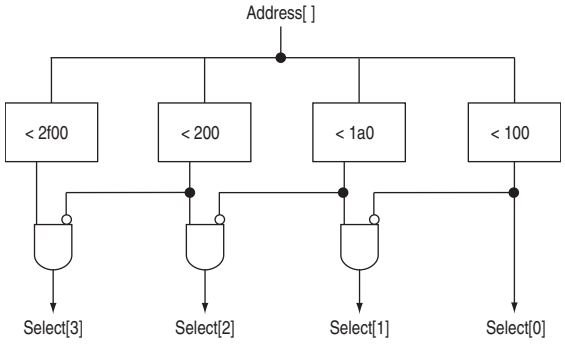
```
wire [6:0] a,b;
wire [7:0] tmp = a - b;
wire alb = tmp[7]
```

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 in Figure 6–5. This type of logic occurs frequently in address decoders.

Figure 6–5. 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*.

Document Revision History

Table 6–2 shows the revision history for this chapter.

Table 6–2. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Added support for mixed-width RAM ■ Updated support for no_rw_check for infering RAM blocks ■ Added support for byte-enable
November 2009	9.1.0	■ Updated support for Controlling Inference and Implementation in Device RAM Blocks ■ Updated support for Shift Registers
March 2009	9.0.0	■ Corrected and updated several examples ■ Added support for Arria II GX devices ■ Other minor changes to chapter
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	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

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

Take an online survey to provide feedback about this handbook chapter.

7. Managing Metastability with the Quartus II Software

QII51018-10.0.0

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 caused by synchronization of 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 caused by signal synchronization.

Introduction

All registers in digital devices, such as FPGAs, have defined signal-timing requirements that allow each register to correctly capture data at its input ports and produce an output signal. To ensure reliable operation, the input to a register must be stable for a minimum amount of time before the clock edge (register setup time or t_{SU}) and a minimum amount of time after the clock edge (register hold time or t_H). The register output is available after a specified clock-to-output delay (t_{CO}).

If the data violates the setup or hold time requirements, the output of the register might go into a metastable state. In a metastable state, the voltage at 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 that metastability does not occur. Metastability problems commonly occur when a signal is transferred between circuitry in unrelated or asynchronous clock domains, because the signal can arrive at any time relative to the destination clock.

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 in the context of your entire system and taking in account 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 on the Altera website. 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* on the Altera website.

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 Timing Analyzer. Both typical and worst-case MBTF values are generated for select device families.

For information about device and version support for the metastability features in the Quartus II software, refer to the Quartus II Help.

This chapter contains the following topics:

- "Metastability Analysis in the Quartus II Software"
- "Metastability and MTBF Reporting" on page 7–5
- "MTBF Optimization" on page 7–8
- "Reducing Metastability Effects" on page 7–9
- "Scripting Support" on page 7–11

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 a standard of three registers provides better metastability protection.

The 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 your design requires longer synchronizer chains.

This section contains the following subsections:

- "Synchronization Register Chains"
- "Identifying Synchronizers for Metastability Analysis" on page 7–4
- "How Timing Constraints Affect Synchronizer Identification and Metastability Analysis" on page 7–4

For information about the reports generated by the timing analyzer, refer to "Metastability and MTBF Reporting" on page 7–5. For more information about optimizing the MTBF, refer to "MTBF Optimization" on page 7–8.

Synchronization Register Chains

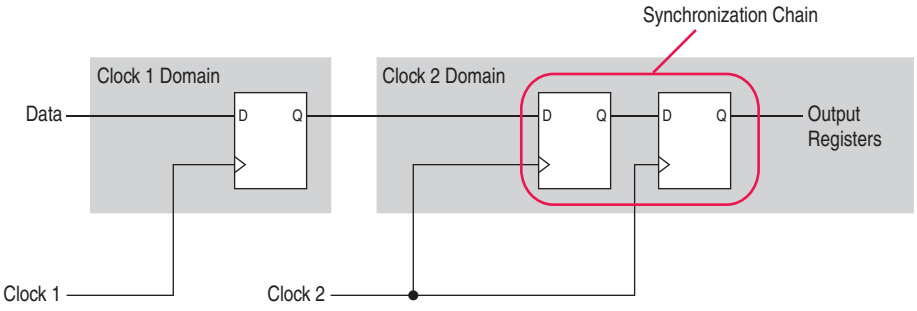
A synchronization register chain, or synchronizer, is defined as a sequence of registers that meets the following requirements:

- The registers in the chain are all clocked by the same clock or phase-related clocks.

- The first register in the chain is driven asynchronously or from an unrelated clock domain.
- Each register fans out to only one register, except the last register in the chain.

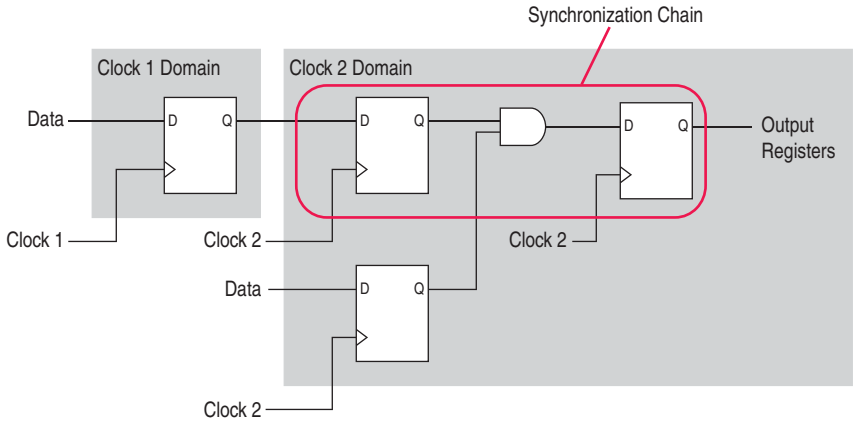
The length of the synchronization register chain is the number of registers in the synchronizing clock domain that meet the above requirements. Figure 7–1 shows a sample two-register synchronization chain.

Figure 7–1. Sample Synchronization Register Chain



The path between synchronization registers can contain combinational logic as long as all registers of the synchronization register 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 Identification and Metastability Analysis" on page 7–4.

The timing slack available in the register-to-register paths of the synchronizer allows a metastable signal to settle, and is referred to as the available settling time. The available settling time in the MTBF calculation for a synchronizer 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 in enabling metastability MTBF analysis and optimization in the Quartus II software is to identify which registers are part of a synchronization register chain. You can apply synchronizer identification settings globally to automatically list possible synchronizers with the **Synchronizer identification** option on the **Timing Analyzer** page in the **Settings** dialog box.

Synchronization chains are already identified within most Altera intellectual property (IP) cores.

For more information about how to enable metastability MTBF analysis and optimization in the Quartus II software, and more detailed descriptions of the synchronizer identification settings, refer to *Identifying Synchronizers for Metastability Analysis* in Quartus II Help.

How Timing Constraints Affect Synchronizer Identification and Metastability Analysis

The 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, you must ensure that your design is 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 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, apply a `set_input_delay` constraint to the input port; otherwise, the input register might be reported as a synchronization register. Constraining a synchronous input port with a `set_max_delay` constraint for a setup (t_{SU}) requirement does not prevent synchronizer identification because the constraint does not associate the input port with a clock domain.

Instead, use the following command 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. To do so, set **Synchronizer Identification** to **Off** for the first synchronization register in these register chains.

Metastability and MTBF Reporting

The Quartus II software reports the metastability analysis results in the Compilation Report and Timing Analyzer reports as described in "Metastability Reports". 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 7–7.

If you change the **Synchronizer Identification** settings, you can generate new metastability reports by rerunning the 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 7–8.

For more information about how metastability MTBF is calculated, refer to the *Understanding Metastability in FPGAs* white paper.

Metastability Reports

Metastability reports provide summaries of the metastability analysis results. In addition to the MTBF Summary and Synchronizer Summary reports, the Timing Analyzer tool reports additional statistics in a report for each synchronizer chain.

For more information about how to access metastability reports in the Quartus II software, refer to *Viewing Metastability Reports* in Quartus II Help.

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 Identification and Metastability Analysis" on page 7–4.

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** 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 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.

For more information about turning on multicorner timing analysis in the Quartus II software, refer to the *Timing Analyzer page* in Quartus II Help.

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 Report

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 statistics report described in the following section.

Synchronizer Chain Statistics Report in the Timing Analyzer

The 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, including whether the **Method of Synchronizer Identification** was **User Specified** (with the **Forced if Asynchronous** or **Forced** settings for the **Synchronizer Identification** setting), or **Automatic** (with the **Auto** setting). The **Number of Synchronization Registers in Chain** report provides information about the parameters that affect the MTBF calculation, including 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 7–7.

The following information is also included to help you locate the chain is in your design:

- **Source Clock** and **Asynchronous Source** node of the signal.
- **Synchronization Clock** in the destination clock domain.
- 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 an approximate rate at which the data changes, specify it with the **Synchronizer Toggle Rate** assignment in the Assignment Editor. You can also apply this assignment to an entity or the entire design. Set the data toggle rate, in number of transitions per second, on the first register of a synchronization chain. The timing analyzer takes the specified rate into account when computing the MTBF of that particular register chain. If a data signal never toggles and does not affect the reliability of the design, you can set the **Synchronizer Toggle Rate** to **0** for the synchronization chain so the MTBF is not reported. 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 which is described in the next section.

In addition, the Fitter optimizes identified synchronizers for improved MTBF by placing and routing the registers to increase their output setup slack values. Adding slack in the synchronizer chain increases the available settling time for a potentially metastable signal, which improves the chance that the signal resolves 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.

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 Reports" on page 7–5, and 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 Timing Analyzer

To enable the Quartus II metastability features, make sure that the 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, MTBF cannot be calculated. If the clock domain constraints are set up incorrectly, the signal transfers between circuitry in unrelated or asynchronous clock domains might be identified incorrectly.

Use industry-standard system-centric I/O timing constraints instead of using FPGA-centric timing constraints. As described in "How Timing Constraints Affect Synchronizer Identification and Metastability Analysis" on page 7–4, 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 **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 7–7.

Optimize Metastability During Fitting

Ensure that the **Optimize Design for Metastability** setting described in "MTBF Optimization" on page 7–8 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 7–8. 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 the Fitter's effort during your design compilation, setting the Fitter effort to **Standard Fit** might improve the design's MTBF results. In the **Settings** dialog box, on the **Fitter Settings** page, set **Fitter effort** to **Standard Fit**.

Increase the Number of Stages Used in Synchronizers, If Possible

Designers commonly use two registers in a synchronization chain to minimize the occurrence of metastable events, and a standard of three registers provides better metastability protection. However, synchronization chains with two or even three registers 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. However, adding a synchronization stage introduces 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 three or more synchronization stages. You can increase the number of stages to more than three using the **How many sync stages?** setting.

Select a Faster Speed Grade Device, if Possible

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 ↵
```

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* and *About Quartus II Scripting* in Quartus II Help.

Identifying Synchronizers for Metastability Analysis

To apply the global Synchronizer Identification assignment described in "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 7–7, use the following command:

```
set_instance_assignment -name SYNCHRONIZER_TOGGLE_RATE <toggle rate in transitions/second> -to <register name>
```

report_metastability and Tcl Command

If you use a command-line or scripting flow, you can generate the metastability analysis reports described in "Metastability Reports" on page 7–5 outside of the Quartus II and user interfaces. Table 7–1 describes the options for the `report_metastability` 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 7–8 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 7–8, 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 to make your design more robust with respect to metastability.

Document Revision History

Table 7–2 shows the revision history for this chapter..

Table 7–2. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Technical edit.
November 2009	9.1.0	■ Clarified description of synchronizer identification settings. ■ Minor changes to text and figures throughout document.
March 2009	9.0.0	■ Initial release.

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

Take an online survey to provide feedback about this handbook chapter.

8. Best Practices for Incremental Compilation Partitions and Floorplan Assignments

QII51017-10.0.0

This chapter 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 when they are recommended to support the flow.

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 an average of 75% for large design changes
- Preserves performance for unchanged design blocks
- Provides repeatable results and reduces the number of compilations
- Enables team-based design flows

For more information about the incremental compilation feature 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*. For feature support, refer to *About Incremental Compilation* in Quartus II Help.

This document contains the following sections:

- "Overview: Incremental Compilation" on page 8–2
- "Incremental Design Flows" on page 8–3
- "Why Plan Partitions and Floorplan Assignments for Incremental Compilation?" on page 8–5
- "General Partitioning Guidelines" on page 8–7
- "Design Partition Guidelines" on page 8–10
- "Design Partition Guidelines for Third-Party IP Delivery" on page 8–25
- "Checking Partition Quality" on page 8–31
- "Including SDC Constraints from Lower-Level Partitions for Third-Party IP Delivery" on page 8–36
- "Introduction to Design Floorplans" on page 8–40
- "Design Floorplan Placement Guidelines" on page 8–43
- "Checking Floorplan Quality" on page 8–48
- "Recommended Design Flows and Application Examples" on page 8–49

Overview: Incremental Compilation

Quartus II incremental compilation is an optional compilation flow that enhances the default Quartus II compilation. If you do not partition 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.

To prepare your design for incremental compilation, you first determine which logical hierarchy boundaries should be treated as separate partitions in your design, and ensure your design hierarchy and source code is set up to support this partitioning. Then create design partition assignments in the Quartus II software to specify which hierarchy blocks are compiled independently as partitions (including empty partitions for missing or incomplete logic blocks).

During compilation, Quartus II Analysis & Synthesis and the Fitter create separate netlists for each partition. These netlists are internal post-synthesis and post-fit database representations of your design.

In subsequent compilations, you can select which netlist type to preserve for each partition. You can either reuse the synthesis or fitting netlist, or instruct the Quartus II software to resynthesize the source files. You can also use compilation results exported from another Quartus II project.

When you make changes to your design, the Quartus II software recompiles only the required partitions and merges the new compilation results with existing netlists for other partitions, according to degree of results preservation you set with the netlist for each design partition.

In some cases, as described in "Introduction to Design Floorplans" on page 8–40, Altera recommends that you create a design floorplan with placement assignments to constrain parts of the design to specific regions of the device.

For step-by-step information on using incremental compilation to recompile only changed parts of your design, refer to *Using the Incremental Compilation Design Flow* in Quartus II Help.

Recommendations for the Netlist Type

For subsequent compilations, you must specify which post-compilation netlist you want to use by specifying a **Netlist Type** setting for each partition.

Use the following general guidelines to set the Netlist Type for partitions:

- **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 partitions with the appropriate 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 required to compile the source files with new assignments or constraints that affect synthesis.
- **Post-Synthesis** (default)—Use this setting to re-fit the design (with any new Fitter assignments) but preserve the synthesis results.

- **Post-Fit—**Use this setting to preserve Fitter and performance results.
  - The default setting for post-fit is to use the highest available level of netlist preservation
- **Post-Fit with Fitter Preservation Level set to Placement—**Use the advanced Fitter Preservation Level setting on the **Advanced** tab in the **Design Partition Properties** dialog box to allow more flexibility to find the best routing for all partitions given their placement.

The Quartus II software also includes a Rapid Recompile feature, which allows you to reuse previous compilation results for unchanged logic when you have changed a very small portion of the design. You can set the **Rapid Recompile** option to preserve compatible results to reduce compilation time when you make small changes inside a partition or the full design. If you choose a netlist type that specifies recompilation and the **Rapid Recompile** option is turned on, then the compatible compilation results are preserved and reused. To ensure you compile from new source files with no compilation results reused, you can turn off the **Rapid Recompile** option.

Incremental Design Flows

The Quartus II incremental compilation feature supports various design flows. Your design flow affects how much impact design partitions have on design optimization, and how much design planning may be required to obtain optimal results.

In the standard incremental compilation flow, the top-level design is divided into partitions, which can be compiled and optimized together in one Quartus II project. If another team member or IP provider is developing source code for the top-level design, they can functionally verify their partition independently, and then simply provide the partition's source code to the project lead for integration into the top-level design. If the project lead wants to compile the top-level design when source code is not yet complete for a partition, they can create an empty placeholder for the partition until the code is ready and added to the top-level design.

Compiling all design partitions in a single Quartus II project ensures that all design logic is compiled with a consistent set of assignments and allows the software to perform global placement and routing optimizations. Compiling all design logic together is beneficial for FPGA design flows because all parts of the design must use the same shared set of device resources. Therefore, it is often easier to ensure good quality of results when partitions are developed within a single top-level Quartus II project.

To enable team-based development, you can design and optimize partitions by accessing the top-level project from a shared source control system or creating copies of the top-level Quartus II project framework, and later exporting each partition so that the post-synthesis netlist or post-fitting results can be integrated into the top-level design.

If required for third-party IP delivery, or in cases where designers can not access a shared or copied top-level project framework, you can create and compile a design partition logic in isolation and export a partition that is included in the top-level project. If this type of design flow is necessary, planning and rigorous design guidelines may be required to ensure that designers have a consistent view of project

assignments and resource allocations. Therefore developing partitions in completely separate Quartus II projects can be more challenging than having all source code within one project or developing design partitions within the same top-level project framework. See the following subsection "Project Management in Team-Based Design Flows" on page 8–4 for more information.

You can also combine design flows and use exported partitions only when it is necessary to support your design environment. For example, if the top-level design includes one or more design blocks that will be optimized by remote designers or IP providers, you can integrate those blocks into the reserved partitions in top-level design when the code is complete, but also have other partitions that will be developed within the top-level design.

If any partitions are developed independently, the project lead must ensure that top-level constraints (such as timing constraints, any relevant floorplan or pin assignments, and optimization settings) are consistent with those used by all designers working independently.

Project Management in Team-Based Design Flows

If possible, each team member should work within the same top-level project framework. As mentioned in the start of this section, using the same project framework amongst team members ensures that designers have the settings and constraints needed for their partition, and allows designers to analyze how their design block interacts with other partitions in the top-level design.

In a team-based environment where designers have access to the project through source control software, each designer can use project files as read-only and develop their partition within the source control system. As designers check in their completed designs, other team members can see how their design partitions interact. If designs do not have access to a source control system, the project lead can provide each designer with a snapshot copy of the top-level project framework to use as they develop their partitions. In both cases, each designer exports their completed design as a partition, and then the project lead integrates the partition into the top-level design. The project lead can choose to use just the post-synthesis netlist and re-perform placement and routing, or to use the post-fitting results to preserve the placement and routing results from the other designer's project.

Alternatively, designers can use their own Quartus II project for their independent design block. You might use this design flow if a designer, such as a third-party IP provider, does not have access to the entire top-level project framework. In this case, each designer must create their own project with all the relevant assignments and constraints. As mentioned at the start of this section, this type of design flow requires more planning and rigorous design guidelines. If the project lead plans to incorporate the post-fitting compilation results for the partition, then this design flow requires especially careful planning to avoid resource conflicts.

The project lead also has the option to generate design partition scripts to manage resource and timing budgets in the top-level design when partitions are developed outside the top-level project framework. Scripts make it easier for any designers of independent Quartus II projects to implement instructions from the project lead. The Quartus II design partition scripts feature creates Tcl files and makefiles that an independent designer can run to set up an independent Quartus II project.

For more information on how to generate design partition scripts, refer to *Generating Design Partition Scripts for Project Management* in Quartus II Help.

If designers create Quartus II assignments or timing constraints for their partitions, they must ensure that the constraints are integrated into the top-level design. If partition designers use the same top-level project framework (and design hierarchy), then the constraints or Synopsys Design Constraints File (**.sdc**) can be easily copied or included in the top-level design. If any partition designers use a separate Quartus II project with a different design hierarchy, they must ensure that constraints are applied to the appropriate level of hierarchy in the top-level design, and should design the **.sdc** for easy delivery to the project lead, as described in "Including SDC Constraints from Lower-Level Partitions for Third-Party IP Delivery" on page 8–36.

You cannot use an exported partition 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, exporting partitions does not support this requirement.

For more information about the various types of incremental design flows and example applications, as well as documented restrictions and limitations, 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 design 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 you to be more rigorous about following good design practices than flat compilations.

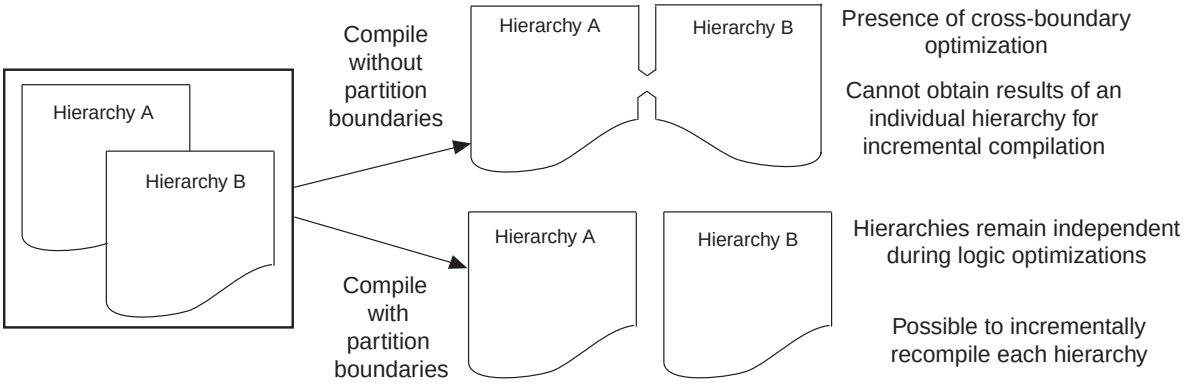
Planning involves setting up the design logic for partitioning, and may also involve planning placement assignments to create a floorplan. Not all design flows require floorplan assignments. If you decide to add floorplan assignments later, when the design is close to completion, well-planned partitions make floorplan creation much easier. Poor partition or floorplan assignments can worsen design area utilization and performance, making timing closure more difficult.

As FPGA devices get larger and more complex, following good design practices becomes more important for all design flows. Adhering to 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 cross-boundary optimizations between partitions, the Quartus II software cannot obtain separate results for each individual partition. The logical hierarchical boundaries between partitions are treated as hard boundaries for logic optimization to allow the software to size and place each partition independently. Figure 8–1 shows the effects of partition boundaries during 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 prevent any cross-boundary optimizations, the Quartus II software synthesizes each partition without using any information about logic 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.

You can use the **Merge** command to combine hierarchical partitions into a single partition, as long as they have the same immediate parent partition. Merging partitions allows for some additional optimizations for partition I/O ports that connect between or feed more than one of the merged hierarchical design blocks.

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. If partitions are fit separately in different projects, 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.

There are possible timing performance utilization effects due to partitioning and creating a floorplan. Not all designs encounter these issues, but you should consider these effects if a flat version of your design is very close to meeting its timing requirements, or is close to using all the device resources, before adding partition or floorplan assignments:

- Partitions can increase resource utilization due to cross-partition optimization limitations, and floorplan assignments can increase resource utilization because regions can lead to unused logic. If your device is very full with the flat version of your design, you can focus on creating partitions and floorplan assignments for timing-critical or often-changing blocks to get the most benefit out of the Quartus II features.
- Partitions and floorplan assignments might increase routing utilization compared to a flat design. If long compilation times are due to routing congestion, you might not be able to use the incremental flow to reduce compilation time. Review the Fitter messages to check how much time is spent during routing optimizations and to see the percentage of routing utilization. When routing is difficult, you can use incremental compilation to lock the 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.
- Partitions can reduce timing performance in some cases because of the optimization and resource effects described above, causing longer logic delays. Floorplan assignments restrict logic placement, which can make it more difficult for the Fitter to meet timing requirements. Use the guidelines in this document to reduce any effect on your design performance.

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, consider limiting the number of partitions to prevent degradation in the quality of results.

Creating good design partitions and good floorplan location assignments helps to improve the design resource utilization and timing performance results for cross-partition paths. Guidelines for creating these assignments are discussed in the following sections.

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 or merge hierarchical blocks into the same 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 8–8
- "Partition Design by Functionality and Block Size" on page 8–9
- "Partition Design by Clock Domain and Timing Criticality" on page 8–9
- "Consider What Is Changing" on page 8–9

Plan Design Hierarchy and Source Design Files

Start by planning 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 can also use the **Merge** command to combine hierarchical partitions into a single partition, as long as they have the same immediate parent 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 is more difficult to group them to form reasonably-sized partitions.

Create each entity in an independent file. The software uses a file checksum to detect changes, and automatically recompiles a partition if its source file changes and their netlist type is set to either post-synthesis or post-fit. If the design entities for two partitions are defined in the same file, changes to the logic in one partition initiates 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 entity 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 information about what changes initiate an automatic recompilation of a partition, 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, designate the top-level entity from each netlist as 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 information about incremental synthesis in third-party tools, refer to the documentation provided by your tool vendor, 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 is often 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 approximately 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. Additionally, 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, you should consider what is changing in the design. Is there 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 one time and immediately preserve the results, so that you will not have to compile that part of the design again. Is 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 remains unchanged

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.

Design Partition Guidelines

Follow the partitioning guidelines presented in this section when you create or modify the HDL code for each design block that you might want to assign as a design partition. You do not have to follow all the recommendations exactly to achieve a good quality of results with the incremental compilation flow, but adhering to as many as possible maximizes your chances for success.

This section includes the following topics:

- "Register Partition Inputs and Outputs" on page 8–10
- "Minimize Cross-Partition-Boundary I/O" on page 8–11
- "Avoid the Need for Logic Optimization Across Partitions" on page 8–12
- "Consider a Cascaded Reset Structure" on page 8–24

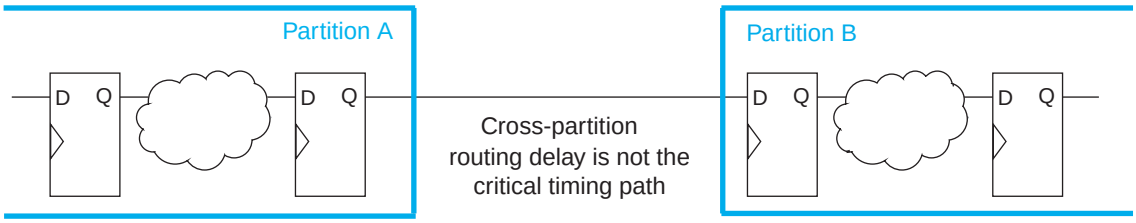
This section includes examples of the types of optimizations that are prevented by partition boundaries, and describes how you can structure or modify your partitions to avoid these limitations.

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 8–2, 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 flows in which partitions are placed independently in separate Quartus II projects. Additionally, the partition boundary does not affect combinational logic optimization because each register-to-register logic path is contained within a single partition.

Figure 8–2. 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 a 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 "Partition Statistics Report", described on page 8–34 lists how many I/Os are registered or unregistered. The "Incremental Compilation Advisor" described on page 8–48 lists the unregistered ports for each partition.

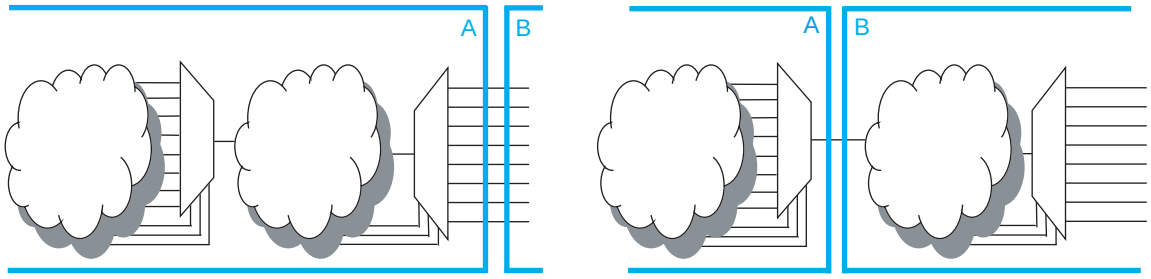
Minimize Cross-Partition-Boundary I/O

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 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 or merge the partitions to avoid these inter-partition paths.

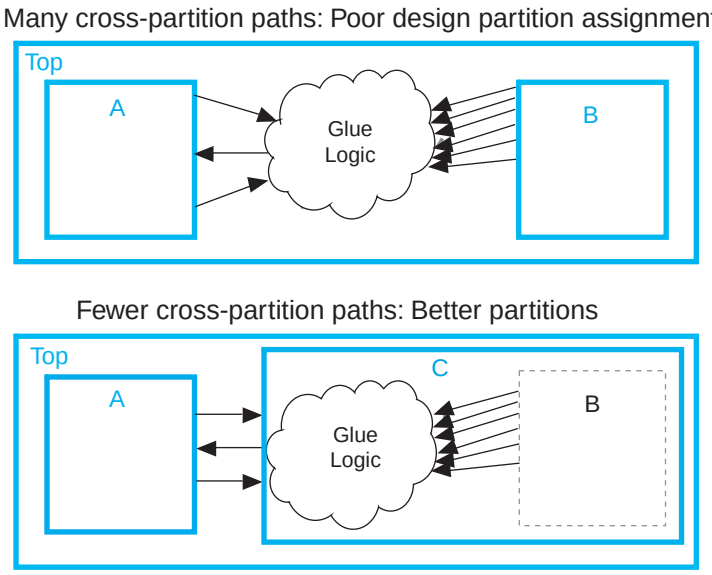
When dividing your design into partitions, consider the types of functions at the partition boundaries. Figure 8–3 shows an expansive function with more outputs than inputs on the left side, which makes a poor partition boundary, and, on the right side, a better place to assign the partition boundary that minimizes cross-partition I/Os. Adding registers to one or both sides of the cross-partition path in this example would further improve the partition quality.

Figure 8–3. 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 8–4 includes a new level of hierarchy C 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 8–4. Minimizing I/O between Partitions by Modifying Glue Logic



The "Partition Statistics Report", described on page 8–34 lists the number of I/O ports, as well as the number of inter-partition connections, for each partition.The "Incremental Compilation Advisor" described on page 8–48 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 8–6, 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 complete or comes from another designer, the designer might not have followed these guidelines 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 suggest simple changes to your partition definitions or hierarchy to move the partition boundary and improve your results.

The following guidelines ensure that your design does not require logic optimization across partition boundaries:

- "Keep Logic in the Same Partition for Optimization and Merging" on page 8–13
- "Keep Constants in the Same Partition as Logic" on page 8–14
- "Avoid Unconnected Partition I/O" on page 8–15
- "Avoid Signals That Drive Multiple Partition I/O or Connect I/O Together" on page 8–16

- "Invert Clocks in Destination Partitions" on page 8–17
- "Connect I/O Pin Directly to I/O Register for Packing Across Partition Boundaries" on page 8–17
- "Do Not Use Internal Tri-States" on page 8–21
- "Include All Tri-State and Enable Logic in the Same Partition" on page 8–22
- "Include Bidirectional I/O Registers in the Same Partition (For Older Device Families)" on page 8–23

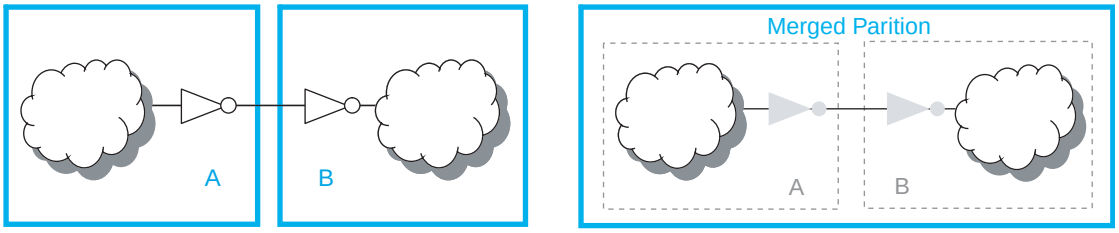
Keep Logic in the Same Partition for Optimization and Merging

If any design logic requires logic optimization or merging to obtain optimal results, ensure that 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 8–5. To maintain correct incremental functionality, these two inverters cannot be removed from the design during optimization because they occur in different design partitions. The Quartus II 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, partitions A and B are merged to group the logic in blocks A and B into one partition. If the two blocks A and B are not under the same immediate parent partition, you can create a wrapper file to define a new level of hierarchy that contains both blocks, and set this new hierarchy block as the partition. 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. Inversion logic is readily available in Altera device architecture; however, it is a simple demonstration of the types of logic optimization that are prevented by partition boundaries.

Figure 8–5. 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 merged partition can be removed: Better partition

In a flat design, the 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 only a concern if merging is required to fit the design in the target device. Therefore, you are more likely to encounter this issue during troubleshooting rather than during planning, if your design uses more logic than is available in the device.

Merging PLLs and Transceivers (GXB)

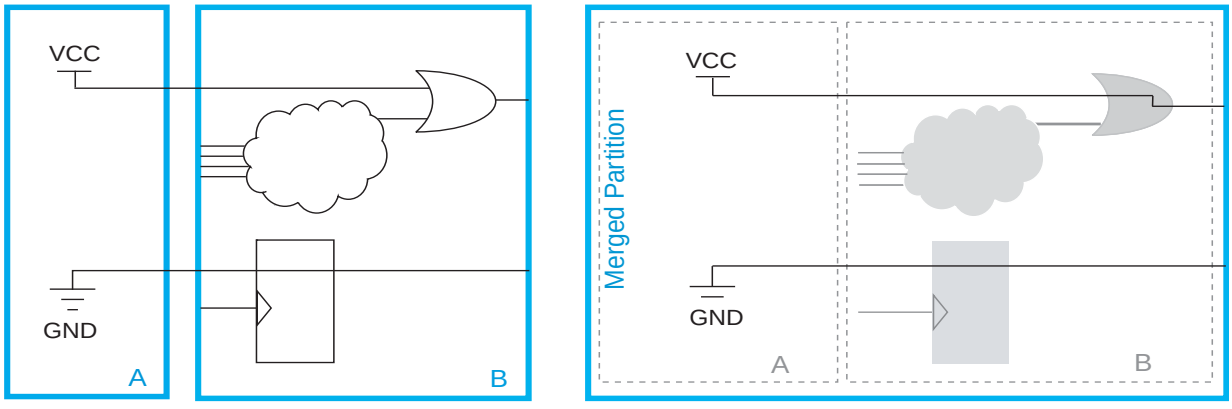
Multiple instances of the ALTPLL megafunction can use the same PLL resource on the device. Similarly, GXB transceiver instances can share high-speed serial interface (HSSI) resources in the same quad as other instances. The Fitter can merge multiple instantiations of these blocks into the same device resource, even if it requires optimization across partitions. 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 Quartus II 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 8–6 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 such as these could appear due to parameter or 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 8–6 shows a merged partition that groups the logic in blocks A and B. If the two blocks A and B are not under the same immediate parent partition, you can create a wrapper file to define a new level of hierarchy that contains both blocks, and set this new hierarchical block as the partition. Within the single merged partition, the Quartus II software can use the constants to optimize and remove much of the logic in block B (shown in gray).

Figure 8–6. Keeping Constants in the Same Partition as the Logic They Feed



Connections to constants in another partition: Poor design partition assignment

Constants in merged partition are used to optimize: Better partition

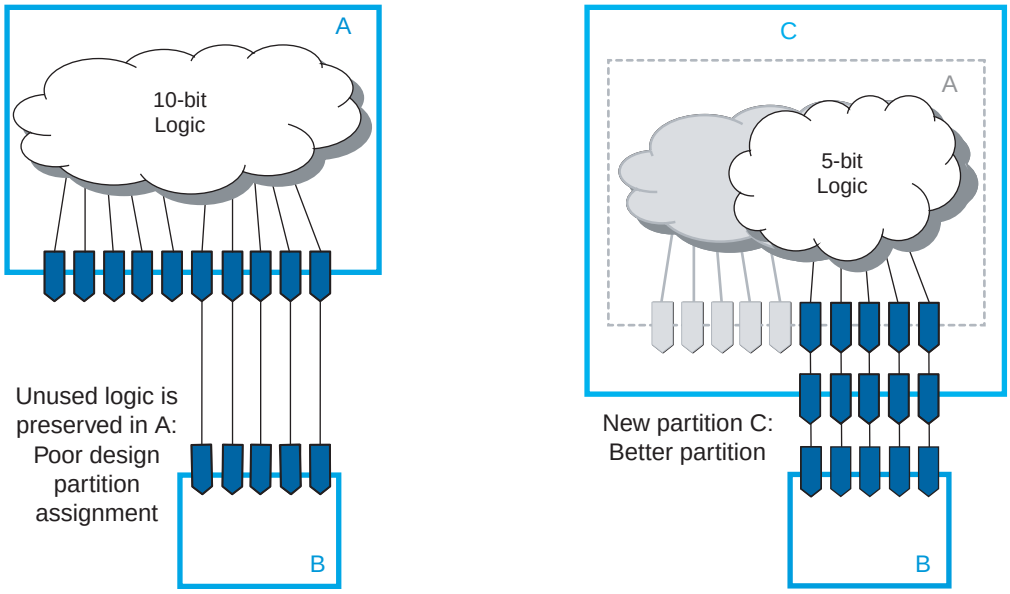
The "Partition Statistics Report" on page 8–34, lists how many input ports are fed by GND or V_{CC}.The "Incremental Compilation Advisor" described on page 8–48 lists the ports.

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 could create cross-partition dependence. For best results, connect ports to an appropriate node or remove them from the partition. If you know that a port will not be used, consider defining a wrapper module with a different port interface.

For example, the left side of Figure 8–7 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 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 can later connect all 10 bits in partition B and use all 10 bits from partition A. In this design example, 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 8–7. Avoiding Unconnected Partition I/O by Creating a Wrapper File



The "Partition Statistics Report" on page 8–34 lists how many I/Os are unconnected. The "Incremental Compilation Advisor" described on page 8–48 lists the unconnected ports.

Avoid Signals That Drive Multiple Partition I/O or Connect I/O Together

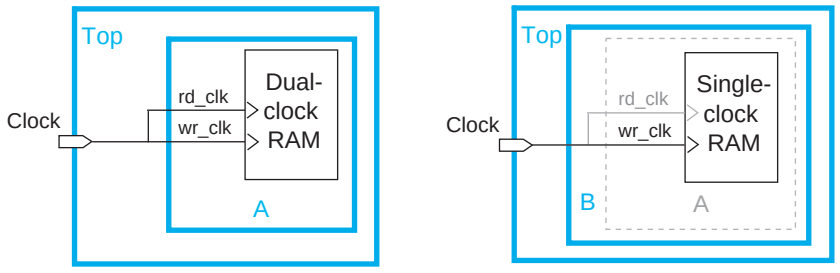
Do not use the same signal to drive multiple ports of a single partition or directly connect two ports of a partition.

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-level 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 partition.

Figure 8–8 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-level partition), redefine the port interface so that 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. Additionally, 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 8–8. Preventing One Signal from Driving Multiple Partition Inputs



Two clocks cannot be treated as the same signal: Poor design partition assignment

With Partition B, RAM can be optimized for one clock: Better partition

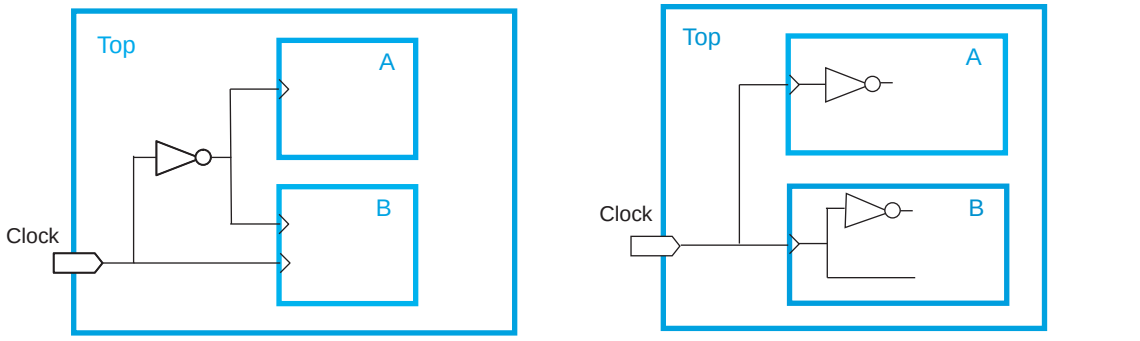
The "Incremental Compilation Advisor" described on page 8–48 lists partition ports that have the same driving signal, and ports that are directly connected together.

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 Quartus II 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 8–9, the Quartus II software uses logic to invert the signal in the partition that defines the inversion (the top-level 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 does not behave like a gated clock.

Figure 8–9. Inverting Clock Signal in Destination Partitions



Inverter acts as clock gating (skew!): Poor design partition assignment

Clock inverted inside destination LABs, only one global routing signal: Better partition

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 Pin Directly to I/O Register for Packing Across Partition Boundaries

The software allows cross-partition register packing of I/O registers in certain cases where your input and output pins are defined in the top-level hierarchy (and the top-level partition), but the corresponding I/O registers are defined in other partitions.

Input pin cross-partition register packing requires the following specific circumstances:

- 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.

Output pin cross-partition register packing requires the following specific circumstances:

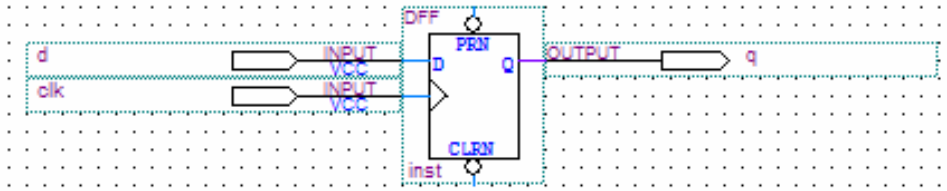
- The register feeds exactly one output pin.
- The output pin is fed by only one signal.
- The path between the register and output pin includes only output ports of partitions that have one fan-out each.

The following examples of I/O register packing illustrate this point using Block Design 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 8–10.

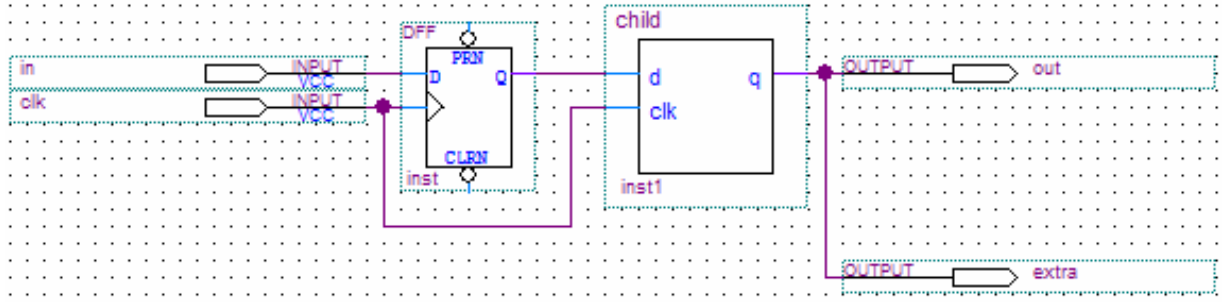
Figure 8–10. Subdesign with One Register, Designated as a Separate Partition



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 8–11.

Figure 8–11. Top-Level Design Instantiating the Subdesign in Figure 8–10 with Two Output Pins



In this case, the Quartus II 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 allowed because it requires modification to the interface of the subdesign partition. To perform incremental compilation, the Quartus II software must preserve the interface of design partitions.

To allow the Quartus II software to pack the register in the subdesign from Figure 8–10 with the output pin `out` in Figure 8–11, 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. Doing so guarantees that the Fitter can optimize the two nodes without violating partition boundaries.
- Duplicate the register in your subdesign HDL as in Figure 8–12 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 8–13. Doing so converts the cross-partition register packing into the simplest case where each register has a single fan-out.

Figure 8–12. Modified Subdesign from Figure 8–10 with Two Output Registers and Two Output Ports

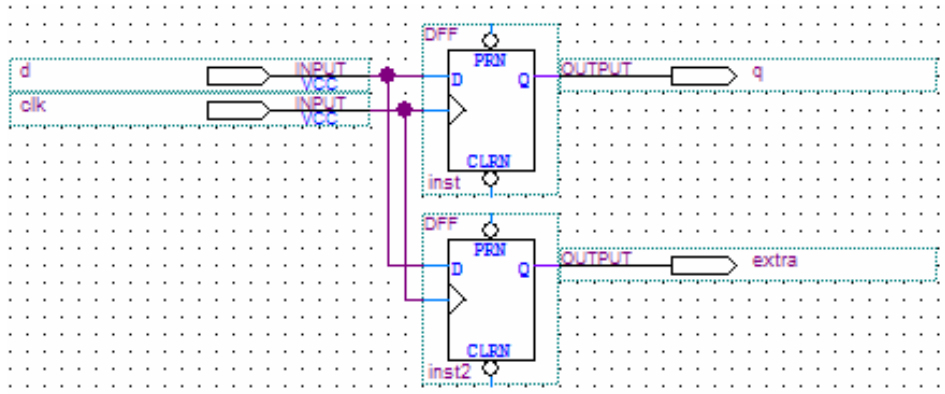
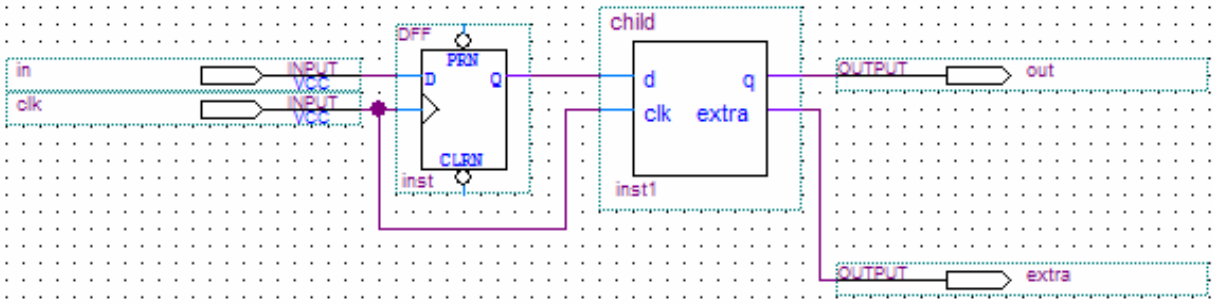


Figure 8–13. Modified Top-Level Design from Figure 8–11 Connecting Two Output Ports to Output Pins



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 8–10. The top-level design in Figure 8–14 instantiates the subdesign as an input register with the input pin inverted. The top-level design in Figure 8–15 instantiates the subdesign as an output register with the signal inverted before feeding an output pin.

Figure 8–14. Top-Level Design Instantiating the Subdesign in Figure 8–10 as an Input Register with an Inverted Input Pin

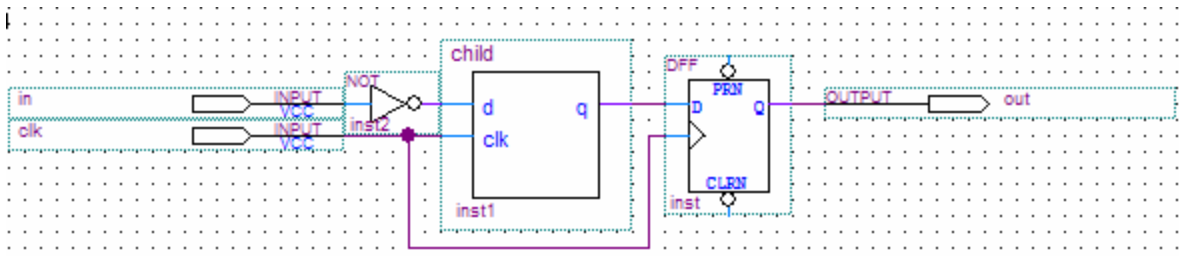
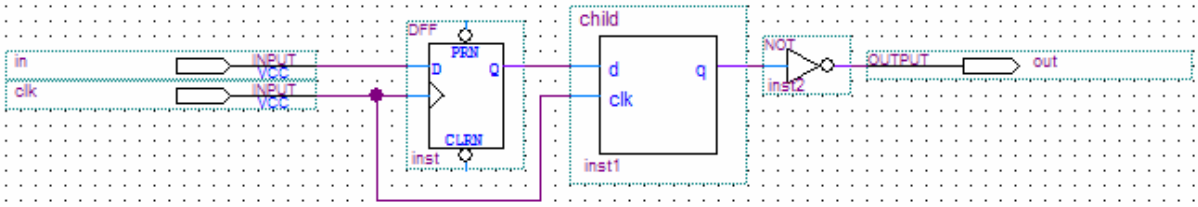


Figure 8–15. Top-Level Design Instantiating the Subdesign in Figure 8–10 as an Output Register Feeding an Inverted Output Pin



In these cases, the Quartus II software does not perform register packing. If there is a **Fast Input Register** assignment on pin `in`, as in Figure 8–14, or a **Fast Output Register** assignment on pin `out`, as in Figure 8–15, the Quartus II 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 allowed 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 Quartus II software to pack the register in the subdesign from Figure 8–10 with the input pin `in`, as in Figure 8–14 or the output pin `out`, as in Figure 8–15, 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. Doing so ensures that the Fitter can optimize the I/O register and inverter without violating 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. Doing so allows the Fitter to optimize the inverter into the register implementation, so that 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 usually converted to OR gates or multiplexing logic. If tri-state logic occurs on a hierarchical partition boundary, the Quartus II 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 8–16 and Figure 8–17 show 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 Quartus II software to convert the tri-state signals into logic.

Figure 8–16. Unsupported Internal Tri-State Signals

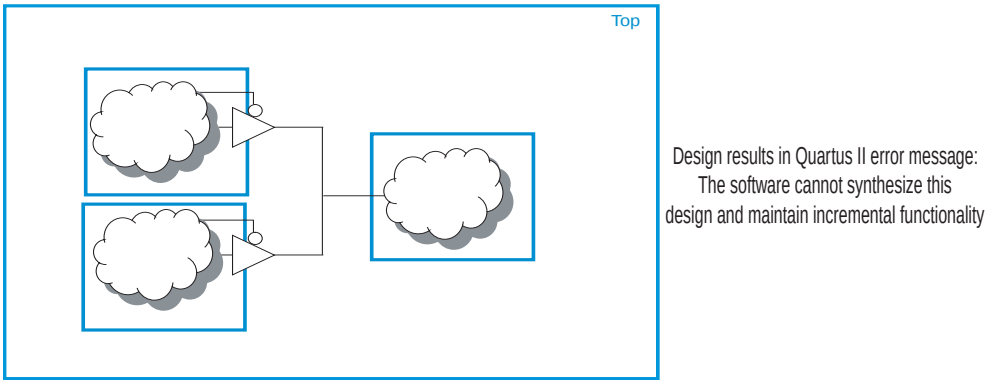
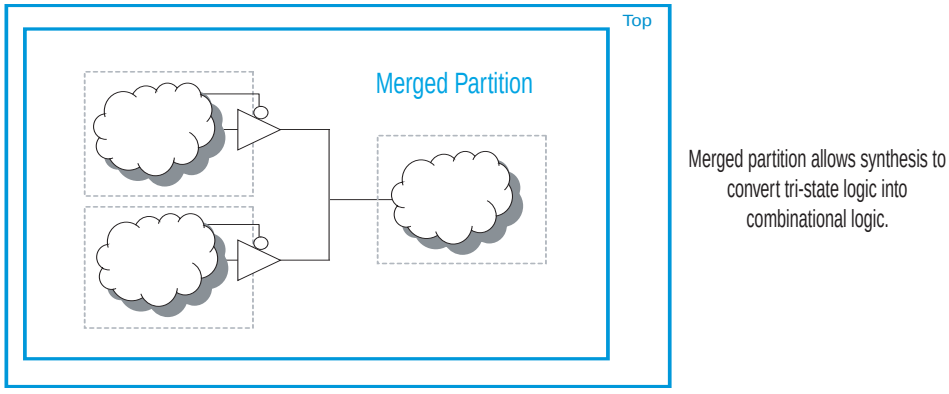


Figure 8–17. Merged Partition Allows Synthesis to Convert Internal Tri-State Logic to Combinational Logic



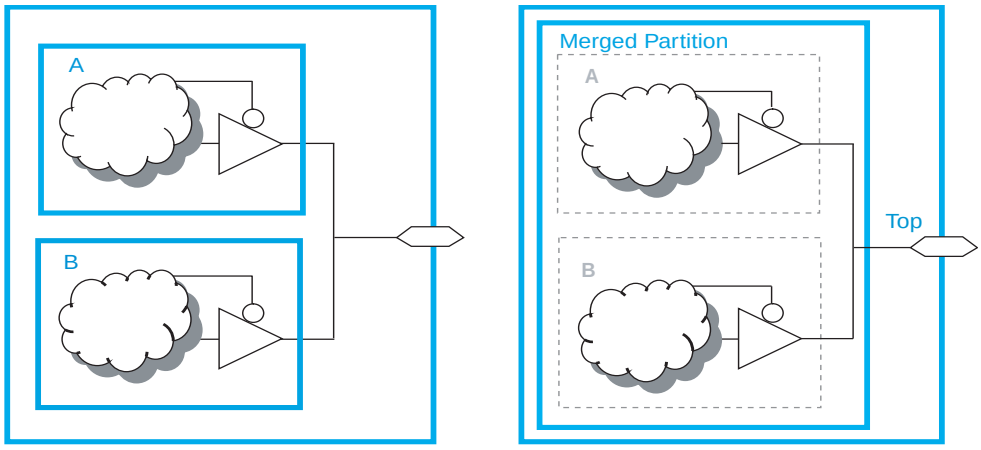
Do not use tri-state signals or bidirectional ports on hierarchical partition boundaries, unless the port is connected directly to a top-level I/O pin on the device. If you must use internal tri-state logic, ensure that all the control and destination logic is contained in the same partition, in which case the Quartus II software can convert the internal tri-state signals into combinational logic like in a flat design. In this example, you can also merge all three partitions into one partition, as shown in Figure 8–17, to allow the Quartus II software to treat the logic as internal tri-state and perform the same type of optimization as 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 Quartus II 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 Quartus II software cannot implement the specified logic and maintain incremental functionality. On the right side, partitions A and B are merged to group the logic from the two blocks. With this single partition, 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



Multiple tri-states on partition boundaries: Illegal partitions

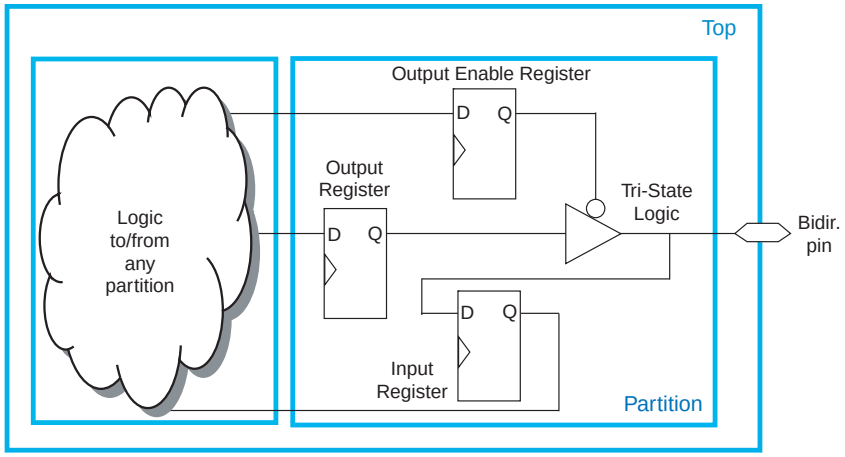
Tri-state output logic within merged partition: Better partition

Include Bidirectional I/O Registers in the Same Partition (For Older Device Families)

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 in the Stratix II, Stratix, Cyclone® II, and Cyclone families (but this restriction does not apply to newer devices). Additionally, 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, all the I/O logic must be defined inside the same partition for the Quartus II software to implement all three registers in the I/O element along with the tri-state logic in the affected devices. 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 (for Older Devices)



Bidirectional logic is within one partition, and I/O logic directly feeds I/O pin

Summary of Guidelines Related to Logic Optimization Across Partitions

To ensure that your design does not require logic optimization across partitions, follow the guidelines in this section:

- 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 (in older device families)

Remember that these guidelines are not mandatory when implementing an incremental compilation flow, but can improve the quality of results. When creating source design code, follow these guidelines and organize your HDL code to support good partition boundaries. For designs that are complete, assess whether assigning a partition affects the resource utilization or timing performance of a design block as compared to the flat design. Make the appropriate changes to your design or hierarchy, or merge partitions as required, to improve your results.

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. In an incremental flow, the global reset signal creates a timing dependency between the top-level partition and lower-level partitions.

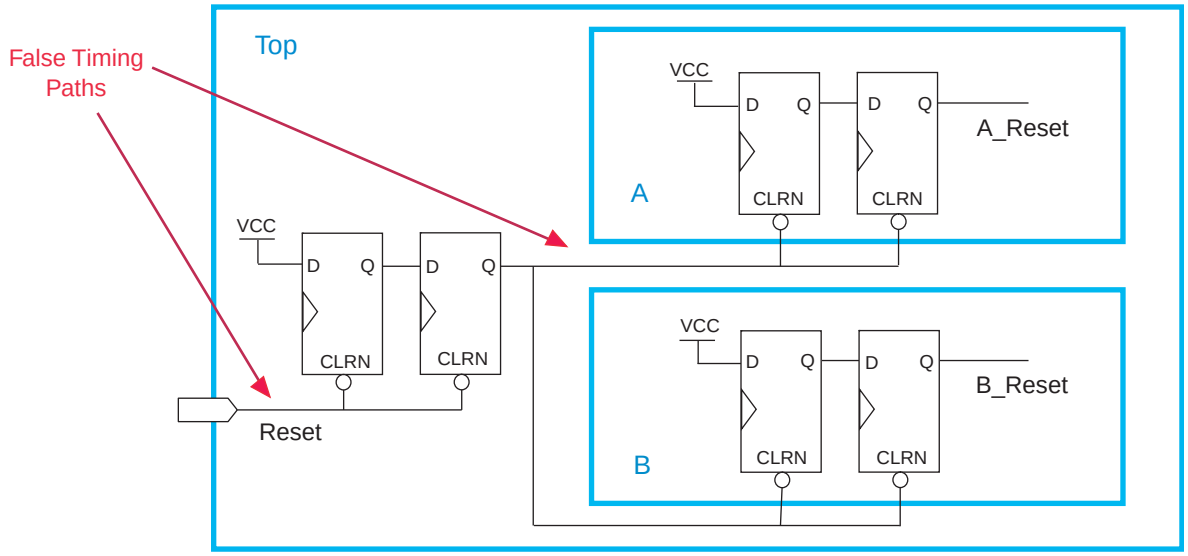
For incremental compilation, it is helpful to minimize the impact of global structures. To isolate each partition, consider adding reset synchronizers. By using cascaded reset structures, the intent is to reduce the inter-partition fan-out of the reset signal, thereby minimizing the effect of the global signal. Reducing the fan-out of the global reset signal also provides more flexibility in routing the cascaded signals, and might help recovery and removal times in some cases.

This recommendation 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. Additionally, 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 on the chip following good synchronous design practices, meaning that 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, the reset paths for each partition are independent. This structure 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 reset signal of the partition 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 design can help you achieve timing closure and partition independence for your global reset signal. Evaluate the circuit and consider how it works for your design.

Design Partition Guidelines for Third-Party IP Delivery

This section includes additional design guidelines that can improve incremental compilation flows where exported partitions are developed independently. These guidelines are not always required but usually recommended with exported partitions compiled in a separate Quartus II project, such as when delivering intellectual property. A unique challenge of IP delivery for FPGAs is the fact that the partitions being developed independently must share a common set of resources. To minimize issues that might arise from sharing a common set of resources, you can design partitions within a single Quartus II project, or a copy of the top-level design. A common project ensures that designers have a consistent view of the top-level design framework, as described in "Project Management in Team-Based Design Flows" on page 8–4.

Alternatively, an IP designer can export just the post-synthesis results to be integrated in the top-level design when the post-fitting results from the IP project are not required. Using a post-synthesis netlist provides more flexibility to the Quartus II Fitter so that less resource allocation is required. If a common project is not possible, especially when the project lead plans to integrate the IP's post-fitting results, it is important that the project lead and IP designer clearly communicate their requirements, as explained in the following sections:

- "Allocate Logic Resources" on page 8–26
- "Allocate Global Routing Signals and Clock Networks if Required" on page 8–27
- "Assign Virtual Pins" on page 8–28
- "Perform Timing Budgeting if Required" on page 8–29
- "Drive Clocks Directly" on page 8–29
- "Recreate PLLs for Lower-Level Partitions if Required" on page 8–30

Allocate Logic Resources

In an incremental compilation design flow in which designers such as third-party IP providers optimize partitions and then export them to a top-level design, the Quartus II software places and routes each partition separately. In some cases, partitions can use conflicting resources when combined at the top level. Allocation of logic resources requires that you to decide on a set of logic resources (including I/O, LAB logic blocks, RAM and DSP blocks) that the IP block will "own". This process can be interactive; the project lead and the IP designer might work together to determine what resources are required for the IP block and are available in the top-level design.

You can constrain logic utilization for the IP core using design floorplan location assignments, as described in "Introduction to Design Floorplans" on page 8–40. The design should use pin assignments to specify I/O pin locations.

You can also specify limits for Quartus II synthesis to allocate and balance resources. This procedure can also help if device resources are overused in the individual partitions during synthesis.

In the standard synthesis flow, the Quartus II software can perform automated resource balancing for DSP blocks or RAM blocks and convert some of the logic into regular logic cells to prevent overuse.

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.

You can, in certain device families, assign a number of LAB, DSP or RAM resources for each partition. Use the following logic options to specify the maximum number of logic blocks that the Quartus II software can use in the specified partition: **Maximum Number of LABs**, **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 in the **More Analysis & Synthesis Settings** dialog box.

For information on how to set global logic options for partitions, refer to *More Analysis & Synthesis Settings Dialog Box* in Quartus II Help.

You can also set an option for a specific partition with the Assignment Editor by selecting the assignment name, applying it to the root entity of a partition, and then setting an integer as the value. Partition-specific assignments override global assignments, if any. However, each partition that does not have a partition-specific assignment can use the number of LAB, DSP, or RAM blocks set by the global assignment. You should note that this behavior can lead to over-allocation of logic blocks, eventually resulting in a no-fit error.

For more information about resource balancing DSP and RAM blocks 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 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*.

Allocate Global Routing Signals and Clock Networks if Required

In most cases, you do not have to allocate global routing signals because the Quartus II software finds the best solution for the global signals. However, if your design is complex and has multiple clocks, especially for a partition developed by a third-party IP designer, you may have to allocate global routing resources between various partitions.

Global routing signals can cause conflicts when independent partitions are integrated into a top-level design. The Quartus II software automatically promotes high fan-out signals to use global routing resources available in the device. Third-party partitions can use the same global routing resources, thus causing conflicts in the top-level design. Additionally, LAB placement depends on whether the inputs to the logic cells within the LAB use a global clock signal. Problems can occur if a design does not use a global signal in a lower-level partition, but does use a global signal in the top-level design.

If the exported IP core is small, you can reduce the chance of problems by using constraints to promote clock and high fan-out signals to regional routing signals that cover only part of the device, instead of global routing signals. In this case, the Quartus II software is likely to find a routing solution in the top-level design because there are many regional routing signals available on most Altera devices, and designs do not typically overuse regional resources.

To ensure that an IP block can utilize a regional clock signal, 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. When 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 device.

If global resources are heavily used in the overall design, or the IP designer requires global clocks for their partition, you can set up constraints to avoid signal overuse at the top-level by assigning the appropriate type of global signals or setting a maximum number of clock signals 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. Alternatively, partition designers can specify the number of clocks allowed in the project using the maximum clocks allowed options under **Fitter Settings** in the **Settings** dialog box. 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 particular type in your design.

If you require more control when planning a design with integrated partitions, you can assign a specific signal to use a particular clock network in Stratix II and newer device families by assigning the clock control block instance called CLKCTRL. To do this, 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 in the **More Fitting Settings** dialog box.

For information on how to disable automatic global promotion, refer to *More Fitter Settings Dialog Box* in Quartus II Help.

If you are using design partition scripts for independent partitions, the Quartus II software can automatically write the commands to pass global constraints and turn off automatic options.

For more information on how to generate design partition scripts, refer to *Generating Design Partition Scripts for Project Management* in Quartus II Help.

Alternatively, to avoid problems when integrating partitions into the top-level design, you can direct the Fitter to discard the placement and routing of the partition netlist by using the Post-Synthesis netlist, which forces the Fitter to reassign all the global signals for the partition when compiling the top-level design.

Assign Virtual Pins

Virtual pins map lower-level design I/Os to internal cells. If you are developing an IP block in an independent Quartus II project, use virtual pins when the number of I/Os on a partition exceeds the device I/O count, and to increase the timing accuracy of cross-partition paths.

You can create a virtual pin assignment in the Assignment Editor for partition I/Os that will become internal nodes in the top-level design. Leave the 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, and 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 design partition scripts. The scripts place the virtual pins to correspond with the placement of the other partitions in the top-level design.

For more information on how to generate design partition scripts, refer to *Generating Design Partition Scripts for Project Management* in Quartus II Help.

Tri-state outputs cannot be assigned as virtual pins because internal tri-state signals are not supported in Altera devices. Connect the signal in the design with regular logic, or allow the software to implement the signal as an external device I/O pin.

Perform Timing Budgeting if Required

If you optimize partitions independently and integrate them to the top-level design, or compile with empty partitions, any unregistered paths that cross between partitions are not optimized as an entire path. In these cases, the software 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 timing requirements. You can reduce this effect by ensuring that input and output ports of the partitions are registered whenever possible. Additionally, using the same top-level project framework helps avoid this problem by providing the software with full information about other design partitions in the top-level design.

To ensure that the software correctly optimizes the input and output logic in any independent partitions, you might 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 software optimizes the paths appropriately.

When performing manual timing budgeting in a partition for I/O ports that become internal partition connections in a top-level design, you can assign location and timing constraints to the virtual pin that represents each connection to further improve the quality of the timing budget. Refer to "Assign Virtual Pins" on page 8–28 for a description of virtual pins.

If you are using design partition scripts, the Quartus II software can write I/O timing budget constraints automatically for virtual pins. For more on how to generate design partition scripts, refer to *Generating Design Partition Scripts for Project Management* in Quartus II Help.

Drive Clocks Directly

When partitions are exported from another Quartus II project, you should 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 with exported partitions because the partitions have no information about gating that takes place at the top-level design 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 8–17.

Direct connections to input clock pins also allows design partition scripts to send constraints from the top-level device pin to lower-level partitions.

Recreate PLLs for Lower-Level Partitions if Required

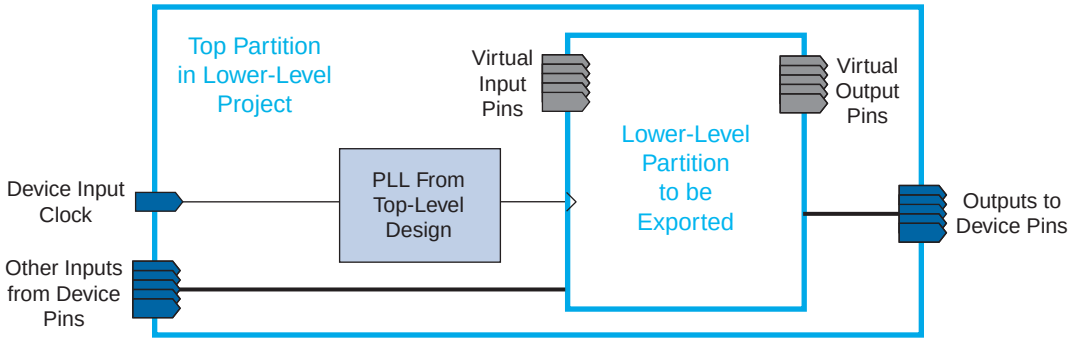
If you use a PLL in your top-level design and connect it to partitions designed in separate Quartus II projects by third-party IP designers, the IP partitions do not have information about the multiplication, phase shift, or compensation delays for the PLL in the top-level design. To accommodate the PLL timing, you can make appropriate timing assignments in the projects created by IP designers to ensure that clocks are not left unconstrained or constrained with an incorrect frequency. Alternatively, you can duplicate the top-level PLL (or other derived clock logic) in the design file for the project created by the IP designer to ensure that you have the correct PLL parameters and clock delays for a complete and accurate timing analysis.

If the project lead creates a copy of the top-level project framework that includes all the settings and constraints needed for the design, this framework should include PLLs and other interface logic if this information is important to optimize partitions.

If you use a separate Quartus II project for an independent design block (such as when a designer or third-party IP provider does not have access to the entire design framework), include a copy of the top-level PLL in the lower-level partition as shown in Figure 8–21.

In either case, the IP partition in the separate Quartus II project should contain just the partition logic that will be exported to the top-level design, while the full project includes more information about the top-level design. When the partition is complete, you can export just the partition without exporting the auxiliary PLL components to the top-level design. When you export a partition, the Quartus II software exports any hierarchy under the specified partition into the Quartus II Exported Partition File (**.qxp**), 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 to create and analyze 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

The Incremental Compilation Advisor checks that your design follows the recommendations for creating design partitions that are presented in this document.

Each recommendation provides an explanation, describes the effect of the recommendation, and provides the action required to make the suggested change.

You can check whether your design follows the recommendations on the **Timing Independent Recommendations** page or the **Timing Dependent Recommendations** page (for the TimeQuest Timing Analyzer). For large designs, these operations can take a few minutes.

After you check your design, a symbol appears next to each recommendation that indicates whether 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 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 your design that could be improved.

For example, if not all 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 advisor provides a list of nodes, you can right-click on a node, and then 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.

For more information about the Incremental Compilation Advisor, refer to *Incremental Compilation Advisor Command* and *Example of Using the Incremental Compilation Advisor to Identify Non-Global Ports That Are Not Registered* in Quartus II Help.

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.

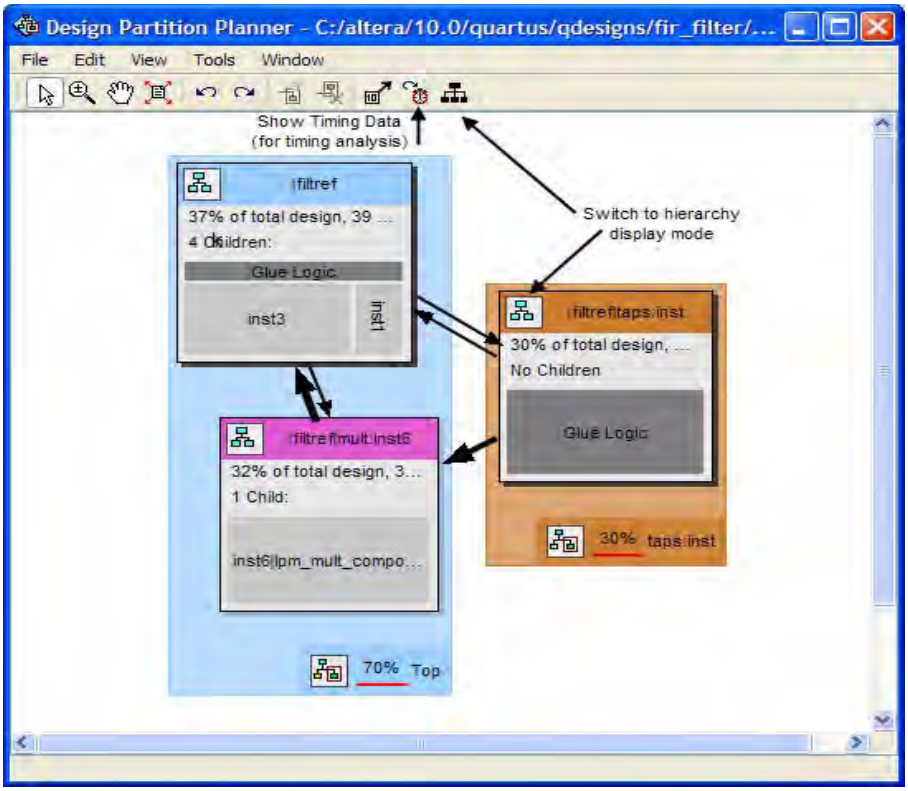
To view a design and create design partitions in the Design Partition Planner, you must first compile the design, or perform Analysis and Synthesis. In the Design Partition Planner, the design appears as a single top-level design block, with lower-level instances displayed as color-specific boxes.

In the Design Partition Planner, you can show connectivity between blocks and extract instances from the top-level design block. 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 the 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. You can right-click on the design block you want to partition (such as the top-level design hierarchy), and the 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, 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 connections between each partition and information about the size of each design instance.

Figure 8–22. Design Partition Planner



You can switch between connectivity display mode and hierarchical display mode, or temporarily to a view-only hierarchy display. You can also remove the connection lines between partitions and I/O banks by turning off **Display connections to I/O banks**, or use the settings on the **Connection Counting** tab to adjust how the connections are counted in the bundles.

To optimize design performance, confine failing paths within individual design partitions so that there are no failing paths passing between partitions, as discussed in earlier sections. In the top-level entity, child entities that contain failing paths are marked by a small red dot in the upper right corner of the entity box.

To view the critical timing paths from a timing analyzer report, first perform a timing analysis on your design, and then in the Design Partition Planner, click **Show Timing Data** on the View menu.

For more information about the Design Partition Planner, refer to *About the Design Partition Planner* and *Using the Design Partition Planner* in Quartus II Help.

Viewing Design Partition Planner and Floorplan Side-by-Side

You can use the Design Partition Planner together with the Chip Planner to analyze natural placement groupings. This information can help you decide whether the design blocks should be grouped together in one partition, or whether they will make good partitions in the next compilation. It can also help determine whether the logic can easily be constrained by a LogicLock region. If logic naturally groups together when compiled without placement constraints, you can probably assign a reasonably sized LogicLock region to constrain the placement for subsequent 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 previous compilation.

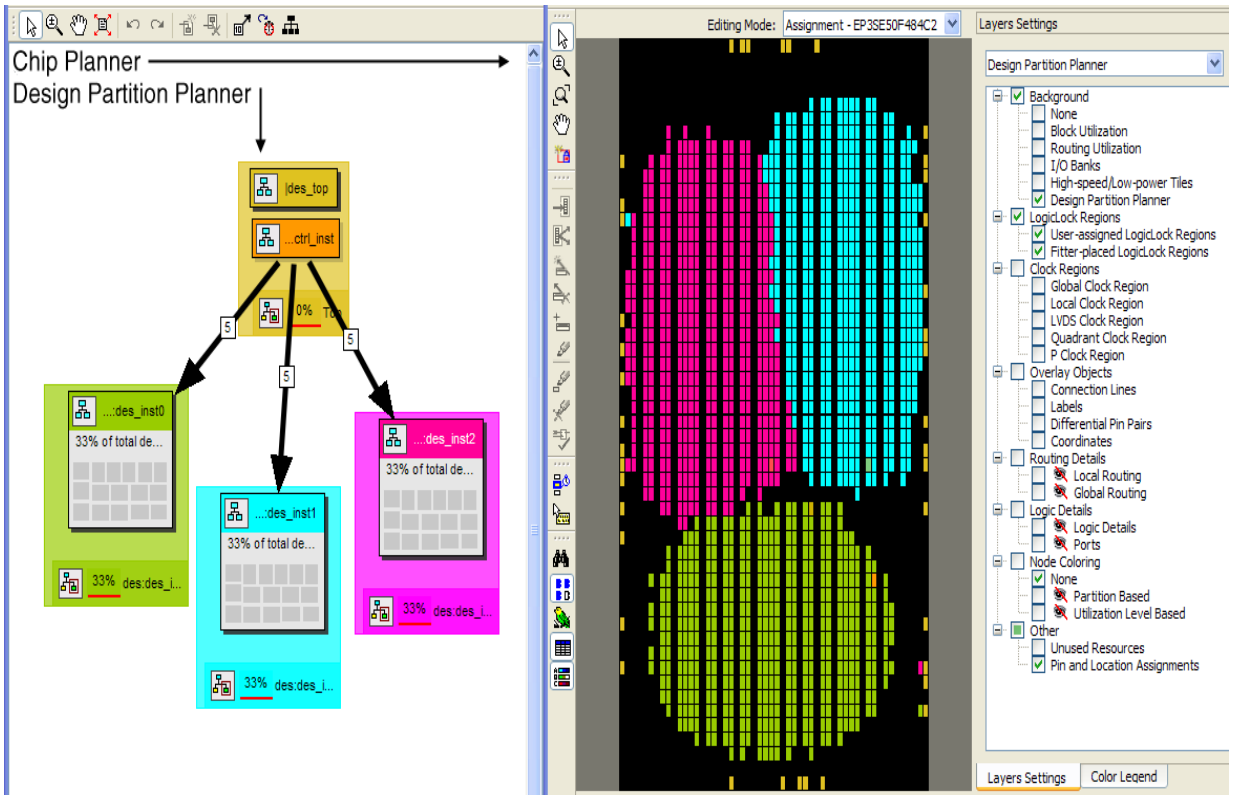
To view the Design Partition Planner and Chip Planner side-by-side, open the Design Partition Planner, and then open the Chip Planner and select the **Design Partition Planner** task. The **Design Partition Planner** task displays the physical locations of design entities with the same colors as in the Design Partition Planner.

In the Design Partition Planner, you can extract instances of interest from their parents by dragging and dropping, or with 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. Use the Design Partition Planner to analyze the design connections. For child instances that are unsuitable to be set as separate design partitions or placed in LogicLock regions, you can return those instances to their parent by dragging and dropping, or with the **Collapse to Parent** command.

Figure 8–23 shows a design displayed in the Design Partition Planner and the Chip Planner with different colors for the top-level design and the three major design instances.

For more information about the Design Partition Planner and Chip Planner, refer to *About the Design Partition Planner* and *About the Chip Planner* in Quartus II Help.

Figure 8–23. Design Partition Planner and Chip Planner



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, or when compiling the completed top-level design in a team-based compilation flow to ensure that 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. The **Show All Partitions** command allows you 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 information represents how many signals are connected within the partition. It then lists the inter-partition connections for each partition, which helps to see how partitions are connected to each other.

For more information about the Partition Merge Reports, refer to *Partition Merge Reports* in Quartus II Help.

Report Partition Timing in the TimeQuest Timing Analyzer

The Report Partitions diagnostic report and the `report_partitions` SDC command in the TimeQuest Timing Analyzer produce a **Partition Timing Overview,** and **Partition Timing Details** table, which lists the partitions and the number of failing paths and the worst case timing slack within each partition.

You can use these reports to analyze where the critical timing paths in the design are with respect to partitions. If a certain partition contains many failing paths, or failing inter-partition paths, you might be able to change your partitioning scheme and improve timing performance.

For more information about the TimeQuest `report_timing` command and reports, see the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Check if Partition Assignments Impact the Quality of Results

You can ensure that you limit negative effect on the quality of results by following an iterative methodology during the partitioning process. In any incremental compilation flow where 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 region assignments.

   After Analysis & Synthesis, and Partition Merge, perform a placement and timing analysis estimate with the **Start Early Timing Estimate** command. To run a full compilation instead, use the **Start Compilation** command.

2. Record the quality of results from the Compilation Report (timing slack or f_{MAX}, area, and any other relevant results).

3. Create design partitions following the guidelines described in this chapter.

4. Perform another Early Timing Estimate or a full compilation.

5. 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 3 through step 5 to change your partition assignments and use a different partitioning scheme.

6. Even if the quality of results is acceptable, you can repeat step 3 through step 5 by further dividing a large partition into several smaller partitions. Doing so improves compilation time in subsequent incremental compilations. You can repeat these steps 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).

You can also remove partition assignments defined in the top-level design at any time during the design flow to compile the design as one flat compilation and get all possible design optimizations to assess the results. Doing so can be useful if you are using partitions to reduce compilation time as you develop various parts of the design, but can run a long compilation near the end of the design cycle to ensure the design meets its timing requirements.

Including SDC Constraints from Lower-Level Partitions for Third-Party IP Delivery

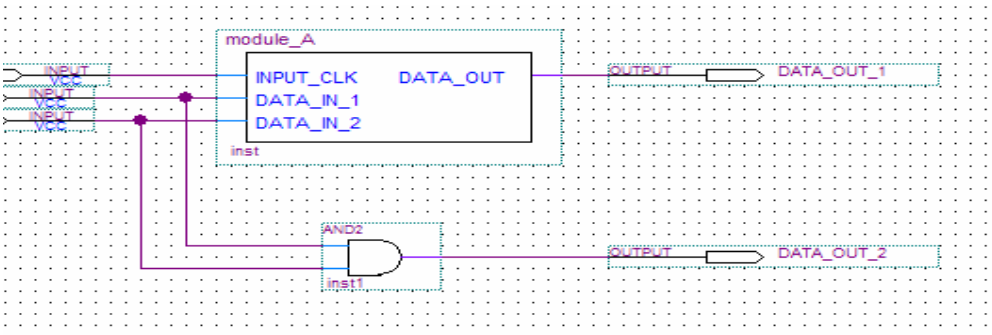
When exported partitions are compiled in a separate Quartus II project, such as when a third-party designer is delivering IP, the project lead must transfer the top-level project framework information and constraints to the partitions, so that each designer has a consistent view of the constraints that apply to the entire design. If the independent partition designers make any changes or add any constraints, they 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. Many assignments from the partition are carried with the partition into the top-level design, however, SDC format constraints for the TimeQuest Timing Analyzer are not copied into the top-level design automatically.

Passing additional timing constraints from a partition to the top-level design must be managed carefully. This section provides recommendations for managing the timing constraints in a third-party IP delivery flow. Note that you can design within a single Quartus II project, or a copy of the top-level design to simplify constraint management.

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

This section uses the example design shown in Figure 8–24 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 an IP designer in a separate Quartus II project.

Figure 8–24. Example Design to Illustrate SDC Constraints



In this top-level design, there is a single clock setting called `clk` associated with the FPGA input called `top_level_clk`. The top-level **.sdc** 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 with Project-Wide Constraints

The **.sdc** with project-wide constraints for the separate Quartus II project should contain all constraints that are not completely localized to the partition. The **.sdc** should be maintained by the 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 separate Quartus II projects. This communication can be challenging when the design is in flux and hierarchies change. The project lead can use design partition scripts to automatically pass some of these constraints to the separate Quartus II projects.

For more information on how to generate design partition scripts, refer to *Generating Design Partition Scripts for Project Management* in Quartus II Help.

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

The **.sdc** should include clock creation and clock constraints for any clock used by more than one partition. These constraints are 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

Additionally, the **.sdc** 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** can also contain any `set_input_delay` or `set_output_delay` constraints that are used for ports in separate Quartus II projects, because these represent delays external to a given partition. If the partition designer wants to set these constraints within the separate Quartus II projects, the team must ensure that the I/O port names are identical in all projects so that the assignments can be integrated successfully without changes.

Similarly, a constraint on a path that crosses a partition boundary should be in the project-wide **.sdc**, because it is not completely localized in a separate Quartus II project.

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

The device input `top_level_clk` in Figure 8–24 drives the `input_clk` port of `module_A`. To make sure the clock constraint is passed correctly to the partition, the project lead creates an **.sdc** 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** as part of the separate Quartus II project.

Creating an .sdc with Partition-Specific Constraints

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

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

The partition-specific **.sdc** is used in the separate Quartus II 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** to simplify these export and integration steps:

- Create a hierarchy variable for the partition (such as `module_A_hierarchy`) and set it to an empty string because the partition is the top-level instance in the separate Quartus II 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 design does not use this empty hierarchy path: `if {![info exists module_A_hierarchy]}`.
- Use the hierarchy variable in the partition-specific **.sdc** 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. In most cases, these assignments should be specified in the **.sdc** with project-wide constraints because the partition interface depends on the top-level design. If you want to set I/O constraints within the partition, the team must ensure that the I/O port names are identical in all projects so that the assignments can be integrated successfully without changes.
- Be careful with the `derive_clocks` and `derive_pll_clocks` commands. In most cases, the **.sdc** with project-wide constraints should call these commands. Because these commands impact the entire design, integrating them unexpectedly into the top-level design might cause problems.

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

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

The partition designer compiles the design with the **.sdc** 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 with the wildcard character (such as " * "). This constraint applies entirely within the partition and must be exported to the top-level design, so it qualifies for inclusion in the **.sdc** 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 in the Top-Level Design

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

To set up the top-level **.sdc** constraint file to accept the **.sdc** files from the separate Quartus II projects, the top-level **.sdc** should define the hierarchy variables specified in the partition **.sdc** files. List the variable for each partition and set it to the hierarchy path, up to and including the instantiation of the partition in the top-level design, 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**.

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

With these commands, the 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 design, and ensures that the wildcard does not affect any nodes outside the partition that it was intended to target.

By following the guidelines in this section, constraint propagation between the separate Quartus II projects can be managed effectively.

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.

In the Quartus II software, LogicLock regions can be used to constrain blocks of a design to a particular region of the device. LogicLock regions represent an area on the device with a user-defined or Fitter-defined size and location in 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 particular 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. Actually, logic from a partition can be placed anywhere in the device if it is not constrained to a LogicLock region, although the Fitter can pack related logic together to improve timing performance. A logical design partition does not refer to any physical area on 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. Altera recommends creating a design floorplan by assigning design partitions to LogicLock regions to improve the quality of results and avoid placement conflicts in some situations for incremental compilation. For more information, refer to "Why Create a Floorplan?" on page 8–41.

Another misconception is that LogicLock assignments are used to preserve placement results for incremental compilation. Actually, LogicLock regions only *constrain* logic to a physical region on 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 a previous compilation.

Why Create a Floorplan?

Creating a design floorplan is usually required if you want to preserve placement for partitions that will be exported, to avoid resource conflicts between partitions in the top-level design. 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 integrating partitions exported from another Quartus II project.
- To ensure a good quality of results when recompiling individual timing-critical partitions.

Location assignments for each partition ensure that there are no placement conflicts between partitions. If there are no LogicLock region assignments, or if LogicLock regions are set to auto-size or floating location, no device resources are specifically allocated for the logic associated with the region. If you do not clearly define resource allocation, logic placement can conflict when you integrate the partitions in the top-level design if you reuse the placement information from the exported netlist.

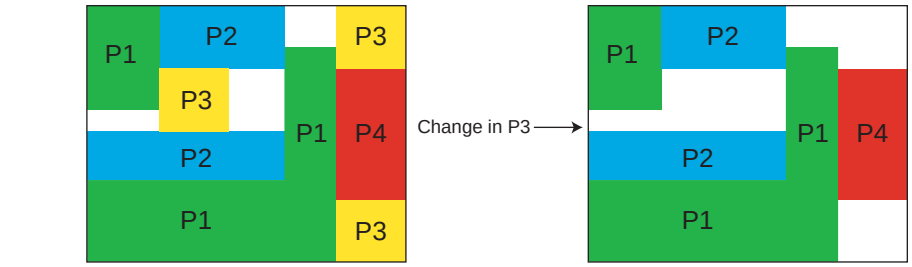
Creating a floorplan is also recommended for timing-critical partitions that have little timing margin to maintain good quality of results when the design changes.

Floorplan assignments are not required for non-critical partitions compiled in the same Quartus II project. The logic for partitions that are not timing-critical 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 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 that may be 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 in 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 been used.

Figure 8–25. Representation of Device Floorplan without Location Assignments



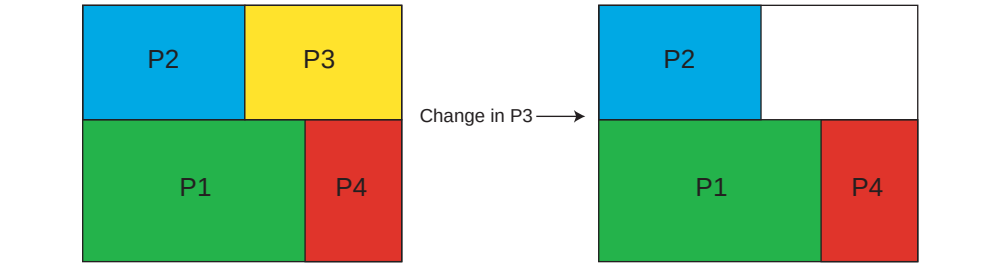
No floorplan assignments: Device has 4 partitions and the logic is placed throughout

Device after removing changed partition P3: New P3 must be placed in empty areas

The Fitter must work harder because of 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. Additionally, 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 is 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



Altera recommends that you create a LogicLock floorplan assignment for timing-critical blocks with little timing margin 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 partition follows partitioning guidelines. You can create floorplan assignments at different stages of the design flow, early or late in the flow. These guidelines help ensure better results as you begin 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 design to allocate each partition a portion of the device resources. Doing so allows the designer for each block to create the logic for their design partition without conflicting with other logic. Each partition can be optimized in a separate Quartus II project if required, and the design can still be easily integrated in the top-level design. Even within one Quartus II project, each partition can be locked down with a post-fit netlist, and you can be sure there is space in the device floorplan for other partitions.

When you have compiled your complete design, or after you have integrated the first versions of partitions developed in separate Quartus II projects, 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 is created, when the code is close to complete and the design structure is likely to remain stable. Creating a late floorplan is typically necessary only if you are starting to use incremental compilation late in the design flow, or need to reserve space for a logic block that becomes timing-critical but still has HDL changes to be integrated. When the design is complete, you can take advantage of the Quartus II analysis features to check the floorplan quality. To adjust 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.

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 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?" on page 8–41. 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 time to find good region placement is required to match the performance of a full flat compilation.

Use the following general procedure 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 might have to iterate through these steps several times to find the best combination of design partitions and LogicLock regions that meet the resource and timing goals of the design.

For more information 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

Before compiling a design with new LogicLock assignments, ensure that the partition **Netlist Type** is set to **Post-Synthesis** or **Source File**, so that the Fitter does not reuse previous placement results.

In most cases, you should assign each LogicLock region to contain logic from only one partition. This organization helps to prevent resource conflicts when partitions are exported and can lead to better performance preservation when locking down parts of a design in a single project.

The Quartus II software is flexible and allows exceptions to this rule. For example, you can place more than one partition in the same LogicLock region if the partitions are tightly connected, but you do not want to merge the partitions into one larger partition. For best results, ensure that you recompile all partitions in the LogicLock region every time the logic in one partition changes. Additionally, if a partition contains multiple lower-level entities, you can place those entities in different areas of the device with multiple LogicLock regions, even if they are defined in the same partition.

You can use the **Reserved** LogicLock option to ensure that you avoid conflicts with other logic which is not locked into a 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.

For more information about LogicLock region properties, refer to the *LogicLock Region Properties Dialog Box* in Quartus II Help.

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 locations or regions chosen by the Fitter as a guideline. If you have compiled the full design, you can view the location the partition logic in the Chip Planner. Refer to "Checking Partition Quality" on page 8–31 for information about viewing placement results for each partition in the device floorplan. You can use the natural grouping of each unconstrained partition as a starting point for a LogicLock region constraint. View the placement for each partition that requires a floorplan constraint, and create a new LogicLock region by drawing a box around the area on the floorplan.

For step-by-step information on how to create a LogicLock region, refer to *Creating and Manipulating LogicLock Regions* in Quartus II Help.

Instead of creating regions based on the previous compilation results, you can start with the Fitter results for a default auto size and floating origin location for each new region when the design logic is complete. 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, if the design logic is complete with auto sized or floating location regions, you can specify the size based on the synthesis results and use the locations chosen by the Fitter with the **Set to Estimated Size** command. 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, save the Fitter size and origin location of the Fitter with the **Set Size and Origin to Previous Fitter Results** command.

For more information on LogicLock region, refer to *Creating and Manipulating LogicLock Regions* in Quartus II Help.

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 device. On average, regions with fixed size and location yield better timing performance than auto-sized regions.

Modifying Region Size and Origin

After saving the Fitter 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 when the design has not yet been created, 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. Make sure that 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 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 subsequent 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. 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 integrating results into a top-level design. In a single project, 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 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

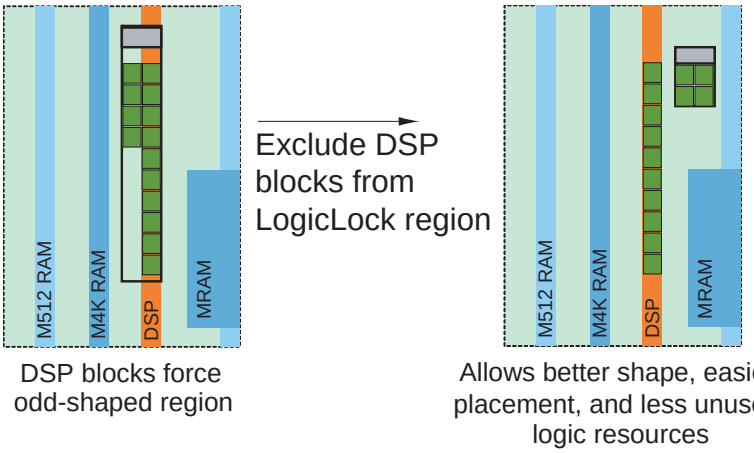
You can exclude certain resource types from a LogicLock region to manage the ratio of logic to dedicated DSP and RAM resources in the region.

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 **Reserved** property of the region 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, due to 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 **LogicLock Regions Properties**. In the **LogicLock Region Properties** dialog box, highlight the design element (module or entity) in the **Members** box, and then click **Edit**. To set up a resource exception, click the **Edit** 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 or non-contiguous areas of the device, you can connect multiple rectangular regions together using the **Merge** command.

For devices that do not support the **Merge** command (Arria ™ GX, Cyclone, Cyclone II, HardCopy, HardCopy II, MAX ™ II, Stratix, Stratix II, Stratix II GX, and Stratix GX devices), you can limit entity placement to a sub-area of a LogicLock region to create non-rectangular constraints. In these devices, 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 the **Reserved** option for the region prevents the Fitter from placing nodes that are not assigned to the region inside the boundary of the region.

For more information and examples of creating non-rectangular regions, refer to *Creating and Manipulating LogicLock Regions* in 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. You can use 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. For more information, refer to "Incremental Compilation Advisor" on page 8–31.

LogicLock Region Resource Estimates

You can view resource estimates for a LogicLock region to determine the region's resource coverage, and use this estimate before compilation to check region size. Using this estimate helps ensure adequate resources when you are sizing or moving regions.

For information about how to view the properties of a LogicLock region, refer to *LogicLock Region Properties dialog box* in Quartus II Help.

LogicLock Region Properties Statistics Report

LogicLock region statistics are similar to design partition properties, but also include resource usage details after compilation.

The statistics report the number of resources used and the total resources covered by the region, and also lists 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).

For information about the **Statistics** tab in the **LogicLock Region Properties** dialog box, refer to *LogicLock Region Properties dialog box* in Quartus II Help.

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 and perform a report timing operation (for example, report timing for all paths with less than 0 ns slack).

For information about how to locate paths between the TimeQuest Timing Analyzer and the Chip Planner, refer to the *Locate Dialog Box* in Quartus II Help.

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 to visualize how many connections there are between each LogicLock region to improve floorplan assignments or to change partition assignments, if required.

For information about how to display bundles of connection between LogicLock regions, refer to the *Generate Inter-region Bundles Dialog Box* in Quartus II Help.

Routing Utilization

The Chip Planner includes a feature 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, red LAB blocks indicate higher routing congestion. You can move your mouse pointer over a LAB to see a tool tip that reports the logic and routing utilization information.

For information about how to how to view a color map of routing congestion in the Chip Planner, refer to the *About the Chip Planner* in Quartus II Help.

Ensure Floorplan Assignments Do Not Significantly 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 only minor degradation in f_{MAX} after the design is partitioned and floorplan location assignments are created. There is some performance cost associated with setting up a design for incremental compilation; approximately 3% is typical.
- 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 with 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 provides a step-by-step description of the commands required to implement the flow.

Create a Floorplan for Major Design Blocks

Use this incremental compilation flow for designs where you want to assign a floorplan location for each major block in your design. 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 floorplan for major design blocks, follow this general methodology:

1. In the Design Partitions window, 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 set as a partition by default).
3. Run a full compilation of your design to view the initial Fitter-chosen placement of the LogicLock regions as a guideline.
4. In the Chip Planner, you can view the placement results of each partition and LogicLock on the device.
5. If required, modify the size and location of the regions with option in the Chip Planner. For example, make the regions bigger to fill up the device and allow for future logic changes.

   You can also, if needed, create a new LogicLock region by drawing a box around an area on the floorplan.
6. Run an early timing estimate with the **Start Early Timing Estimate** command to estimate the timing performance of your design with the modified or new LogicLock regions.
7. Repeat steps 5 and 6 until you are satisfied with the quality of results for your design floorplan. Once satisfied with your results, run a full compilation of your design.

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 do not want 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 on the device. To create a region for critical design block, follow these steps:

1. Divide up your design into partitions. Consider the guidelines in "Design Partition Guidelines" on page 8–10 to determine partition boundaries. Ensure that you isolate the timing-critical logic in a separate partition.

2. Define a LogicLock region for the timing-critical 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 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 will be re-placed in the same part of the device, which helps ensure good quality of results.

Create a Floorplan as the Project Lead in a Team-Based Flow

Use this approach when you have several designs that will be implemented in separate Quartus II projects by different designers, or third-party IP designers who want to optimize their designs independently and pass the results the project lead.

As the project lead in this scenario, follow these steps to prepare the top-level design for a successful team-based 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. Create a "skeleton" or framework 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. Set the **Netlist Type** for each partition that will be implemented in a separate Quartus II project and later exported and integrated with the top-level design to **Empty**.
5. Create LogicLock regions for each of the 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. Provide the constraints from the top-level design to partition designers using one of the following procedures:
   a. Create a copy of the top-level Quartus II project framework by checking out the appropriate files from a source control system, using the **Copy Project** command, or creating a project archive. Provide each partition designer with the copy of the project.
   b. Provide the constraints with documentation or scripts. To use design partition scripts to pass constraints and generate separate Quartus II projects, refer to *Generating Design Partition Scripts for Project Management* in Quartus II Help.

Conclusion

Incremental compilation provides a number of benefits, especially to large, complex designs. To take advantage of the feature, it is worth spending 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 when design blocks are developed independently, and are recommended for timing-critical partitions that are expected to change. 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 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.

Revision History

Table 8–1 shows the revision history for this chapter.

Table 8–1. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Removed the explanation of the "bottom-up design flow" where designers work completely independently, and replaced with Altera's recommendations for team-based environments where partitions are developed in the same top-level project framework, plus an explanation of the bottom-up process for including independent partitions from third-party IP designers. ■ Expanded the Merge command explanation to explain how it now accommodates cross-partition boundary optimizations. ■ Restructured Altera recommendations for when to use a floorplan.
October 2009	9.1.0	■ Redefined the bottom-up design flow as team-based and reorganized previous design flow examples to include steps on how to pass top-level design information to lower-level projects. ■ Added "Including SDC Constraints from Lower-Level Partitions for Third-Party IP Delivery" from the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the *Quartus II Handbook*. ■ Reorganized the "Recommended Design Flows and Application Examples" on page 8–49 section. ■ Removed HardCopy APEX and HardCopy Stratix Devices section.
March 2009	9.0.0	■ 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.
November 2008	8.1.0	■ Changed to 8-1/2 x 11 page size. No change to content.
May 2007	8.0.0	■ Initial release.

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

Take an online survey to provide feedback about this handbook chapter.

Section III. Synthesis

As programmable logic devices 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 integrated Analysis and Synthesis module of the Compiler to synthesize your design files and create the project database for future stages of the compilation flow. You can also use other EDA synthesis tools to first synthesize your designs, and then generate EDIF netlist files or Verilog Quartus Mapping Files (**.vqm**) that you can use with the Quartus II software. The Quartus II netlist viewers allow you to visually analyze the design netlist at different stages of synthesis and compilation. This section explains the options that are available for each of these flows and how they are supported in the Quartus II software.

This section includes the following chapters:

- Chapter 9, Quartus II Integrated Synthesis

  This chapter documents the integrated synthesis design flow and language support in the Quartus II software. It explains how you can improve synthesis results with Quartus II synthesis options and optimization techniques, 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.

- Chapter 10, Synopsys Synplify Support

  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 with the Synplify software.

- Chapter 11, Mentor Graphics Precision Synthesis Support

  This chapter documents support for the Mentor Graphics® Precision Synthesis software in the Quartus II software, as well as key design flows, methodologies, and techniques for achieving good results in Altera® devices with the Precision Synthesis software.

- Chapter 12, Mentor Graphics LeonardoSpectrum Support

  This chapter documents key design methodologies and techniques for Altera devices using the Mentor Graphics LeonardoSpectrum™ software and Quartus II design flow. The LeonardoSpectrum software is a mature synthesis tool supporting legacy devices and many current devices. Altera recommends using the advanced Precision Synthesis software for new designs in new device families.

- Chapter 13, Analyzing Designs with Quartus II Netlist Viewers

  This chapter shows how to use the Quartus II netlist viewers to analyze your design at various stages of the design cycle. It also provides an introduction to the Quartus II design flow using netlist viewers, an overview of each viewer, and an explanation of the user interface.

9. Quartus II Integrated Synthesis

QII51008-10.0.0

This chapter documents the design flow and features of the Quartus II software. Scripting techniques for applying all the options and settings described are also provided. 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 contains the following sections:

- "Design Flow" on page 9–1
- "Language Support" on page 9–3
- "Incremental Compilation" on page 9–20
- "Quartus II Synthesis Options" on page 9–22
- "Analyzing Synthesis Results" on page 9–70
- "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

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 & Synthesis stage of the compilation flow runs integrated synthesis, which fully supports Verilog HDL, VHDL, and Altera-specific languages, and major features of the SystemVerilog language. For more information, refer to "Language Support" on page 9–3.

In the synthesis stage of the compilation flow, the Quartus II software performs logic synthesis to optimize design logic and performs technology mapping to implement the design logic in device resources such as logic elements (LEs) or adaptive logic modules (ALMs), and other dedicated logic blocks. The synthesis stage also generates a 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 & Synthesis to perform any of the following compilation processes:

- **Analyze Current File**—Parses the current design source file to check for syntax errors. This command does not report on many semantic errors that require further design synthesis. To perform this analysis, on the Processing menu, click **Analyze Current File**.
- **Analysis & Elaboration**—Checks a design for syntax and semantic errors and perform elaboration to identify the design hierarchy. To perform Analysis & Elaboration, on the Processing menu, point to **Start,** and then click **Start Analysis & Elaboration**.
- **Analysis & Synthesis**—Performs complete Analysis & Synthesis on a design, including technology mapping. To perform Analysis & Synthesis, on the Processing menu, point to **Start,** and then 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 with 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.
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.
5. Add timing constraints to specify the timing requirements.

   If you want to partition your design to reduce compilation time, refer to "Incremental Compilation" on page 9–20.

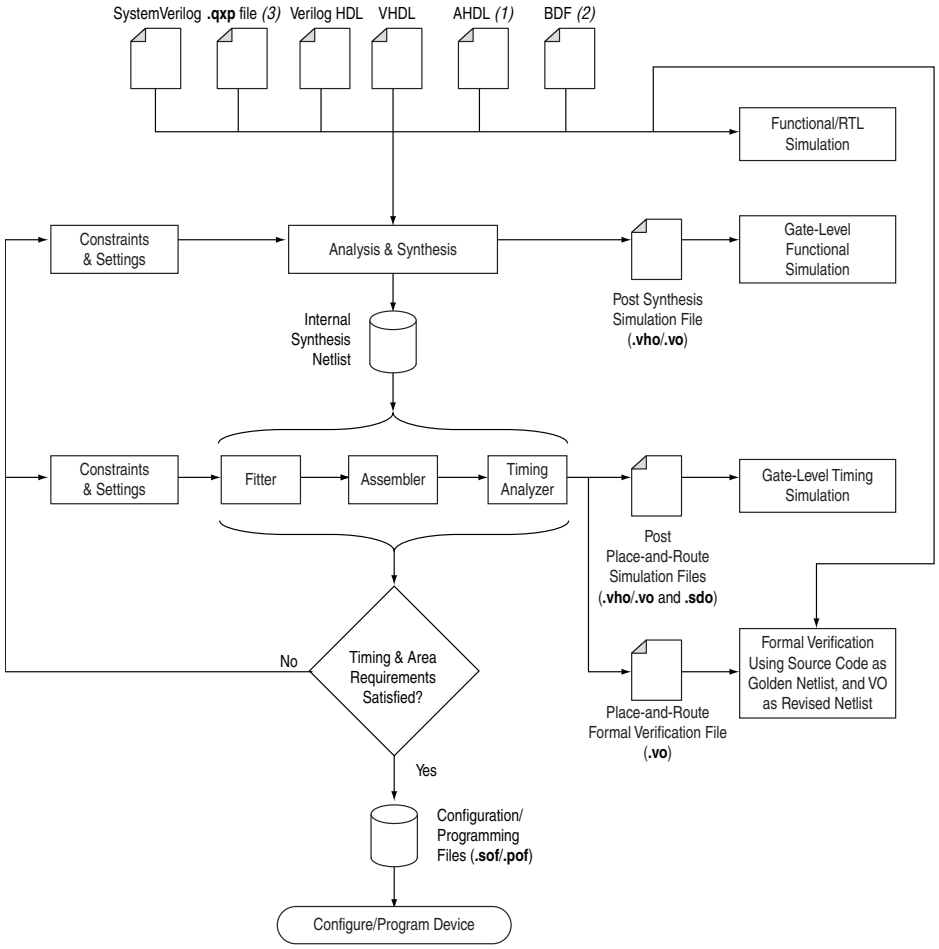
6. Compile the design. To synthesize the design, on the Processing menu, point to **Start**, and then click **Start Analysis & Synthesis**. To run a complete compilation flow including placement, routing, creation of a programming file, and timing analysis, click **Start Compilation** on the Processing menu.
7. After obtaining synthesis and placement and routing results that meet your requirements, program or configure your Altera device.

The Quartus II software produces netlists that allow you to perform functional simulation or gate-level timing simulation, timing analysis, and formal verification.

For more information about Quartus II projects and the compilation flow, refer to *Managing Files in a Project* and *About Compilation Flows* in Quartus II Help. For an overall summary of features in the Quartus II software, 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 stands for the Altera Hardware Description Language.

(2) BDF stands for the Altera schematic Block Design File format (**.bdf**).

(3) The Quartus II Exported Partition File (**.qxp**) is a precompiled netlist that can be used as a design source file. For more information about using **.qxp** as a design source file, refer to "Quartus II Exported Partition File as Source" on page 9–21.

Language Support

This section explains Quartus II integrated synthesis support for HDL, schematic design entry, graphical state machine entry, and 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 custom libraries, and "Using Parameters/Generics" on page 9–16 describes how to use parameters or generics and pass them between 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 & 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 more information about Verilog HDL, refer to *About Verilog HDL* in Quartus II Help.

The Verilog HDL code samples provided in this document follow the Verilog-2001 standard unless otherwise specified. 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**.

For more information about Quartus II Verilog HDL support, refer to *Quartus II Verilog HDL Support* in Quartus II Help.

You can specify a default Verilog HDL version for all files by performing 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 and 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**.
2. On the **Files** page, select the appropriate file in the list, and then click the **Properties** button.
3. In the **HDL Version** list, select **SystemVerilog_2005**, **Verilog_2001**, or **Verilog_1995**, and then click **OK**.

You can also control the Verilog HDL version used to compile the design in a design file with 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*> uses 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 setting 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.

For more information about Verilog HDL synthesis attributes and directives, refer to *Verilog HDL Synthesis Attributes and Directives* in Quartus II Help.

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 9–82.

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. When searching for a relative path, the Quartus II software initially searches relative to the project directory. If the Quartus II 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)

  Quartus II integrated synthesis does not support multiple loop variable declaration

- `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`, and `$size`
- Packed array (include multidimensional packed array and packed arrays of packed structures)
- Unpacked array (include single-valued range dimension)
- Implicit port connections with **.name** and **.***
- Immediate assertions in procedural constructs
- System task such as `$fatal`, `$error`, `$warning`, `$info`, and `$display` statements
- Implicit port and argument direction for functions and tasks (Implicit port direction in modules is not supported)
- Declaration of time unit and time precision using **timeunit** and **timeprecision** in a module

Quartus II integrated synthesis also parses, but otherwise ignores the SystemVerilog assertions.

Designs written to comply with the Verilog-2001 standard might not compile successfully using the SystemVerilog setting because the SystemVerilog standard adds several new reserved keywords.

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. For more information, refer to "Power-Up Level" on page 9–41.

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, and then specify the memory word such as `110101` or `abcde` on the next line. Example 9–3 shows a portion of a Memory Initialization File (**.mif**) 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, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, expand **Analysis & Synthesis Settings** and select **Verilog HDL Input**.
3. Under **Verilog HDL macro**, type the macro name in the **Name** box and the value in the **Setting** box.
4. 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 & Synthesis module supports the following VHDL standards:

- VHDL 1987 (IEEE Standard 1076-1987)
- VHDL 1993 (IEEE Standard 1076-1993)
- VHDL 2008 (IEEE Standard 1076-2008)

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, follow these steps:

1. On the Assignments menu, click **Settings**.
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, and 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**.
2. On the **Files** page, select the appropriate file in the list, and then click **Properties**.
3. In the HDL version list, select **VHDL_2008**, **VHDL_1993**, or **VHDL_1987**, and then click **OK**.

You can also specify the VHDL version used to compile the design for each design file by 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>
```

Example 9–8. VHDL 2008—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:

- `VHDL_1987`
- `VHDL_1993`
- `VHDL_2008`

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 9–82.

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. For more information, refer to "Power-Up Level" on page 9–41.

VHDL-2008 Support

The Quartus II software supports the following VHDL 2008 functions:

- Block comments
- Simplified sensitivity lists
- Extensions to `generate`
- Matching case statement
- Matching equality/inequality operators
- Enhanced bit string literals
- Unconstrained elements in arrays

VHDL Standard Libraries and Packages

The Quartus II software includes the standard IEEE libraries and several 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.

Altera recommends that you 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 one VHDL `wait until` statement per process block. Other VHDL wait constructs, such as `wait for`, or `wait on` statements, or processes with multiple `wait` statements, are not supported.

Example 9–9 is a VHDL code example of a supported `wait until` construct.

Example 9–9. 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;
```

AHDL Support

The Quartus II Compiler's Analysis & 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 more information about AHDL, refer to *About AHDL* in 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 & Synthesis module fully supports **.bdf** for schematic design entry.

You can use the Quartus II 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 *About Schematic Design Entry* in 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 then select **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**, click on the source state, and 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, in which 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 *Creating and Editing State Machines with the State Machine Editor* in 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 referenced 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, while 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" on page 9–13
- "Specifying a Destination Library Name in the Quartus II Settings File or Using Tcl" on page 9–13

When the Quartus II Compiler analyzes the file, it stores the analyzed design units in the file's destination library.

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" on page 9–14. 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, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Files**.
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 assignments specify that the Quartus II software analyzes the **my_file.vhd** and stores its contents (design units) in the VHDL library **my_lib**, and then analyzes the Verilog HDL file **my_header_file.h** and stores its contents in a library called **another_lib**. Refer to Example 9–10.

Example 9–10. 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 as 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 in the **Settings** dialog box, any existing QSF 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 9–11 uses the `library` synthesis directive to create a library called **my_lib** that contains the design unit `my_entity`.

Example 9–11. 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**, or using the Tcl interface. Using the `library` directive to change the destination VHDL library in 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 in a design unit.

Mapping a VHDL Instance to an Entity in a Specific Library

The VHDL language provides several 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 9–12.

Example 9–12. 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 9–13.

Example 9–13. 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 9–14 and Example 9–15 show this instantiation method.

Example 9–14. 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 9–15. 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, known as generics in VHDL, are supported in the Quartus II software, and how you can pass these parameters between design languages.

You can enter default parameter values for your design in the **Default Parameters** page under the **Analysis & Synthesis Settings** page in the **Settings** dialog box. Default parameters allow you to add, change, and delete global parameters for the current assignment. 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**. 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**, and 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" on page 9–16.

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, there is no additional procedure necessary to pass parameters from one language to another. However, in some cases you must specify the type of parameter you are passing. In those cases, you must follow certain guidelines to ensure that the parameter value is interpreted correctly. For parameter type rules, refer to "Passing Parameters Between Two Design Languages" on page 9–18.

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 characters 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 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 correctly interpret the parameter value. Table 9–1 provides a list of valid parameter strings and shows how they are interpreted in the Quartus II software. Use the type-encoded format only when necessary to resolve ambiguity.

Table 9–1. Valid Parameter Strings and Interpretations

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"`	Enumeration 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 (...)

In the Quartus II software version 8.1 and later, 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. Specify parameter type with the pull-down list to avoid ambiguity.

If you open a **.bdf** in the Quartus II software version 8.1 and later, 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.

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 for the parameter to 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 language of the higher-level design block (block that is higher in the hierarchy). 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 language of the lower-level entity 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 9–16 shows a VHDL subdesign that is instantiated in a top-level Verilog HDL design in Example 9–17. Example 9–18 shows a Verilog HDL subdesign that is instantiated in a top-level VHDL design in Example 9–19.

Example 9–16. 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 9–17. Verilog HDL Top-Level Design Instantiating and Passing Parameters to VHDL Entity from Example 9–16

```
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 9–18. 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;
```

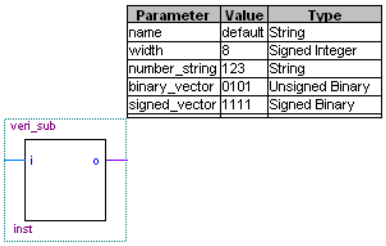
Example 9–19. VHDL Top-Level Design Instantiating and Passing Parameters to the Verilog HDL Module from Example 9–18

```
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 9–18 in a top-level **.bdf** design, you must 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 then 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**, and then click the **Parameters** tab. Right-click on the symbol and choose **Update Design File from Selected Block** to pass the updated parameter to the HDL file.

Figure 9–2. BDF Top-Level Design Instantiating and Passing Parameters to the Verilog HDL Module from Example 9–18.



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 updated partitions of the design 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.

For more information about incremental compilation, refer to *About Incremental Compilation* in Quartus II Help, *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* and *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapters in volume 1 of the *Quartus II Handbook*.

Partitions for Preserving Hierarchical Boundaries

A design partition represents a portion of the design that you want to synthesize and fit incrementally.

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.

The **Preserve Hierarchical Boundary** logic option is available only in Quartus II software versions 8.1 and earlier. Altera recommends using design partitions if you want to preserve hierarchical boundaries through the synthesis and fitting process, because incremental compilation maintains the hierarchical boundaries of design partitions.

Parallel Synthesis

The **Parallel Synthesis** option is one of the Analysis & Synthesis options that you can use to reduce compilation time for synthesis. The feature enables the Quartus II software to use multiple processors to synthesize multiple partitions in parallel.

This feature is available only if the following requirements are met:

- The number of processors allowed in a single machine 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 turned on and your design has two or more partitions.
- In version 9.1 of the Quartus II software, **Parallel Synthesis** runs when **Physical Synthesis** is turned on.
- **Parallel Synthesis** is turned on.

By default, the **Parallel Synthesis** option is turned on. To disable parallel synthesis, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, click **Analysis & Synthesis Settings**, and then click **More Settings** to select **Parallel Synthesis**.

You can also set the **Parallel Synthesis** option with the following Tcl command, as shown in Example 9–20:

Example 9–20. Setting the Parallel Synthesis option with Tcl Command

```
set_global_assignment -name parallel_synthesis off
```

You can view all the interleaved messages from different partitions in the Messages window. The **Partition** column at the Messages window displays the partition ID of the partition referred to in the message. After compilation, you can sort the messages by partition. For more information about displaying the **Partition** column, refer to *About the Messages Window* in Quartus II Help.

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 the Messages page. This option shows the partition ID in parenthesis for each message.

Quartus II Exported Partition File as Source

You can use a Quartus II Exported Partition File (**.qxp**) as a source file in the Quartus II software version 8.1 and later. The **.qxp** is used in incremental compilation, and contains the precompiled design netlist exported as a partition from another Quartus II project, and fully defines the entity. Project team members or IP providers can use a **.qxp** to send their design to the project lead, instead of sending the original HDL source code. The **.qxp** preserves the 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** by applying assignments in the top-level design.

To create a **.qxp**, follow these steps:

1. On the Project menu, click **Export Design Partition**.
2. In the **Export file** box, type the name of the **.qxp**. 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, Post-synthesis netlist,** or both. 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** in the specified directory.

The Quartus II software adds the file into the project and **.qxp** into a specific library. The design entity in the **.qxp** 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 several 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, in which 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.

When you apply a Quartus II Synthesis option globally or to an entity it affects all lower-level entities in the hierarchy path, including entities instantiated with Altera and third-party IP.

The other subsections describe the following common synthesis options in the Quartus II software, and provide HDL examples on how to use each option, where applicable:

- Major Optimization Settings:
  - "Optimization Technique" on page 9–28
  - "Auto Gated Clock Conversion" on page 9–28
  - "PowerPlay Power Optimization" on page 9–31
  - "Restructure Multiplexers" on page 9–34
- Settings Related to Timing Constraints:
  - "Timing-Driven Synthesis" on page 9–30
  - "Optimization Technique" on page 9–28
  - "Auto Gated Clock Conversion" on page 9–28
  - "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 9–38
  - "Safe State Machines" on page 9–39
- Register Power-Up Settings:
  - "Power-Up Level" on page 9–41
  - "Power-Up Don't Care" on page 9–42
- Controlling, Preserving, Removing, and Duplicating Logic and Registers:
  - "Limiting Resource Usage in Partitions" on page 9–32
  - "Remove Duplicate Registers" on page 9–42
  - "Preserve Registers" on page 9–43
  - "Disable Register Merging/Don't Merge Register" on page 9–44
  - "Noprune Synthesis Attribute/Preserve Fan-out Free Register Node" on page 9–44
  - "Keep Combinational Node/Implement as Output of Logic Cell" on page 9–45
  - "Disabling Synthesis Netlist Optimizations with dont_retime Attribute" on page 9–46
  - "Disabling Synthesis Netlist Optimizations with dont_replicate Attribute" on page 9–47
  - "Maximum Fan-Out" on page 9–48
  - "Controlling Clock Enable Signals with Auto Clock Enable Replacement and direct_enable" on page 9–49
  - "Auto Gated Clock Conversion" on page 9–28
  - "Partitions for Preserving Hierarchical Boundaries" on page 9–20
- Megafunction Inference Options:
  - "Inferring Multiplier, DSP, and Memory Functions from HDL Code" on page 9–50
  - "RAM Style and ROM Style—for Inferred Memory" on page 9–53
  - "Turning Off the Add Pass-Through Logic to Inferred RAMs no_rw_check Attribute" 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 9–66
- Setting Quartus II Logic Options in Your HDL Source Code:
  - "Using altera_attribute to Set Quartus II Logic Options" on page 9–67
- Other Settings:
  - "Synthesis Effort" on page 9–35
  - "Synthesis Seed" on page 9–35

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 in your HDL source code.

Analysis & Synthesis Settings Page of the Settings Dialog Box

The **Analysis & Synthesis Settings** page allows you to set global synthesis options that apply to the entire project. You can also use a corresponding Tcl command.

In the Quartus II software version 9.0 and later, some of the advanced synthesis settings are set in the **Physical Synthesis Optimizations** 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*.

Quartus II Logic Options

The 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 to set global assignments. The Quartus II logic options allow you to set instance or node-specific assignments without editing the source HDL code.

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 scripting. 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**. Assignments or settings made with the Quartus II GUI, the **.qsf**, 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 of 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 9–27, in which <*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 in Verilog HDL files are also case-sensitive.

Example 9–21. Specifying Synthesis Attributes in Verilog-1995

```
// synthesis <attribute> [ = <value> ]
or
/* synthesis <attribute> [ = <value> ] */
```

Verilog-1995 comment-embedded attributes, as shown in Example 9–21, should 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 shown in Example 9–22:

Example 9–22. Applying Multiple Attributes to the Same Instance in Verilog-1996

```
//synthesis <attribute1>  [ = <value> ] <attribute2> [ = <value> ]
```

For example, to set the `maxfan` attribute to `16` (for details, refer to "Maximum Fan-Out" on page 9–48) and set the `preserve` attribute (for details, refer to "Preserve Registers" on page 9–43) on a register called `my_reg`, use the following syntax as shown in Example 9–23:

Example 9–23. Setting maxfan and preserve Attribute on a Register

```
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 9–24. Specifying Synthesis Attributes in Verilog-2001 and SystemVerilog

```
(* <attribute> [ = <value> ] *)
```

Verilog-2001 attributes, as shown in Example 9–24, should 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 9–25:

Example 9–25. Applying Multiple Attributes

```
(* <attribute1> [ = <value1>], <attribute2> [ = <value2> ] *)
```

For example, to set the `maxfan` attribute to `16` (refer to "Maximum Fan-Out" on page 9–48 for details) and set the `preserve` attribute (refer to "Preserve Registers" on page 9–43 for details) on a register called `my_reg`, use the following syntax as shown in Example 9–26:

Example 9–26.

```
(* maxfan = 16, preserve *) reg my_reg;
```

VHDL attributes, as shown in Example 9–27, declare the attribute type and then apply it to a specific object.

Example 9–27. Synthesis Attributes in VHDL

```
attribute <attribute> : <attribute type> ;
attribute <attribute> of <object> : <object type> is <value>;
```

Each attribute is defined and applied separately to a given node. 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 shown in Example 9–28:

Example 9–28.

```
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 9–29, Example 9–30, and Example 9–31, in which <*directive*> and <*value*> are variables, and the entry in brackets is optional. Notice that for synthesis directives there is no equal sign before the value; this is different than the Verilog 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 9–29. Specifying Synthesis Directives with Verilog HDL

```
// synthesis <directive> [ <value> ]
or
/* synthesis <directive> [ <value> ] */
```

Example 9–30. Specifying Synthesis Directives with VHDL

```
-- synthesis <directive> [ <value> ]
```

Example 9–31. Specifying Synthesis Directives with VHDL-2008

```
/* synthesis <directive> [<value>] */
```

In addition to the `synthesis` keyword shown above, the `pragma`, `synopsys`, and `exemplar` keywords are supported in 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 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 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 or speed trade-off. Results are design-dependent and vary depending on which device family you use.

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** logic option automatically converts qualified gated clocks (base clocks as defined in the synopsys design constraints [SDC]) to clock enables. To use **Auto Gated Clock Conversion**, you must select the option from the **More Analysis & Synthesis Settings** dialog box, which is found in the **Analysis & Synthesis Settings** page.

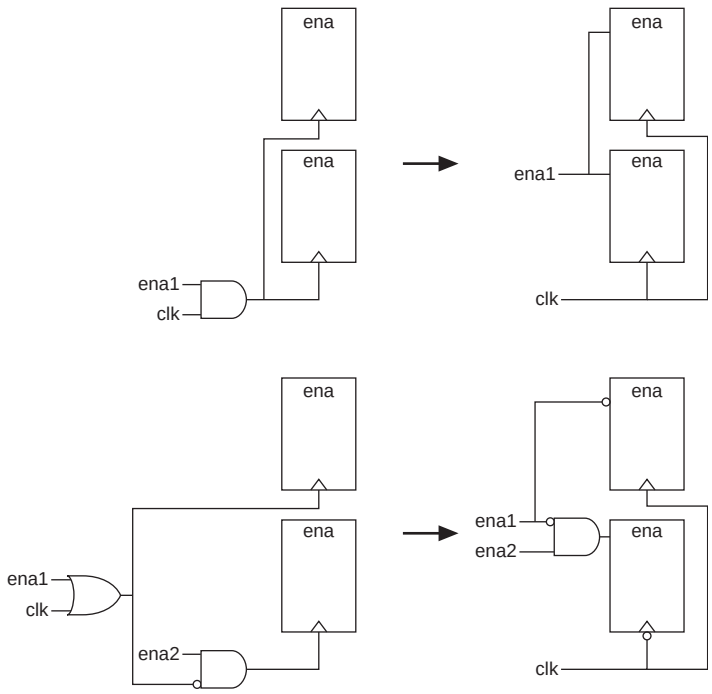
The gated clock conversion occurs when the following conditions are met:

- 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 primitives. Because the gated-clock conversion cannot trace the base clock from the gated clock, 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. Each conversion is based on a gated clock tree instead of every gated clock. Therefore, the entire gated clock tree is not converted if you cannot convert a gated clock from a root gated clock of a multiple cascaded 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.

This feature is available when using the TimeQuest Timing Analyzer.

For more information about **Auto Gated Clock Conversion** logic option and a list of supported devices, refer to *Auto Gated Clock Conversion logic option* in Quartus II Help.

Timing-Driven Synthesis

The **Timing-Driven Synthesis** logic option specifies whether Analysis & Synthesis should use the design's SDC timing constraints to better optimize the circuit. When this option is turned on, Analysis & Synthesis runs timing analysis to obtain timing information about the netlist, and then takes into account the SDC timing constraints to focus on critical portions of the design when optimizing for performance, while optimizing non-critical portions for area. When you turn on this option, Analysis & Synthesis also protects SDC constraints by not merging duplicate registers that have incompatible timing constraints. For more information, refer to "SDC Constraint Protection" on page 9–31.

Turning on the **Timing-Driven Synthesis** option causes Analysis & Synthesis to increase performance by improving logic depth on critical portions of the design, and to improve area on non-critical portions of the design. The increased performance comes at the cost of area, specifically adaptive look-up tables (ALUTs) and registers in the design. Depending on how much of the design is timing critical, overall area can increase or decrease when the **Timing-Driven Synthesis** option is turned on. Runtime and peak memory use increases slightly if you turn on the **Timing-Driven Synthesis** option.

When you turn on the **Timing-Driven Synthesis** option, the **Optimization Technique** logic option has the following effect. With **Optimization Technique Speed**, **Timing-Driven Synthesis** optimizes timing-critical portions of the design for performance at the cost of increasing area (logic and register utilization). With an **Optimization Technique** of **Balanced**, **Timing-Driven Synthesis** also optimizes the timing-critical portions of the design for performance, but it only allows limited area increase. With **Optimization Technique Area**, **Timing-Driven Synthesis** only optimizes the design for area. **Timing-Driven Synthesis** prevents registers with incompatible timing constraints from merging for any **Optimization Technique** setting. If your design contains multiple partitions, you can select **Timing-Driven Synthesis** options individually for every partition. If you use a **.qxp** as a source file, or if your design uses partitions developed in separate Quartus II projects, these partitions are treated as a black box. This means that **Timing-Driven Synthesis** cannot properly compute timing of paths that cross the partition boundaries.

Even with the **Optimization Technique** logic option set to **Speed**, the **Timing-Driven Synthesis** option still considers the resource usage in your design when increasing area to improve timing. For example, the **Timing-Driven Synthesis** option checks if a device has enough registers before deciding to implement the shift registers in logics instead of RAM for better timing performance.

When using incremental compilation, integrated synthesis allows each partition to use up all the registers in a device. You can use the **Maximum Number of LABs** settings to specify the number of LABs that every partition is allowed to use. If your design has only one partition, you can also use the **Maximum Number of LABs** settings to limit the number of resources that can be used by the design. This is useful when you plan to add more logic to your design later.

To change the **Timing-Driven Synthesis** option, follow these steps:

1. On the Assignment menu, click **Settings**.
2. In the **Category** list, select **Analysis & Synthesis Settings**. In the **Analysis & Synthesis Settings** page, select or unselect **Timing-Driven Synthesis**.

The feature is available when using the TimeQuest analyzer. 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.

The feature is turned on by default for Arria II GX, Cyclone III, Cyclone IV, Hardcopy III, Hardcopy IV, Stratix III, and Stratix IV devices.

For more information about **Timing-Driven Synthesis** logic option and a list of supported devices, refer to *Timing-Driven Synthesis logic option* in Quartus II Help.

SDC Constraint Protection

The **SDC Constraint Protection** option specifies whether Analysis & Synthesis should protect registers from merging when they have incompatible timing constraints. For example, two registers that are duplicates of each other but have different multicycle constraints on them are not merged when this option is on. When **Timing-Driven Synthesis** is turned on, registers with incompatible constraints are automatically detected, and there is no need to explicitly turn on **SDC Constraint Protection**. To use the **SDC constraint protection** option, you must turn on the option in the **More Analysis & Synthesis Settings** dialog box, which is found in the **Analysis & Synthesis Settings** page.

This feature supports Arria II GX, Arria GX, Cyclone II, Cyclone III, Cyclone IV, HardCopy series, MAX II, Stratix II, Stratix II GX, Stratix III, and Stratix IV device families.

PowerPlay Power Optimization

This logic option controls the power-driven compilation setting of Analysis & Synthesis and determines how aggressively Analysis & Synthesis optimizes the design for power.

To display the **Analysis & Synthesis Settings** page, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Analysis & Synthesis Settings**.

For more information about the available settings for the PowerPlay power optimization option and a list of supported devices, refer to *PowerPlay power optimization (Fitter Settings Page) logic option* in Quartus II Help. 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 Resource Usage in Partitions

One important step of Analysis & Synthesis is resource balancing. In this step, Quartus II integrated synthesis logic option allows you to specify the maximum number of digital signal processing (DSP) blocks that the DSP block balancer assumes exist in the current device for the current partition. You can also specify the maximum number resources available for different RAM types. This option overrides the usual method of using the maximum number of DSP or RAM blocks that the current device supports. For incremental compilation, each partition has a separate balancing step.

By default, Quartus II integrated synthesis considers the information in the targeted device to find out the number of DSP or RAM blocks available. However, in incremental compilation, each partition considers the information in the device independently and consequently assumes that it has all the DSP or RAM blocks in the device available for use. This can result in over allocation of DSP or RAM blocks in the design, which means that the total number of DSP or RAM blocks used by all the partitions is greater than the number of DSP or RAM blocks available in the device. This can eventually lead to a no-fit error during the fitting process.

To avoid this, 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 entity 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 Analysis & Synthesis gives an error.

The **Maximum DSP Block Usage** assignment is available only for supported device families. For a list of the devices, refer to *Maximum DSP Block Usage logic option* in Quartus II Help.

For more information about using the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

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, always set this assignment on each partition individually.

Manually limiting the DSP blocks usage is also useful for HardCopy ASIC device migration, in which the number of DSP blocks that are implemented in HardCopy ASIC devices is more than the number of DSP blocks that are implemented in its companion Stratix II device.

In the Quartus II software version 8.1 and later, the floorplan aware synthesis feature allows you to use LogicLock regions to define resource allocation for DSP blocks and RAM blocks 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*.

Altera recommends that you always use LogicLock assignments before setting the maximum resource allocation assignments per partition. However, this recommendation might not affect 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.

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 RAM blocks from one RAM type to another to meet Fitter requirements in which the RAM block utilization of each RAM type is within limits of the available blocks for each RAM type. The floorplan aware logic 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**, **Maximum Number of M-RAM/M144K Memory Blocks**, or **Maximum Number of LABs**.

The RAM balancing feature does not support Stratix V devices because Stratix V only has M20K memory blocks.

You can specify the maximum DSP and RAM resource allocation by selecting either the **Maximum DSP Block Usage** or **Maximum Number <***block type***> Memory Blocks** option. To select either the **Maximum DSP Block Usage** or **Maximum Number** <*block type*> **Memory Blocks** option, follow these steps:

1. On the Assignment menu, click **Settings**.
2. Under Category, click **Analysis & Synthesis Settings**.
3. In the **Analysis & Synthesis Settings** dialog box, click **More Settings**.
4. In the **Name** pull-down list, select either the **Maximum DSP Block Usage** or **Maximum Number <***block type***> Memory Blocks** option**.**

HardCopy II devices have a limited number of DSP and RAM blocks and there is no assignment to limit the DSP and 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 DSP and RAM block usage after balancing from the Compilation Report.

Restructure Multiplexers

The **Restructure Multiplexers** logic 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 all Altera devices supported by the Quartus II software except MAX 3000 and MAX 7000 devices.

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 arrays 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, this 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 to an individual node or to an entity that is a design partition. You can also specify 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 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. When the Optimization Technique option is set to **Area** or **Balanced**, Quartus II integrated synthesis restructures all multiplexers. 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 compilation, 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 (Part 1 of 2)

Heading	Description
Multiplexer Inputs	The number of different inputs 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.

Table 9–4. Multiplexer Information in the Multiplexer Restructuring Statistics Report (Part 2 of 2)

Heading	Description
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 multiplexer outputs. This name can help determine the source location of the multiplexer bus in your design.

For more information about **Restructure Multiplexers** logic option, refer to *Restructure Multiplexers logic option* in Quartus II Help. 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

The **Synthesis Effort** option specifies the overall synthesis effort level in the Quartus II software. The level can be either **Fast** or **Auto**.

The **Auto** setting indicates standard synthesis effort. The Quartus II software attempts to optimize your design as much as possible.

When the effort level is set to **Fast**, Quartus II integrated synthesis skips several steps to make synthesis run much faster (at the cost of performance and resource utilization). Use the **Fast** synthesis effort level with the Fitter early timing estimate feature. 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. When you use the **Fast** synthesis effort level as part of a full compilation, Fitter runtime might increase because fast synthesis produces a netlist that is slightly more difficult for the Fitter to route as compared to the netlist from a normal synthesis. When the **Synthesis Effort** option is set to **Fast**, **Timing-Driven Synthesis** automatically turns off.

To set the **Synthesis Effort** option from the Quartus II GUI, on the **Analysis & Synthesis Settings** page, click **More Settings**. Select **Auto** or **Fast** from the pull-down menu in the **Synthesis Effort** option, and then click **OK** to close the **Settings** dialog box.

You can run this flow from the Tasks pane in the Quartus II software. Select and expand **Compile Design**, and then **Analysis & Synthesis**. Double-click **Early Timing Estimate** to start the flow.

For more information about **Synthesis Effort** logic option, refer to *Synthesis Effort logic option* in Quartus II Help.

Synthesis Seed

The **Synthesis Seed** option specifies the seed that Synthesis uses to randomly run synthesis in a slightly different way. The Synthesis Seed value specifies the seed used for the current design. You can use this seed when your design is close to meeting requirements, to get a slightly different result. The seeds that lead to the best result for a design may change if the design changes.

To set the **Synthesis Seed** option from the Quartus II GUI, on the **Analysis & Synthesis Settings** page, click **More Settings**. The default value is set to 1. You can specify a non-negative integer value.

State Machine Processing

The **State Machine Processing** logic option specifies the processing style used to synthesize 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 one-hot style. See the example below for details.
User-Encoded	Encodes the state machine in the manner that you specify.
Sequential	Uses a binary encoding in which the first enumeration literal in the Enumeration Type has encoding `0` and the second `1`.
Gray	Uses an encoding in which the encodings for adjacent enumeration literals differ by 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 MSB of each state is the negation of the LSB of the previous state. An *N*-bit Johnson code can represent at most 2^N 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, as shown in Example 9–32:

Example 9–32. 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 9–33:

Example 9–33.

```
parameter S0 = 4'b1010, S1 = 4'b0101, ...
```

If the software infers the states `S0, S1,...` the encoding `4'b1010, 4'b0101,...` encoding is used. 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.

You can view the state machine encoding from the Compilation Report under the State Machines of the Analysis & Synthesis Report. The State Machine Viewer only displays a graphical representation of the state machines as interpreted from your design. For more information on the State Machine Viewer, refer to the State Machine Viewer section of the *Analyzing Designs with Quartus II Netlist Viewers* chapter in volume 1 of the Quartus II Handbook

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. For more information, refer to "Manually Specifying State Assignments Using the syn_encoding Attribute".

For information about the **Safe State Machine** option, refer to "Safe State Machines" on page 9–39.

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 and the second 1
"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, in which N is the number of enumeration literals in the Enumeration Type.
"compact"	Use an encoding with the fewest bits.
"user"	Encode each state using its value in the Verilog source. By chaining the values of your state constants, you can change the encoding of your state machine.

The syn_encoding attribute must follow the enumeration type definition but precede its use.

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 should 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 should 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 9–34, the enum_encoding attribute specifies an arbitrary user encoding for the enumeration type fruit.

Example 9–34. 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";
```

Example 9–35 shows the encoded enumeration literals:

Example 9–35.

```
apple   = "11"
orange  = "01"
pear    = "10"
mango   = "00"
```

Altera recommends that you 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 and the second 1.
"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, in which *N* is the number of enumeration literals in the enumeration type.

In Example 9–34, 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 9–36.

Example 9–36. 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. The software inserts extra logic to detect an illegal state and forces the transition of the state machine to the reset state. This option is most 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 clock-crossing FIFO, because the state machine must have inputs from another clock domain. This option protects only state machines by forcing them into the reset state. All other registers in the design are not protected this way. You can use this option if your design has asynchronous inputs. However, Altera recommends using a synchronization register chain instead of relying on the safe state machine option.

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 added by Quartus II integrated synthesis 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, on the **Analysis & Synthesis Settings** page, select **More Settings**. In the **Existing option settings** list, select **Safe State Machine**, and turn on this option in the **Setting** list.

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 9–37 through Example 9–39.

Example 9–37. Verilog HDL Code: a Safe State Machine Attribute

```
reg [2:0] my_fsm /* synthesis syn_encoding = "safe" */;
```

Example 9–38. Verilog-2001 and SystemVerilog Code: a Safe State Machine Attribute

```
(* syn_encoding = "safe" *) reg [2:0] my_fsm;
```

Example 9–39. VHDL Code: a Safe State Machine Attribute

```
ATTRIBUTE syn_encoding OF my_fsm : TYPE IS "safe";
```

If you are also specifying an encoding style (refer to "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 in which 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 correctly inferred, 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, 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 allows the design's synthesized behavior to match, as closely as possible, the power-up state of the HDL code during a functional simulation.

The following register declarations all set a power-up level of V_{CC} or a logic value "1", as shown in Example 9–40:

Example 9–40.

```
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.

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 turned on, 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

When 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 register, and the first register fans out to the destinations of the second register. Also, if the deleted register has different logic option assignments, the compiler ignores them.

Altera recommends using this option only if you want to prevent the compiler from removing duplicate registers. Use this option only with the **Off** setting. You can apply this option to an individual register or a design entity that contains registers.

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 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.

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 9–44.

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 9–41 through Example 9–43. 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 9–41. Verilog HDL Code: syn_preserve Attribute

```
reg my_reg /* synthesis syn_preserve = 1 */;
```

Example 9–42. Verilog-2001 Code: syn_preserve Attribute

```
(* syn_preserve = 1 *) reg my_reg;
```

The `= 1` after the `preserve` in Example 9–41 and Example 9–42 is optional, because the assignment uses a default value of `1` when it is specified.

Example 9–43. 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.

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 9–44 through Example 9–46. In these examples, the `my_reg` register is prevented from merging.

Example 9–44. Verilog HDL Code: dont_merge Attribute

```
reg my_reg /* synthesis dont_merge */;
```

Example 9–45. Verilog-2001 and SystemVerilog Code: dont_merge Attribute

```
(* dont_merge *) reg my_reg;
```

Example 9–46. 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 Logic analyzer. Additionally, it can retain registers if you are creating a preliminary version of the design in which the fan-out logic of the register is not specified.

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 9–47 though Example 9–49. 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 & 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 in 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 9–47. Verilog HDL Code: syn_noprune Attribute

```
reg my_reg /* synthesis syn_noprune */;
```

Example 9–48. Verilog-2001 and SystemVerilog Code: noprune Attribute

```
(* noprune *) reg my_reg;
```

Example 9–49. VHDL Code: noprune Attribute

```
signal my_reg : stdlogic;
attribute noprune: boolean;
attribute noprune of my_reg : signal is true;
```

For more information about **Preserve Fan-out Free Register Node** logic option and a list of supported devices, refer to *Preserve Fan-out Free Register logic option* in Quartus II Help.

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 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 use the **Ignore LCELL Buffers** logic option to direct Analysis & Synthesis to ignore logic cell buffers created by the **Implement as Output of Logic Cell** logic option or the `LCELL` primitive. If you apply this logic option to an entity, it affects all lower-level entities in the hierarchy path.

To avoid unintended design optimizations, make sure the **Ignore LCELL Buffers** logic option is not inherited by an entity instantiated with Altera or third-party IP that relies on logic cell buffers for correct behavior. For example, if an IP core uses logic cell buffers to manage high fan-out signals and inherits the **Ignore LCELL Buffers** logic option, the target device may no longer function properly.

You can turn off the **Ignore LCELL Buffers** logic option for a specific entity to override any assignments inherited from higher-level entities in the hierarchy path if logic cell buffers created by the **Implement as Output of Logic Cell** logic option or the `LCELL` primitive are required for correct behavior.

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 9–50 through Example 9–52. 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 9–50. Verilog HDL Code: keep Attribute

```
wire my_wire /* synthesis keep = 1 */;
```

Example 9–51. Verilog-2001 Code: keep Attribute

```
(* keep = 1 *) wire my_wire;
```

Example 9–52. VHDL Code: syn_keep Attribute

```
signal my_wire: bit;
attribute syn_keep: boolean;
attribute syn_keep of my_wire: signal is true;
```

Disabling Synthesis Netlist Optimizations with dont_retime Attribute

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 9–53 through Example 9–55. In these examples, the `my_reg` register is prevented from being retimed.

Example 9–53. Verilog HDL Code: dont_retime Attribute

```
reg my_reg /* synthesis dont_retime */;
```

Example 9–54. Verilog-2001 and SystemVerilog Code: dont_retime Attribute

```
(* dont_retime *) reg my_reg;
```

Example 9–55. 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`.

Disabling Synthesis Netlist Optimizations with dont_replicate Attribute

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 9–56 through Example 9–58. In these examples, the `my_reg` register is prevented from being replicated.

Example 9–56. Verilog HDL Code: dont_replicate Attribute

```
reg my_reg /* synthesis dont_replicate  */;
```

Example 9–57. Verilog-2001 and SystemVerilog Code: dont_replicate Attribute

```
(* dont_replicate *) reg my_reg;
```

Example 9–58. 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`.

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 containing 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 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 and MAX 7000 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 acknowledged 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 other ports, 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 small differences in timing which could produce functional differences in the implementation (in the third condition above in which 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 the maximum fan-out assignment. To instruct the software not to check node destinations for possible problems such as 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 9–59 through Example 9–61. 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 for compatibility with other synthesis tools.

Example 9–59. Verilog HDL Code: syn_maxfan Attribute

```
reg clk_gen /* synthesis syn_maxfan = 50 */;
```

Example 9–60. Verilog-2001 Code: maxfan Attribute

```
(* maxfan = 50 *) reg clk_gen;
```

Example 9–61. 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. 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 9–62 through Example 9–64 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 9–62. Verilog HDL Code: direct_enable attribute

```
wire my_enable /* synthesis direct_enable = 1 */ ;
```

Example 9–63. Verilog-2001 and SystemVerilog Code: syn_direct_enable attribute

```
(* syn_direct_enable *) wire my_enable;
```

Example 9–64. VHDL Code: direct_enable attribute

```
attribute direct_enable: boolean;
attribute direct_enable of my_enable: signal is true;
```

Inferring Multiplier, DSP, and Memory Functions from HDL Code

The Quartus II Compiler automatically recognizes multipliers, multiply-accumulators, multiply-adders, or memory functions described in HDL code and converts the HDL code into respective megafunction, or, may map them directly to device atoms or memory atoms. If the HDL code is converted into 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. To disable inference, turn off this option for the entire project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box, or turn off the option for a specific block with the Assignment Editor. This logic option is turned on for Stratix V devices.

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 has three settings: **Off**, **Auto** and **Always**. **Auto** is the default setting in which Quartus II integrated synthesis decides which shift registers to replace or leave in registers. Putting shift registers in memory saves logic area, but can have a negative effect on f_{max}. Quartus II integrated synthesis uses the optimization technique setting, logic and RAM utilization of the design, and timing information from **Timing-Driven Synthesis** to determine which shift registers are located in memory and which are located in registers. To disable inference, turn off this option for the entire 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. Even if the logic option is set to **On** or **Auto**, the software might not infer small shift registers because small shift registers 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 or module.

RAM and ROM

Use the **Auto RAM Replacement** and **Auto ROM Replacement** logic options to control RAM and ROM inference, respectively. To disable inference, turn off the appropriate option for the entire 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 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 or 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 black 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.

Resource Aware RAM, ROM, and Shift-Register Inference

The Quartus II integrated synthesis takes resource usage into account when inferring RAM, ROM, and shift registers. During RAM, ROM, and shift register inferencing, synthesis looks at the number of memories available in the current device and does not infer more memory than is available to avoid a no-fit error. Synthesis tries to select the memories that are not inferred in a way that aims at the smallest increase in logic and registers.

Resource aware RAM, ROM and shift register inference is controlled by the **Resource Aware Inference for Block RAM** option. You can disable this option for the entire project in the **More Analysis & Synthesis Settings** dialog box, or per partition in the Assignment Editor.

This option is not supported in Stratix V devices.

When the **Auto** setting is selected, resource aware RAM, ROM, and shift register inference uses the resource counts from the largest device.

For designs with multiple partitions, Quartus II integrated synthesis considers one partition at a time. Therefore, for each partition, it assumes that all RAM blocks are available to that partition. If this causes a no-fit error, the number of RAM blocks available per partition can be limited with the following settings in the assignment editor: **Maximum Number of M512 Memory Blocks**, **Maximum Number of M4K/M9K Memory Blocks**, **Maximum Number of M-RAM/M144K Memory Blocks** and **Maximum Number of LABs**. These options are also used by the balancer. For more information, refer to "Limiting Resource Usage in Partitions" on page 9–32.

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. 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 the entire project in the **More Analysis & Synthesis Settings** dialog box.

For Arria GX and 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`", "`M144K`" and "`M20K`"(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 9–65 through Example 9–67 specify that all memory in the module or entity `my_memory_blocks` should be implemented using a specific type of block.

Example 9–65. Verilog-1995 Code: Applying a romstyle Attribute to a Module Declaration

```
module my_memory_blocks (...) /* synthesis romstyle = "M4K" */;
```

Example 9–66. Verilog-2001 and SystemVerilog Code: Applying a ramstyle Attribute to a Module Declaration

```
(* ramstyle = "M512" *) module my_memory_blocks (...);
```

Example 9–67. 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 9–68 through Example 9–70 specify that the inferred memory `my_ram` or `my_rom` should be implemented using regular logic instead of a TriMatrix memory block.

Example 9–68. Verilog-1995 Code: Applying a syn_ramstyle Attribute to a Variable Declaration

```
reg [0:7] my_ram[0:63] /* synthesis syn_ramstyle = "logic" */;
```

Example 9–69. Verilog-2001 and SystemVerilog Code: Applying a romstyle Attribute to a Variable Declaration

```
(* romstyle = "logic" *) reg [0:7] my_rom[0:63];
```

Example 9–70. 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";
```

You can control the depth of an inferred memory block using the `max_depth` attribute. By using this attribute, you can optimize the usage of the memory block. Example 9–71 through Example 9–73 specify the depth of the inferred memory `mem` using the `max_depth` synthesis attribute.

Example 9–71. Verilog-1995 Code: Applying a max_depth Attribute to a Variable Declaration

```
reg [7:0] mem [127:0] /* synthesis max_depth = 2048 */
```

Example 9–72. Verilog-2001 and SystemVerilog Code: Applying a max_depth Attribute to a Variable Declaration

```
(* max_depth = 2048*) reg [7:0] mem [127:0];
```

Example 9–73. VHDL Code: Applying a max_depth Attribute to a Variable Declaration

```
type ram_block is array (0 to 31) of std_logic_vector (2 downto 0);
signal mem : ram_block;
attribute max_depth : natural;
attribute max_depth OF mem : signal is 2048;
```

The syntax for setting these attributes in HDL is the same as the syntax for other synthesis attributes, as shown in "Synthesis Attributes" on page 9–25.

Turning Off the Add Pass-Through Logic to Inferred RAMs no_rw_check Attribute

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 should 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 with the Quartus II GUI, click **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box. Example 9–74 and Example 9–75 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 a defined read-during-write condition is not required in your design, the extra logic is not needed. With the `no_rw_check` attribute, Quartus II integrated synthesis does not generate the extra logic.

Example 9–74. 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 9–75. 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 in which some asynchronous RAM blocks might be coded with read-during-write behavior that does not match the Stratix III, Stratix IV, and Stratix V architectures. 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 these attributes, set the value of the ramstyle attribute to "`MLAB, no_rw_check`".

Example 9–76 and Example 9–77 show the method of setting two values to the `ramstyle` attribute using a small asynchronous 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.

Example 9–76. 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 9–77. 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 **.mif**. The attribute takes a string value containing the name of the RAM initialization file.

Example 9–78. Verilog-1995 Code: Applying a ram_init_file Attribute

```
reg [7:0] mem[0:255] /* synthesis ram_init_file
= " my_init_file.mif" */;
```

Example 9–79. Verilog-2001 Code: Applying a ram_init_file Attribute

```
(* ram_init_file = "my_init_file.mif" *) reg [7:0] mem[0:255];
```

Example 9–80. 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** 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 the dedicated multiplication hardware.

Example 9–81. Verilog-1995 Code: Applying a multstyle Attribute to a Module Declaration

```
module my_module (...) /* synthesis multstyle = "dsp" */;
```

Example 9–82. 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 which has a result that is directly assigned to the variable. It overrides the `multstyle` attribute associated with the enclosing module, if present. In Example 9–83 and Example 9–84, the `multstyle` attribute applied to variable `result` directs the Quartus II software to implement `a * b` in general logic rather than the dedicated hardware.

Example 9–83. 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 9–84. 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 9–85, the `multstyle` attribute indicates that `a * b` should be implemented in the dedicated hardware.

Example 9–85. 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 9–86, 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 9–86. 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 which has a result that is directly assigned to the signal or variable. It overrides the `multstyle` attribute associated with the enclosing entity or architecture, if present. In Example 9–87, the `multstyle` attribute associated with signal `result` directs the Quartus II software to implement `a * b` in general logic rather than the dedicated hardware.

Example 9–87. 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 should 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 the Verilog HDL functional and the post-Quartus II simulation because unknown case statement cases can still function as latches during functional simulation. For example, a simulation mismatch can occur with the code in Example 9–88 when `sel` is `2'b11` because a functional HDL simulation output behaves as a latch and the Quartus II simulation output behaves as a "don't care" value.

Altera recommends making the case statement "full" in your regular HDL code, instead of using the `full_case` attribute.

The case statement in Example 9–88 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 9–88. 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 9–89 in the `case` header instead of the comment form shown in Example 9–88.

Example 9–89. 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 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.

Altera recommends that you only use this attribute when the `case` statement is truly parallel. If you use the attribute in any other situation, the generated logic does 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 the Verilog HDL functional and the 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 9–90 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 9–90. 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 9–91 in the `case` (or `casez`) header instead of the comment form, as shown in Example 9–90.

Example 9–91. 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 9–92, Example 9–93, and Example 9–94 show these directives.

Example 9–92. Verilog HDL Code: Translate Off and On

```
// synthesis translate_off
parameter tpd = 2;    // Delay for simulation
#tpd;
// synthesis translate_on
```

Example 9–93. VHDL Code: Translate Off and On

```
-- synthesis translate_off
use std.textio.all;
-- synthesis translate_on
```

Example 9–94. VHDL 2008 Code: Translate Off and On

```
/* synthesis translate_off */
use std.textio.all;
/* synthesis translate_on */
```

If you want 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 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 9–95, Example 9–96, and Example 9–97, the commented code enclosed by `read_comments_as_HDL` is visible to the Quartus II Compiler and is synthesized. VHDL 2008 allows block comments, which is also supported for synthesis directives.

Because synthesis directives are case-sensitive in Verilog HDL, you must match the case of the directive, as shown in the following examples.

Example 9–95. 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 9–96. 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
```

Example 9–97. VHDL 2008 Code: Read Block 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.

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 9–98 and Example 9–99, 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 9–98. 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 9–99 and Example 9–100 instead of the comment form shown in Example 9–98.

Example 9–99. Verilog-2001 Code: the useioff Attribute

```
(* useioff = 1 *)   input [1:0] a, b;
(* useioff = 1 *)   output [2:0] o;
```

Example 9–100. 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 allows 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 the `chip_pin` attribute, 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 9–101 through Example 9–103 show different ways of assigning input pin `my_pin1` to Pin C1 and `my_pin2` to Pin 4 on a different target device.

Example 9–101. 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 9–102. 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 9–103. 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, leave its corresponding pin assignment blank.

Example 9–104 assigns `my_pin[2]` to `Pin_4`, `my_pin[1]` to `Pin_5`, and `my_pin[0]` to `Pin_6`.

Example 9–104. Verilog-1995 Code: Applying Chip Pin to a Bus of Pins

```
input [2:0]  my_pin /* synthesis chip_pin = "4, 5, 6" */;
```

Example 9–105 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 9–105. Verilog-1995 Code: Applying Chip Pin to Part of a Bus

```
input [0:2]  my_pin /* synthesis chip_pin = "4, ,6" */;
```

Example 9–106 assigns `my_pin[2]` to Pin 4 and `my_pin[0]` to Pin 6, but leaves `my_pin[1]` unassigned.

Example 9–106. 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 allows you to apply Quartus II logic 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 & Synthesis, such as Fitting.

Assignments or settings made through the Quartus II GUI, the **.qsf**, 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** variable assignments separated by semicolons, as shown in Example 9–107.

Example 9–107. 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 9–108 for each **.qsf** variable assignment.

Example 9–108. Syntax for Each .qsf 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** assignments, is shown in Example 9–109.

Example 9–109. 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 the assigned value of a variable is a string of text, you must use escaped quotes around the value in Verilog HDL (as shown in Example 9–110), or double-quotes in VHDL (as shown in Example 9–111):

Example 9–110. Assigned Value of a Variable in Verilog HDL (With Non-Existent Variable and Value Terms)

```
"VARIABLE_NAME \"STRING_VALUE\""
```

Example 9–111. Assigned Value of a Variable in VHDL (With Non-Existent Variable and Value Terms)

```
"VARIABLE_NAME ""STRING_VALUE"""
```

To find the **.qsf** 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**. You can also refer to the *Quartus II Settings File Reference Manual*, which documents all variable names.

Example 9–112 through Example 9–114 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. Therefore, you must apply the attribute to one of the instance's output nets. The Quartus II software moves the attribute to the inferred instance automatically.

Example 9–112. Verilog-1995 Code: Applying altera_attribute to an Instance

```
reg my_reg /* synthesis altera_attribute = "-name POWER_UP_LEVEL HIGH"
*/;
```

Example 9–113. Verilog-2001 Code: Applying altera_attribute to an Instance

```
(* altera_attribute = "-name POWER_UP_LEVEL HIGH" *) reg my_reg;
```

Example 9–114. 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 9–115 through Example 9–117 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 9–115. Verilog-1995 Code: Applying altera_attribute to an Entity

```
module my_entity(…) /* synthesis altera_attribute = "-name
AUTO_SHIFT_REGISTER_RECOGNITION OFF" */;
```

Example 9–116. Verilog-2001 Code: Applying altera_attribute to an Entity

```
(* altera_attribute = "-name AUTO_SHIFT_REGISTER_RECOGNITION OFF" *)
module my_entity(…) ;
```

Example 9–117. 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 9–119 through Example 9–121, the `altera_attribute` cuts all timing paths from `reg1` to `reg2`, equivalent to this Tcl or QSF command, as shown in Example 9–118:

Example 9–118.

```
set_instance_assignment -name CUT ON -from reg1 -to reg2 ↵
```

Example 9–119. Verilog-1995 Code: Applying altera_attribute with the -to Option

```
reg reg2;
reg reg1 /* synthesis altera_attribute = "-name CUT ON -to reg2" */;
```

Example 9–120. Verilog-2001 and SystemVerilog Code: Applying altera_attribute with the -to Option

```
reg reg2;
(* altera_attribute = "-name CUT ON -to reg2" *) reg reg1;
```

Example 9–121. VHDL Code: Applying altera_attribute with the -to Option

```
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 performing synthesis, you can check your synthesis results in the **Analysis & Synthesis** section of the Compilation Report and the Project Navigator.

Analysis & Synthesis Section of the Compilation Report

The Compilation Report, which provides a summary of results for the project, appears after a successful compilation. After Analysis & Synthesis, the Summary section 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 & 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 *Analysis & Synthesis Summary Reports* in 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 & 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 & Synthesis describe many of the optimizations that the software performs during the synthesis stage, and provide information about how the design is interpreted. Altera recommends checking 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.

For more information about the Messages window and message suppression, refer to *About the Messages Window* and *About Message Suppression* in Quartus II Help and *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

In the Quartus II software version 8.1 and later, you can specify the type of Analysis & 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**.
2. In the **Category** list, click **Analysis & Synthesis Settings**.
3. Click **More Settings**. Select the level for the **Analysis & Synthesis Message Level** option.

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, Altera recommends investigating 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 9–122, 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 typing error: Variable `j` should be listed on the sensitivity list to avoid a possible simulation or synthesis mismatch.

Example 9–122. 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 the HDL code, the Quartus II software generates the following warning message, as shown in Example 9–123:

Example 9–123.

```
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 9–124 are two different variables. However, declaring variables which have names in different cases is potentially confusing, especially if you use VHDL, in which variables are not case-sensitive.

Example 9–124. Generating HDL Info Messages

```
// namecase.v
module namecase (input i, output o);
   reg my_reg;
   reg MY_REG;
   assign o = i;
endmodule
```

When processing the HDL code, the Quartus II software generates the following informational message, as shown in Example 9–125:

Example 9–125.

```
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 nor MY_REG are used in this small design, as shown in Example 9–126:

Example 9–126.

```
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 can view during the Analysis & 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 describes the HDL message levels.

Table 9–8. HDL Info Message Level

Level	Purpose	Description
Level1	Displays high-severity messages only	If you only want to view the 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.

Table 9–8. HDL Info Message Level

Level	Purpose	Description
Level2	Displays high-severity and medium-severity messages	If you want to view 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 view 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 must address all issues reported at the **Level1** setting. The default HDL message level is **Level2**.

To set the HDL Message Level in the GUI, follow these steps:

1. On the Assignments menu, click **Settings.**
2. In the **Category** list, click **Analysis & Synthesis Settings.**
3. Set the desired message level from the pull-down menu in the **HDL Message Level** list, and then 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 9–127 and Example 9–128.

Example 9–127. Verilog HDL Examples of message_level Directive

```
// altera message_level level1
or
/* altera message_level level3 */
```

Example 9–128. 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 to 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, follow these steps:

1. On the Assignments menu, click **Settings**.

2. In the **Category** list, expand **Analysis & Synthesis Settings** and select **Advanced**.
3. In the **Advanced Message Settings** dialog box, add the Message IDs you want to enable or disable.

To enable or disable specific HDL messages in your HDL, use the `message_on` and `message_off` synthesis directives. These 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 9–129. Verilog HDL message_off Directive for Message with ID 10000

```
// altera message_off 10000
or
/* altera message_off 10000 */
```

Example 9–130. VHDL message_off Directive for Message with ID 10000

```
-- altera message_off 10000
```

Node-Naming Conventions in Quartus II Integrated Synthesis

This section provides an overview of the conventions used by the Quartus II software during synthesis to name the nodes created from your HDL design. This information is useful for finding logic node names during verification and debugging of a design. This section focuses on the conventions for Verilog HDL and VHDL code, but AHDL and **.bdf** files 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 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 9–79
- "Preserving Combinational Logic Names" on page 9–80

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 indicates 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 defines 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 in Example 9–131:

Example 9–131.

<entity 0>`:`*<instance_name 0>*`|`*<entity 1>*`:`*<instance_name 1>*`|...|`*<instance_name n>*`|` *<node_name>*

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`.

To instruct the Compiler to generate node names that do not contain entity names, on the **Compilation Process Settings** page of the **Settings** dialog box, click **More Settings**, and then turn off D**isplay entity name for node name**. With this option turned off, the node names use the following convention in Example 9–132:

Example 9–132.

<instance_name 0>`|`*<instance_name 1>*`|...|`*<instance_name n>* `|`*<node_name>*

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.

Example 9–133 is a description of a register in Verilog HDL that creates a DFF primitive called `my_dff_out`:

Example 9–133. Verilog HDL Register

```
wire dff_in, my_dff_out, clk;

always @ (posedge clk)
my_dff_out <= dff_in;
```

Similarly, Example 9–134 is a description of a register in VHDL that creates a DFF primitive called `my_dff_out`.

Example 9–134. 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**, 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, Quartus II integrated synthesis appends `~reg0` to the register name.

For example, the Verilog HDL code in Example 9–135 produces a register called `q~reg0`:

Example 9–135. 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 view 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 9–78
- "Packed Input and Output Registers of RAM and DSP Blocks" on page 9–78
- "Preserving Register Names" on page 9–78
- "Preserving Combinational Logic Names" on page 9–80

Synthesis and Fitting Optimizations

Registers might be removed by logic optimization during synthesis 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. To generate the Quartus II Compilation Report, follow these steps:

1. In the **Analysis & Synthesis** folder, open **Optimization Results.**
2. Open **Register Statistics,** and then click the **Registers Removed During Synthesis** report.
3. Click **Removed Registers Triggering Further Register Optimizations**.

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.

For example, consider a Verilog HDL state machine in which the states are `parameter state0 = 1`, `state1 = 2`, `state2 = 3`, and in which 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 state 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 are 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 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 are 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

Altera recommends you 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. For details, refer to "Preserve Registers" on page 9–43.

Use the `noprune` attribute to preserve a fan-out-free register through the entire compilation flow. For details, refer to "Noprune Synthesis Attribute/Preserve Fan-out Free Register Node" on page 9–44.

Use the synthesis attribute `syn_dont_merge` to ensure that the registers are not merged with other registers, and other registers are not merged with them. For details, refer to "Disable Register Merging/Don't Merge Register" on page 9–44.

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 9–136. Quartus II integrated synthesis uses the names `c`, `d`, `e`, and `f` for the combinational logic cells that are produced.

Example 9–136. 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**, 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 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 in the Quartus II software version 7.2 and later use the Quartus II naming 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 then set the value to **QuartusII**.

If logic cells, such as those created in Example 9–136, 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`, and `w~2`. 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 helps you understand your post-synthesis results, making it easier to debug your design or create 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 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 with the 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.

For more information about **Clock MUX Protection** logic option and a list of supported devices, refer to *Clock MUX Protection logic option* in Quartus II Help.

Preserving Combinational Logic Names

You can 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. For details, refer to "Keep Combinational Node/Implement as Output of Logic Cell" on page 9–45.

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, as shown in Example 9–137:

Example 9–137.

```
quartus_sh --qhelp ↵
```

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook* and *API Functions for Tcl* in Quartus II Help. 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, at the global level, or both.

Use the following Tcl command, as shown in Example 9–138, to make a global assignment:

Example 9–138.

```
set_global_assignment -name <QSF Variable Name> <Value> ↵
```

Use the following Tcl command, as shown in Example 9–139, to make an instance assignment:

Example 9–139.

```
set_instance_assignment -name <QSF Variable Name> <Value>\ -to
<Instance Name> ↵
```

To set the **Synthesis Effort** option at the command line, use the `--effort` option with the `quartus_map` executable, as shown in Example 9–140.

Example 9–140. Command Syntax for Specifying Synthesis Effort Option

```
quartus_map <Design name> --effort= "auto | fast"
```

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. If you want to run fast synthesis with the Fitter **Early Timing Estimate** option, use the command shown in Example 9–141. This command runs the full flow with timing analysis.

Example 9–141. Command Syntax for Running Fast Synthesis with Early Timing Estimate Option

```
quartus_sh --flow early_timing_estimate_with_synthesis <Design name>
```

For more information, refer to "Synthesis Effort" on page 9–35.

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, as shown in Example 9–142:

Example 9–142.

```
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 option shown in Example 9–143, at the end of the VERILOG_FILE or VHDL_FILE command:

Example 9–143.

```
-   HDL_VERSION <language version>
```

The variable <*language version*> takes one of the following values:

- VERILOG_1995
- VERILOG_2001
- SYSTEMVERILOG_2005
- VHDL_1987
- VHDL_1993
- VHDL_2008

For example, to add a Verilog HDL file called **my_file.v** that is written in Verilog-1995, use the following command, as shown in Example 9–144:

Example 9–144.

```
set_global_assignment –name VERILOG_FILE my_file.v –HDL_VERSION \
VERILOG_1995
```

In Example 9–145, 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 9–145. 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 9–146. Verilog-2001 and SystemVerilog Code: Specifying User-Encoded States with the syn_encoding Attribute

```
(* 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 9–146, the states are encoded as follows:

```
init = "00"
last = "11"
next = "01"
later = "10"
```

Without the `syn_encoding` attribute, the Quartus II software encodes 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 9–39), separate the encoding style value in the quotation marks from the safe value with a comma, as follows: "`safe, one-hot`" or "`safe, gray`".

For more information, refer to "Manually Specifying State Assignments Using the syn_encoding Attribute" on page 9–37.

Assigning a Pin

Use the following Tcl command, as shown in Example 9–147, to assign a signal to a pin or device location:

Example 9–147.

```
set_location_assignment -to <signal name> <location>
```

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`, in which `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, as shown in Example 9–148:

Example 9–148.

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

The <*file name*> is the name used for internally generated netlist files during incremental compilation. If you create the partition in the Quartus II GUI, netlist files, netlist files are named automatically by the Quartus II software based on the instance name. 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 should 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 <*destination*> is 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 should 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 (" ").

Quartus II Synthesis Options

Table 9–9 lists the **.qsf** variable names and applicable values for the settings discussed in this chapter. The **.qsf** variable name is used in the Tcl assignment to make the setting along with the appropriate value.

When you apply a Quartus II Synthesis option globally or to an entity, it affects all lower-level entities in the hierarchy path, including entities instantiated with Altera and third-party IP.

Table 9–9. Quartus II Synthesis Options (Part 1 of 4) *(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

Table 9–9. Quartus II Synthesis Options (Part 2 of 4) *(Note 1)*

Setting Name	Quartus II Settings File Variable	Values
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*>
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 Maximum Fan-Out Assignments	IGNORE_MAX_FANOUT_ASSIGNMENTS	On/Off
Ignore ROW GLOBAL Buffers	IGNORE_ROW_GLOBAL_BUFFERS	On/Off

Table 9–9. Quartus II Synthesis Options (Part 3 of 4) *(Note 1)*

Setting Name	Quartus II Settings File Variable	Values
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>*
Maximum Number of LABs	MAX_LABS	*<Maximum number of LAB 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>*
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
Resource Aware Inference for Block RAM	SYNTH_RESOURCE_AWARE_INFERENCE_FOR_ BLOCK_RAM	On/Off
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

Table 9–9. Quartus II Synthesis Options (Part 4 of 4) *(Note 1)*

Setting Name	Quartus II Settings File Variable	Values
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
Synthesis Seed *(2)*	SYNTHESIS_SEED	*<Non-negative integer>*
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.

Conclusion

The Quartus II software includes Verilog HDL, SystemVerilog, and VHDL language support, as well as support for Altera-specific languages, making it an easy-to-use, stand-alone solution for Altera designs. You can use the synthesis options in the software or in your HDL code to better control the way your design is synthesized, helping you improve your synthesis results. Use Quartus II reports and messages to analyze your compilation results.

Document Revision History

Table 9–10 shows the revision history for this chapter.

Table 9–10. Document Revision History (Part 1 of 3)

Date	Version	Changes Made
July 2010	10.0.0	■ Removed Referenced Documents section. ■ Added "Synthesis Seed" on page 9–35 section. ■ Updated the following sections: → "SystemVerilog Support" on page 9–5 → "VHDL-2008 Support" on page 9–10 → "Using Parameters/Generics" on page 9–16 → "Parallel Synthesis" on page 9–20 → "Limiting Resource Usage in Partitions" on page 9–32 → "Synthesis Effort" on page 9–35 → "Synthesis Attributes" on page 9–25 → "Synthesis Directives" on page 9–27 → "Auto Gated Clock Conversion" on page 9–28 → "State Machine Processing" on page 9–36 → "Multiply-Accumulators and Multiply-Adders" on page 9–50 → "Resource Aware RAM, ROM, and Shift-Register Inference" on page 9–52 → "RAM Style and ROM Style—for Inferred Memory" on page 9–53 → "Turning Off the Add Pass-Through Logic to Inferred RAMs no_rw_check Attribute" on page 9–54 → "Using altera_attribute to Set Quartus II Logic Options" on page 9–67 → "Adding an HDL File to a Project and Setting the HDL Version" on page 9–82 → "Creating Design Partitions for Incremental Compilation" on page 9–84 → "Inferring Multiplier, DSP, and Memory Functions from HDL Code" on page 9–50 ■ Updated Table 9–9 on page 9–84.
December 2009	9.1.1	■ Added information clarifying inheritance of Synthesis settings by lower-level entities, including Altera and third-party IP ■ Updated "Keep Combinational Node/Implement as Output of Logic Cell" on page 9–46

Table 9–10. Document Revision History (Part 2 of 3)

Date	Version	Changes Made
November 2009	9.1.0	■ Updated the following sections: → "Initial Constructs and Memory System Tasks" on page 9–7 → "VHDL Support" on page 9–9 → "Parallel Synthesis" on page 9–21 → "Synthesis Directives" on page 9–27 → "Timing-Driven Synthesis" on page 9–31 → "Safe State Machines" on page 9–40 → "RAM Style and ROM Style—for Inferred Memory" on page 9–53 → "Translate Off and On / Synthesis Off and On" on page 9–62 → "Read Comments as HDL" on page 9–63 → "Adding an HDL File to a Project and Setting the HDL Version" on page 9–81 ■ Removed "Remove Redundant Logic Cells" section ■ Added "Resource Aware RAM, ROM, and Shift-Register Inference" section ■ Updated Table 9–9 on page 9–83
March 2009	9.0.0	■ 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
November 2008	8.1.0	■ 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

Table 9–10. Document Revision History (Part 3 of 3)

Date	Version	Changes Made
November 2008	8.1.0	■ 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"
May 2008	8.0.0	■ 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

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

Take an online survey to provide feedback about this handbook chapter.

10. Synopsys Synplify Support

QII51009-10.0.0

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 supports active devices available in the current version of the Quartus II software. Support for newly released device families may require an overlay. Contact Synopsys at www.synopsys.com for more information.

The Synplify software also supports the FLEX 8000 and MAX 9000 legacy devices that are supported only in the Altera MAX+PLUS® II software, as well as ACEX® 1K, APEX™ II, APEX 20K, APEX 20KC, APEX 20KE, FLEX® 10K, and FLEX 6000 legacy devices that are supported by the Quartus II software version 9.0 and earlier.

Design Flow

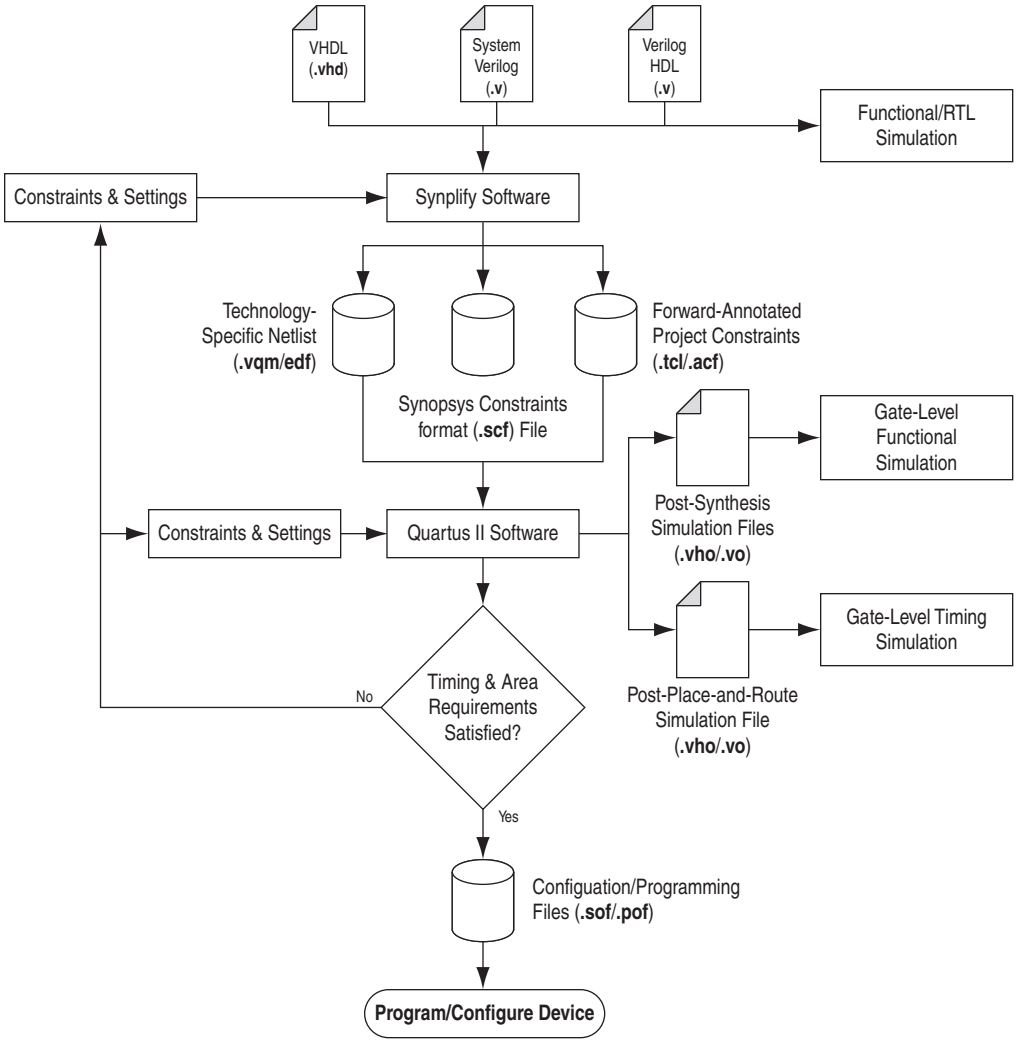
A Quartus II software design flow using the Synplify software consists of the following steps:

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. Use the following files 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 by the MAX+PLUS II software
   - The Synopsys Constraints Format (**.scf**) file for TimeQuest timing constraints
   - The Tcl constraints file (**.tcl**)

   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 10–14.
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 a design in a SCOPE Design Constraints File (**.sdc**) with the SCOPE window in the Synplify software using standard Synopsys Design Constraint (SDC) format, 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. (See Table 10–1 on page 10–5 for a list of the files generated by Synplify.)

The HDL Analyst that is included in the Synplify software is a graphical tool for generating schematic views of the technology-independent 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. The 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 older 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 in the Synplify software and the placement constraints in the Quartus II software. 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

File Extensions	File Description
.srs	Technology-independent RTL netlist that can be read only by the Synplify software.
.srm	Technology view netlist.
.srr *(1)*	Synthesis Report file.
.vqm/**.edf**	Technology-specific netlist in **.vqm** or **.edf** file format. An **.edf** file is created for devices supported by the MAX+PLUS II software. A **.vqm** file is created for all other Altera device families.
.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.
.acf	Assignment and Configurations file for backward compatibility with the MAX+PLUS II software. For devices supported by the MAX+PLUS II software, the MAX+PLUS II assignments are imported from the MAX+PLUS II **.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

To specify the output netlist directory location and name for the Synplify software, perform 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.

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 devices supported by the MAX+PLUS II software. For current 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. This section provides an overview of some of the techniques you can use to help improve the quality of your 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*.

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

Compared to the other Synplify products, Synplify Premier offers additional physical synthesis optimizations. 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. This section explains some of the important timing constraints in the Synplify software.

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 10–17 for more details about how constraints such as clock frequencies, false paths, and multicycle paths are forward-annotated.

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 that has been 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 state machines. The FSM Compiler analyzes state machines 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 a state machine is sequential, the implementation is also sequential. If the FSM Compiler is turned on, the compiler infers and optimizes 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, leading to long compile times and difficult fitting.

If you must duplicate an output register or an 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 that 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.

The `define_reg_input_delay` and `define_reg_output_delay` 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 */ ;
```

I/O Standard

For certain Altera devices, specify the I/O standard type to use for an 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, which 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 for supported Altera devices. Pin location assignments for these devices are written to the output Tcl script.

Example 10–4. Making Location Assignments in 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 series 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, and describes the following topics:

- "Running the Quartus II Software from within the Synplify Software"
- "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 10–17

Running the Quartus II Software from within the Synplify Software

To use the Quartus II software from within the Synplify software, you must set the `QUARTUS_ROOTDIR` environment variable to 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. Synplify also uses this variable to call the Quartus II software in the background to obtain detailed information for Altera megafunctions used in the design.

In the Windows operating system, set the variable using the Control Panel, System options. In the **Advanced** tab, click **Environment Variables**, and create a `QUARTUS_ROOTDIR` system variable.

In the Linux operating system, create an environment variable `QUARTUS_ROOTDIR` that points to the *<Altera Design Suite Installation Directory>*/**quartus** location.

You can create new place and route implementations using the **New P&R** button in the GUI. Under each implementation, the Synplify Pro software creates a place-and-route implementation called **pr**_<*number*> **Altera Place and Route**. 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 to 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 assignments, 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.
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 Synplify-generated **.tcl** file contains 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 10–17. For the TimeQuest Timing Analyzer, the timing constraints are forward-annotated in the Synopsys Constraints Format (**.scf**) file.

For the newest devices, Altera recommends that you use the TimeQuest Timing Analyzer, as specified in the Synplify **.tcl** file that sets up the Quartus II project. 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, rather than using 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 10–17; 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. These 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 10–5.

Example 10–5. 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.

Table 10–3. Command Arguments (Part 1 of 2)

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.

Table 10–3. Command Arguments (Part 2 of 2)

Argument	Description
`-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

In the Synplify software, the `-route` option clock constraint is designed for 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 10–6.

Example 10–6. 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 10–7.

Example 10–7. 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 10–8.

Example 10–8. 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, using the commands shown in Example 10–9.

Example 10–9. 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 10–10.

Example 10–10. 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 10–11.

Example 10–11. 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 in 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 in the Quartus II Classic Timing Analyzer.

Register Input and Output Delays

The 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 takes better advantage of timing-driven optimization, rather 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 turn on 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 turn on 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 model, 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 10–12 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 the IP Toolbench.

Example 10–12. 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 10–13 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 the IP Toolbench.

Example 10–13. 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 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 10–14 for a Verilog HDL example.

Example 10–14. 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 10–15 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 gray box netlist for resource and timing estimation. The files **core.v** and **core_enc8b10b.v** are not compiled by Synplify, but are copied into the place-and-route directory. The Quartus II software compiles these files to implement the "core" IP block.

Example 10–15. 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

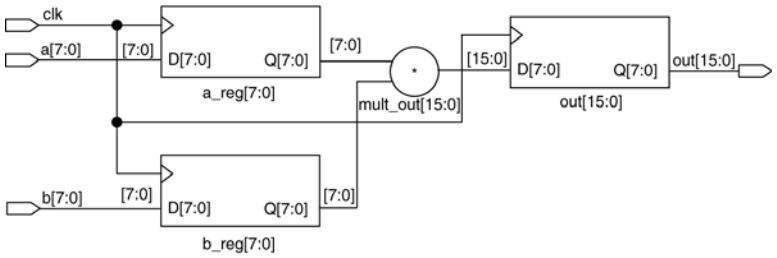
The Synplify software uses Behavior Extraction Synthesis Technology (BEST) algorithms to infer high-level structures such as RAMs, ROMs, operators, FSMs, and DSP multiplication operations. It then keeps the structures abstract for as long as possible in the synthesis process. This allows the use of technology-specific 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 some 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 10–16 and Example 10–17 show simple Verilog HDL and VHDL code using the `syn_multstyle` attribute.

Example 10–16. 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 10–17. 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 some 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 some 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 some 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 10–18 shows sample VHDL code for inferring dual-port RAM.

Example 10–18. 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 10–19 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 10–19. 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 10–20 and Example 10–21 illustrate how RAM memories can be initialized through HDL code, and how the corresponding **.hex** file is generated using Verilog HDL.

Example 10–20. 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 10–21. Sample of .vqm Instance Containing Memory Initialization File from Example 10–20

```
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 some 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. 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 steps show 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.

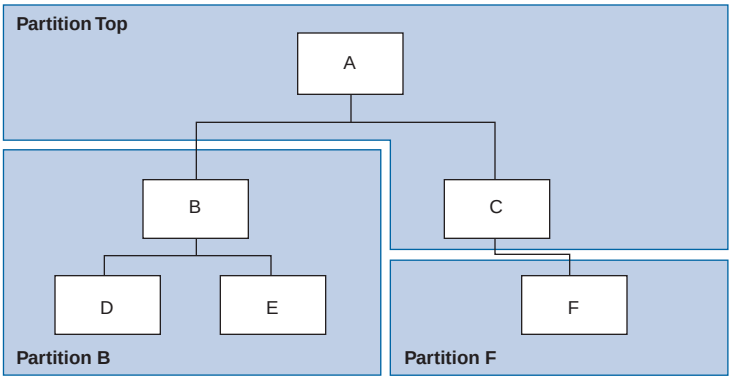
Set Compile Points and Create Constraint Files

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 to the architecture in the Synplify Pro SCOPE spreadsheet, or to 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 10–22.

Example 10–22. The define_compile_point Command

```
define_compile_point [-disable] {<objname>} -type {locked, partition}
```

In Example 10–22, *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 method, the lower-level constraint file is created automatically:

1. On the File menu, click **New** and choose to create a new **Constraint File**.
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 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.

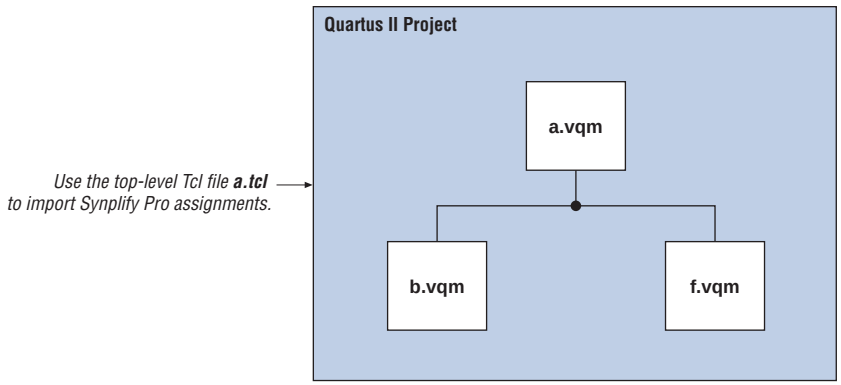
You might require a bottom-up design flow if each partition must be optimized separately, such as in certain team-based design flows. If you use this flow, Altera recommends you create a design floorplan to avoid placement conflicts between each partition. To perform a bottom-up compilation in the Quartus II software, create separate Quartus II projects, 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, while taking advantage of the performance preservation and compilation-time reduction that 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.

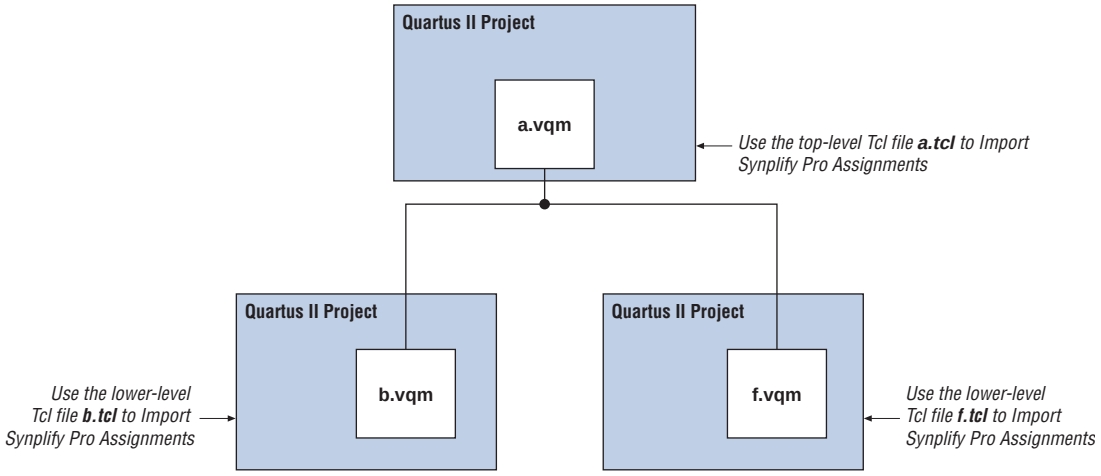
Figure 10–4. Design Flow Using Multiple .vqm Files with One Quartus II Project



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



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 by 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, engineers can work independently 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 10–23 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 10–23. 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 10–24 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 10–24. 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; import each design partition into a top-level design using the incremental compilation export, and import features to maintain the results.

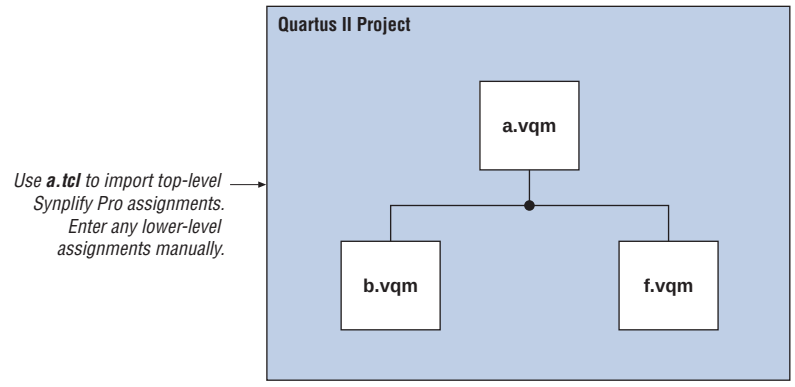
The following sections describe how to create the Quartus II projects for these two design flows.

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 constraints made in the lower-level projects within the Synplify software are 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

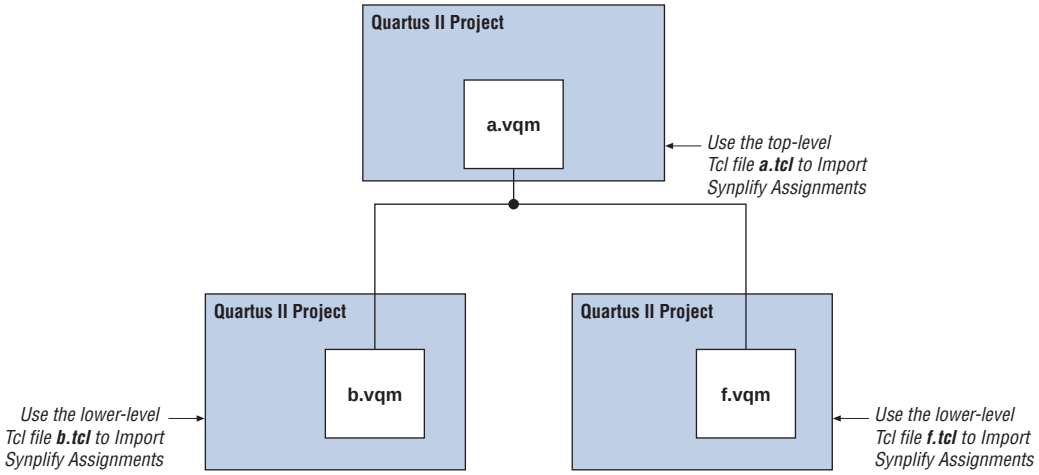


Creating Multiple Quartus II Projects for a Bottom-Up Incremental Compilation Flow

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.

Document Revision History

Table 10–5. Document Revision History

Date	Version	Changes Made
July 2010	v.10.0.0	■ Minor updates for the Quartus II software version 10.0 release
November 2009	v9.1.0	■ Minor updates for the Quartus II software version 9.1 release
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
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
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

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

Take an online survey to provide feedback about this chapter.

11. Mentor Graphics Precision Synthesis Support

QII51011-10.0.0

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.

The topics discussed in this chapter include:

- "Altera Device Family Support"
- "Design Flow" on page 11–2
- "Creating and Compiling a Project in the Precision Synthesis Software" on page 11–5
- "Mapping the Precision Synthesis Design" on page 11–6
- "Synthesizing the Design and Evaluating the Results" on page 11–10
- "Exporting Designs to the Quartus II Software Using NativeLink Integration" on page 11–11
- "Guidelines for Altera Megafunctions and Architecture-Specific Features" on page 11–18
- "Incremental Compilation and Block-Based Design" on page 11–27

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**.

Altera Device Family Support

The Precision Synthesis software supports active devices available in the current version of the Quartus II software. Support for newly released device families may require an overlay. Contact Mentor Graphics for more information.

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, as well as ACEX® 1K, APEX™ II, APEX 20K, APEX 20KC, APEX 20KE, FLEX® 10K, and FLEX 6000 legacy devices that are supported by the Quartus II software version 9.0 and earlier.

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 11–5 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.

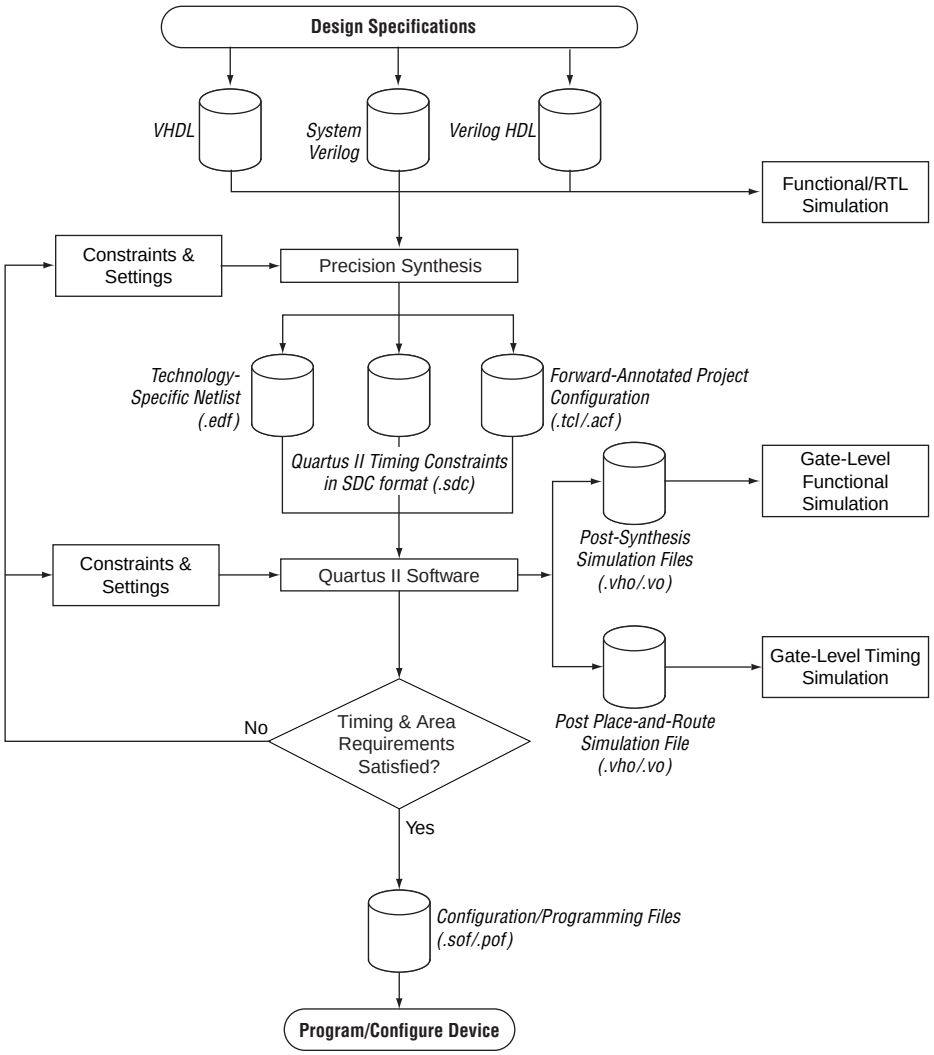
   For best results, Mentor Graphics recommends specifying constraints that are as close as possible to actual operating requirements. Properly setting clock and I/O constraints, assigning clock domains, and indicating false and multicycle paths guide the synthesis algorithms more accurately toward a suitable 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 File (**.sdc**) for the TimeQuest Timing Analyzer
   - Tcl Script Files (**.tcl**) to set up your Quartus II project and pass constraints

   You can run the Quartus II software from within the Precision Synthesis software, or run the Precision Synthesis software using the Quartus II software. Refer to "Running the Quartus II Software from within the Precision Synthesis Software" on page 11–11 and "Using the Quartus II Software to Run the Precision Synthesis Software" on page 11–12 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



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 **WYSIWYG Primitive Resynthesis**, 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 *(2)*	Technology-specific netlist in electronic design interchange format (EDIF).
.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.
.acf	Assignment and Configurations file for backward compatibility with the MAX+PLUS II software. For devices supported by the MAX+PLUS II software, the MAX+PLUS II assignments are imported from the MAX+PLUS II **.acf** file.
.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.

Notes 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 11–10 for details.

(2) The Precision Synthesis software can also generate a Verilog Quartus Mapping File (**.vqm**); however, the Precision-generated VQM is not supported by the Quartus II software version 10.0.

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**. The 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.
7. On the **Design Center** tab, right-click the **Output Files** folder and choose **Output Options**.
8. In the **Options** dialog box, select **EDIF**.

   The Precision software can also generate a Verilog Quartus Mapping File (**.vqm**); however, the Precision-generated VQM is not supported by the Quartus II software version 10.0.

9. To generate additional HDL netlists for post-synthesis simulation, select the desired output format. The Precision Synthesis software generates a separate file for each selected type of file: EDIF and Verilog HDL or 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 during 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** 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 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.

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" on page 11–9 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.
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 the 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 more 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; therefore, the I/O pins 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, perform the following 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 cause significant delays that result in an unroutable net. On a critical path, high fan-out nets can cause longer 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, and then writes out netlists and reports to the implementation subdirectory of your working directory after the implementation is saved, using the following 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 you should use the place-and-route software 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, which is used by the Fitter and the TimeQuest Timing Analyzer in the Quartus II software.

If you want to use the 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 uses individual Quartus II executables, such as analysis and synthesis (`quartus_map`), Fitter (`quartus_fit`), and the TimeQuest Timing Analyzer (`quartus_sta`) for improved runtime and memory utilization during place and route. This flow is referred to as the **Quartus II Modular** flow option in the Precision Synthesis software. By default, the Precision Synthesis software generates a Quartus II Project Configuration File (**.tcl** file) for current device families. Timing constraints that you set during synthesis are exported to the Quartus II place-and-route constraints file *<project name>*_**pnr_constraints.sdc**.

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.

For details on running the Quartus II Software from within the Precision Synthesis Software, refer to the *Altera Quartus II Integration* chapter in the *Precision Synthesis Reference Manual*. To access this manual in the Precision Synthesis software, click **Help** and select **Open Manuals Bookcase**.

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 the Quartus II Software to Run 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. When you use this feature, the Precision Synthesis software does not use any timing constraints or assignments such as incremental compilation partitions that you have set in the Quartus II software.

For detailed information about using NativeLink integration with the Precision Synthesis software, refer to *Using the NativeLink Feature with Other EDA Tools* in the Quartus II Help.

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 TimeQuest Timing Analyzer and the Classic Timing Analyzer.
- Clocks in the same clock domain are annotated with the `create_relative_clock` command to create related clocks for the Classic Timing Analyzer.
- The 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 a place-and-route Tcl constraints file, this false path 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 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 on page 11–17 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, the multicycle path setting in the Precision Synthesis software is mapped to a `set_multicycle_assignment` setting for the Classic Timing Analyzer. For the 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 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 instantiate and parameterize the function using the port and parameter definitions, or you can customize a function with the MegaWizard™ Plug-In Manager. Altera recommends using the MegaWizard Plug-In Manager, which 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 11–19 describe the MegaWizard Plug-In Manager flow with the Precision Synthesis software.

For more information about specific Altera megafunctions and IP functions, refer to the Altera IP and Megafunctions website.

The Precision Synthesis software automatically recognizes certain types of HDL code and infers the appropriate function. 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 11–21.

For a detailed discussion about instantiating functions versus inferring functions to target Altera architecture-specific features, 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 functions 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 arithmetical and relational operators, and memory (RAM and ROM), to efficient technology-specific implementations. This automatic capability allows for technology-specific resources to implement these structures by inferring the appropriate Altera function to provide the 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 technology-specific architecture 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**, which gives the 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 also allows you to control the device resources used to implement multiply-accumulators or multiply-adders in your project or in a particular module.

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 the software maps these functions 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. 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 on page 11–26 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. You can perform optimization on individual blocks and then integrate them into a final design and optimize the design 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 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 cycle time.

Using the incremental synthesis feature, you can create different netlist files for different partitions of a design hierarchy within one partition implementation, which makes each partition independent of the others in an incremental compilation flow. 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 11–28.
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:

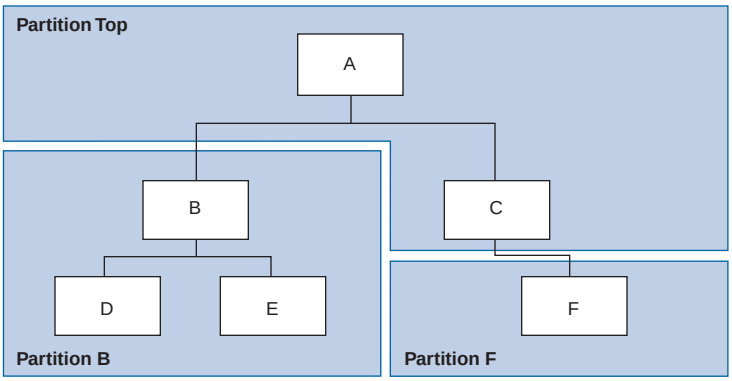
1. Read the HDL files for top-level designs.
2. On the Tools menu, click **Set Options**. On the **Optimization** page, turn on **Add IO Pads**.
3. Create black boxes for lower-level modules in the top-level design.
4. Add constraints.

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.

Creating Black Boxes to Create EDIF Netlists

This section describes how to create 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 11–12.

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.

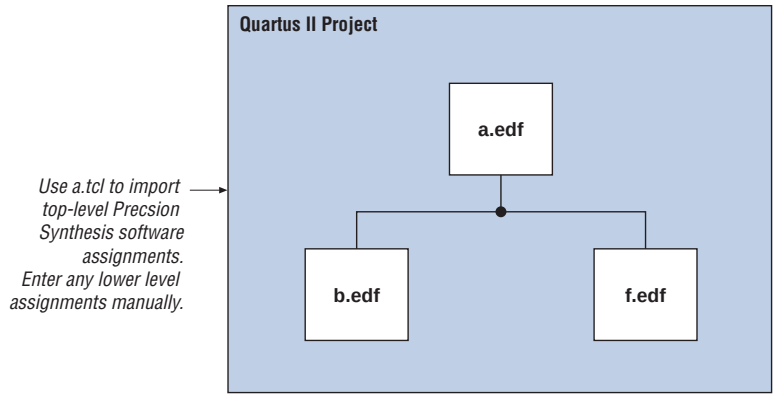
The following sections describe how to create the Quartus II projects for these two design flows.

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.

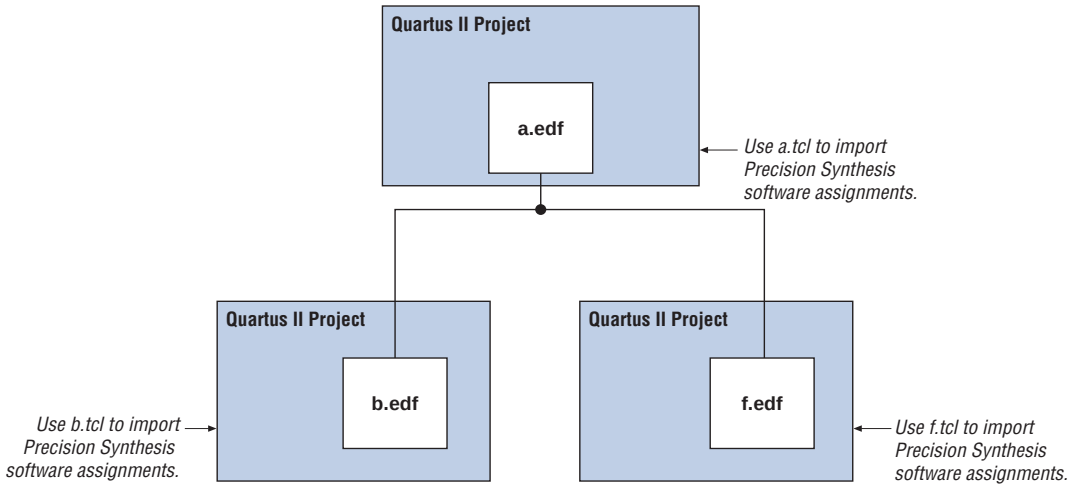
Figure 11–3. Design Flow Using Multiple EDIF Files with One Quartus II Project



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, which allows you to improve performance and optimizes 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.

Document Revision History

Table 11–7 shows the revision history for this chapter.

Table 11–7. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Minor updates for the Quartus II software version 10.0 release
November 2009	9.1.0	■ Minor updates for the Quartus II software version 9.1 release
March 2009	9.0.0	■ 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
November 2008	8.1.0	■ 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
May 2008	8.0.0	■ 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

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

Take an online survey to provide feedback about this chapter.

12. Mentor Graphics LeonardoSpectrum Support

QII51010-10.0.0

This chapter documents key design methodologies and techniques for Altera® devices using the LeonardoSpectrum and Quartus II design flow.

This chapter includes the following sections:

- "Altera Device Family Support"
- "Design Flow" on page 12–2
- "Optimization Strategies" on page 12–4
- "Timing Analysis with the LeonardoSpectrum Software" on page 12–6
- "Exporting Designs Using NativeLink Integration" on page 12–7
- "Guidelines for Altera Megafunctions and LPM Functions" on page 12–8
- "Block-Based Design with the Quartus II Software" on page 12–16

Altera recommends using the advanced Mentor Graphics Precision RTL Synthesis software for new designs in new device families. For more information about Precision RTL Synthesis, refer to the *Mentor Graphics Precision 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*.

Altera Device Family Support

The LeonardoSpectrum software is a mature synthesis tool supporting legacy devices and many current devices; however, newly-released devices may not be supported.

Contact Mentor Graphics for more information about support for newly-released devices.

The LeonardoSpectrum software also supports the FLEX 8000 and MAX 9000 legacy devices that are supported only in the Altera MAX+PLUS® II software, as well as ACEX® 1K, APEX™ II, APEX 20K, APEX 20KC, APEX 20KE, FLEX® 10K, and FLEX 6000 legacy devices that are supported by the Quartus II software version 9.0 and earlier.

Design Flow

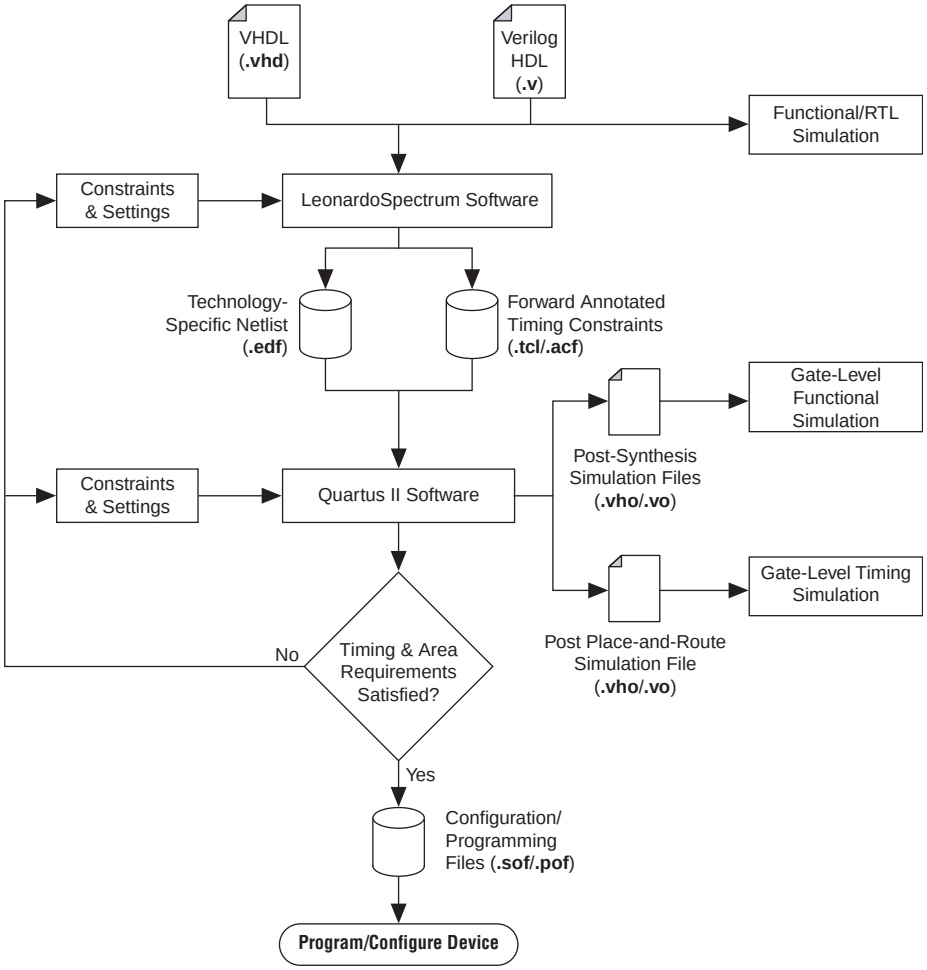
The following are basic steps in a LeonardoSpectrum-Quartus II design flow:

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 on page 12–3 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 on page 12–3, 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).
.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.
.acf	Assignment and Configurations file for backward compatibility with the MAX+PLUS II software. For devices supported by the MAX+PLUS II software, the MAX+PLUS II assignments are imported from the MAX+PLUS II **.acf** file.

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 LeonardoSpectrum software does not generate a Synopsys Design Constraint (SDC) format file for the TimeQuest Timing Analyzer. If you use TimeQuest, you must convert your timing constraints to SDC format for use in the Quartus II software. Altera recommends using the advanced Precision RTL Synthesis software for new designs in new device families, instead of using the LeonardoSpectrum 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 FlowTabs provide additional options, and some FlowTabs include multiple PowerTabs (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 that 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 PowerTab" are set in the **Constraints** FlowTab. In this tab, there are PowerTabs at the bottom, such as **Global** and **Clock**, for setting various constraints.

Global PowerTab

The **Global** tab is the default PowerTab in the **Constraints** FlowTab 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 the **Global** tab reflects the settings you have made.

Clock PowerTab

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** FlowTab. 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 PowerTabs

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** FlowTab. Then make the setting for that pin as described in the following three paragraphs.

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 to 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** FlowTab. 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** FlowTab. 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** FlowTab. The **Map I/O Registers** option applies to Altera FPGAs containing I/O cells (IOC) or I/O elements (IOE). 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 LeonardoSpectrum 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 on page 12–7 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 **.edf** netlist file readable as an input file in the Quartus II software for place-and-route. Select the **.edf** file option name in the **Output** FlowTab. The **.edf** file is also generated if the **Auto** option is turned on in the **Output** FlowTab.

Including Design Files for Black Boxed Modules

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, if the design has black boxed megafunctions.

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** FlowTab.

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 **.edf** netlist files and Tcl scripts 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` 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**.

The LeonardoSpectrum software does not generate a Synopsys Design Constraint (SDC) format file for the TimeQuest Timing Analyzer. If you use TimeQuest, you must convert your timing constraints to SDC format for use in the Quartus II software. Altera recommends using the advanced Precision RTL Synthesis software for new designs in new device families, instead of using the LeonardoSpectrum software.

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** FlowTab 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, refer to the respective megafunction user guide on the Altera IP and Megafunction page.

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

The Stratix series and the Cyclone series support the RAM primitive `altsyncram` with a minimum RAM size of 2 bits, and a minimum RAM address width of 1 bit.

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** FlowTabs, 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** FlowTabs, 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 12–3, 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 12–3. 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 12–3:

(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 by using the `extract_mac` attribute. Setting this attribute for a module affects only the multipliers inside that module. Use the following command:

```
//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 in the following example:

```
//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 12–4 shows the supported values for the dedicated_mult attribute.

Table 12–4. Values for the dedicated_mult Attribute

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)*
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 12–4:

(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 design flow with LogicLock™ constraints enables 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 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** FlowTab.

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 .edf Files

The first stage of a hierarchical design flow is to generate multiple **.edf** files, which allows you to take advantage of the incremental compilation flows in the Quartus II software. If the whole design is in one **.edf** file, changes in one block affect other blocks because of possible node name changes. You can generate multiple **.edf** 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 **.edf** files with one of these methods, you must create the appropriate Quartus II project(s) to place-and-route the design.

Generating Multiple .edf Files Using the LogicLock Option

This section describes how to generate multiple **.edf** 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** FlowTab instead of the **Quick Setup** tab.
2. Set the target technology and speed grade for the device on the **Technology** FlowTab.
3. Open the input source files on the **Input** FlowTab.
4. Click **Read** on the **Input** FlowTab to read the source files but not begin optimization.
5. Select the **Module** PowerTab located at the bottom of the **Constraints** FlowTab.
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** FlowTab. 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 is 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 FlowTab, or click **Run Flow**.

Synthesis creates an **.edf** file for each module that has a LogicLock assignment in the **Constraints** FlowTab. You can now use these files with the incremental compilation flows in the Quartus II software.

You might occasionally see multiple **.edf** 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 **.edf** file, and each **.edf** file has a LogicLock assignment to the same LogicLock region. When you import the **.edf** files to the Quartus II software, the **.edf** 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 **.edf** file.

Creating a Quartus II Project for Multiple .edf 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 **.edf** 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** FlowTab 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 **.edf** file and corresponding **.tcl** file into the Quartus II software:

- Use the **.tcl** file that is created for each **.edf** 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 **.edf** 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 **.edf** 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 .edf Files Using Black Boxes

This section describes how to manually generate multiple **.edf** 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 **.edf** files in the LeonardoSpectrum software, create a separate project for each module and top-level design that you want to maintain as a separate **.edf** 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** FlowTab.

- Read the HDL files for the modules. Modules may include black box instantiations of lower-level modules that are also maintained as separate **.edf** files.
- Add constraints.
- Turn off **Add I/O Pads** on the **Optimize** FlowTab.

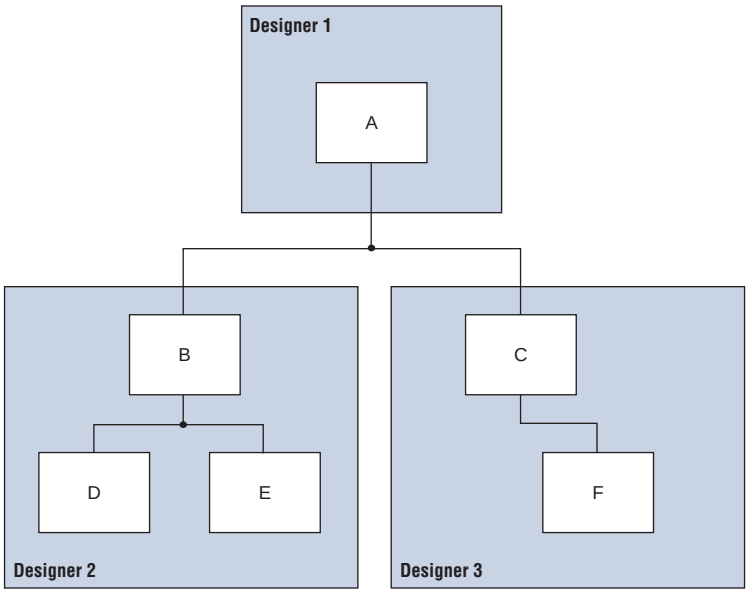
For the top-level design:

- 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** FlowTab.
- 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 **.edf** files, perform the following steps:

1. Generate an **.edf** file for module C. Use **C.v** and **F.v** as the source files.
2. Generate an **.edf** file for module B. Use **B.v**, **D.v**, and **E.v** as the source files.
3. Generate a top-level **.edf** 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 on page 12–23 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 **.edf** 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 .edf Files

The LeonardoSpectrum software creates a **.tcl** file for each **.edf** 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 **.edf** file and corresponding **.tcl** file into the Quartus II software:

- Use the **.tcl** file that is created for each **.edf** 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 **.edf** 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 **.edf** 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 **.edf** 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 **.edf** 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 **.edf** 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** FlowTab in the graphical user interface.

Example 12–8 on page 12–26 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 12–25, 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 **.edf** files, you can make multiple **.edf** files for use with the Quartus II software from a single LeonardoSpectrum software project.

Conclusion

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 with the LeonardoSpectrum software. For the best results with new designs in new device families, Altera recommends migrating to the advanced Mentor Graphics Precision RTL Synthesis software.

Document Revision History

Table 12–5 shows the revision history of this chapter.

Table 12–5. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Updated supported devices ■ Removed Referenced Documents section ■ Minor updates for the version 10.0 release.
November 2009	9.1.0	■ Minor updates for the Quartus II software version 9.1 release. ■ Removed Table 12–3, Inferring RAM Summary.
March 2009	9.0.0	■ No change to content. ■ Chapter 12 was previously Chapter 11 in software release 8.1.
November 2008	8.1.0	■ Changed to 8-1/2” x 11” page size. ■ Updated Table 12–3.
May 2008	8.0.0	Updated date and part number and added hypertext links.

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

Take an online survey to provide feedback about this chapter.

13. Analyzing Designs with Quartus II Netlist Viewers

QII51013-10.0.0

This chapter describes how you can use the Quartus® II netlist viewers to analyze and debug your designs. As FPGA designs grow in size and complexity, the ability to analyze, debug, optimize, and constrain your design is 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, and constraint entry processes.

This chapter contains the following sections:

- "Introduction to the User Interface" on page 13–6
- "Navigating the Schematic View" on page 13–21
- "Filtering in the Schematic View" on page 13–32
- "Probing to a Source Design File and Other Quartus II Windows" on page 13–38
- "Probing to the Netlist Viewers from Other Quartus II Windows" on page 13–39
- "Viewing a Timing Path" on page 13–40
- "Other Features in the Schematic Viewer" on page 13–42

The first section in this chapter, "When to Use the Netlist Viewers: Analyzing Design Problems", describes examples of using the netlist viewers to analyze your design at various stages of the design cycle. The sections following this section provide an introduction to the Quartus II design flow using the 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 in the Quartus II software
- How to view a timing path from the Timing Analyzer report

When to Use the Netlist Viewers: Analyzing Design Problems

You can use the netlist viewers to analyze and debug your design. 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 the netlist viewers display your design:

- "Quartus II Design Flow with the Netlist Viewers" on page 13–2
- "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 necessary 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 view state machine transitions and transition equations with the State Machine Viewer. Viewing your design helps 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 netlist 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 the netlist 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 your design. You can trace back from a point of interest to find the source of the signal and ensure the connections are as expected.

The Technology Map Viewer can 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.

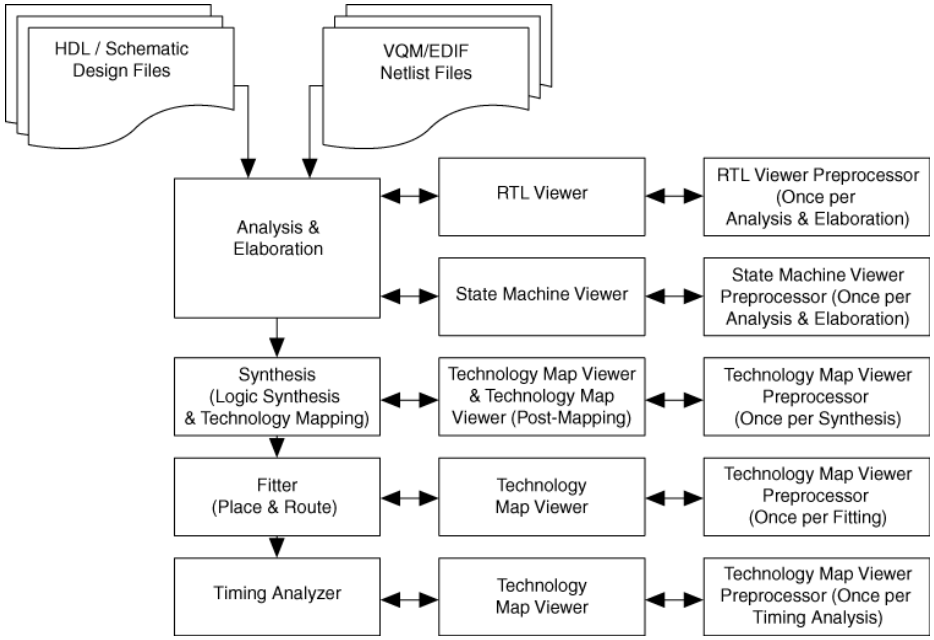
You can use the RTL Viewer, State Machine Viewer, and Technology Map Viewer in many other ways throughout the design, debugging, and optimization stages. Viewing the design netlist is a powerful way to analyze design problems. This chapter shows you how to use the various features of the netlist viewers to increase your productivity when analyzing a design.

Quartus II Design Flow with the Netlist Viewers

When you first open one of the netlist viewers after compiling the design, a preprocessor stage runs automatically before the netlist viewer opens. If you close the netlist viewer and open it again later without recompiling the design, the netlist 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 netlist viewer, and before the netlist 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 netlist viewers without first compiling the design with the appropriate minimum compilation stage, the netlist viewer does not appear. Instead, the Quartus II software issues an error message instructing you to run the necessary compilation stage and restart the netlist viewer.

The netlist 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 netlist viewer cannot be displayed; in this case, the Quartus II software issues an error message when you try to open the netlist viewer.

If the netlist viewer is open when you start a new compilation, the netlist viewer closes automatically. You must open the netlist 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 in 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. For a flow diagram, refer to Figure 13–1.

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 with 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. For more information, refer to "Viewing Contents of Atom Primitives" on page 13–22.

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 changes, 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 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 (for more information, refer to "Viewing a Timing Path" on page 13–40). For a flow diagram, refer to Figure 13–1 on page 13–3.

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 and Technology Map Viewer each consist of three main parts:

- The schematic view
- The **Netlist Navigator** pane
- The **Find** pane

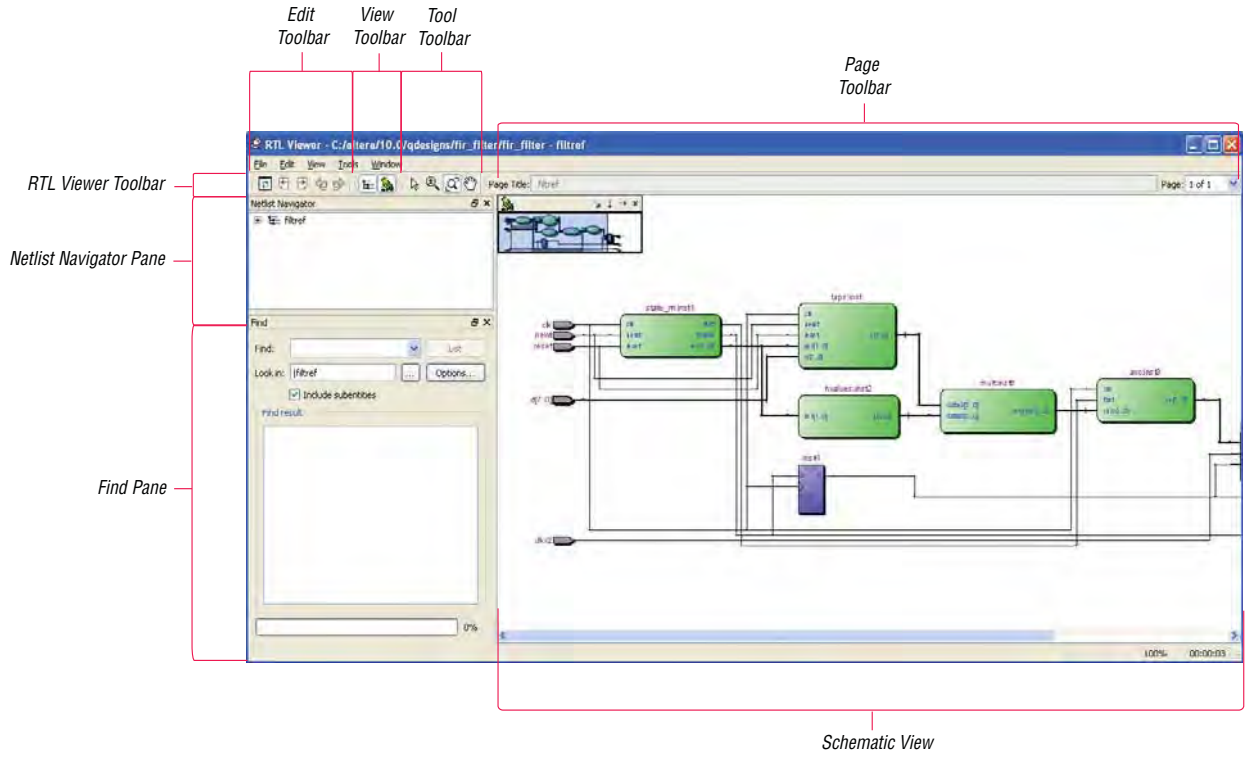
The schematic view displays a graphical representation of the internal structure of your design; the **Netlist Navigator** pane displays a representation of the project hierarchy; and the **Find** pane allows you to find and locate specific design elements in the schematic view.

Figure 13–2 shows the RTL Viewer and indicates these three parts, along with other elements of the user interface. Both the netlist viewers also contain a toolbar that provides tools to use in the schematic view.

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 netlist 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.

Figure 13–2 shows the schematic view and the **Netlist Navigator** pane of the RTL Viewer.

Figure 13–2. RTL Viewer



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.

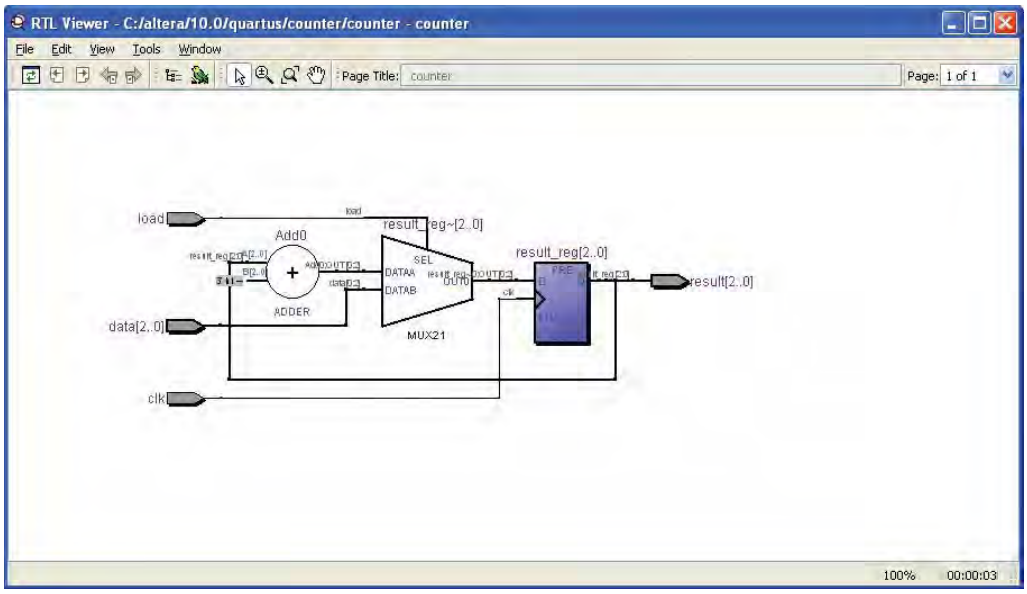
Schematic Symbols

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

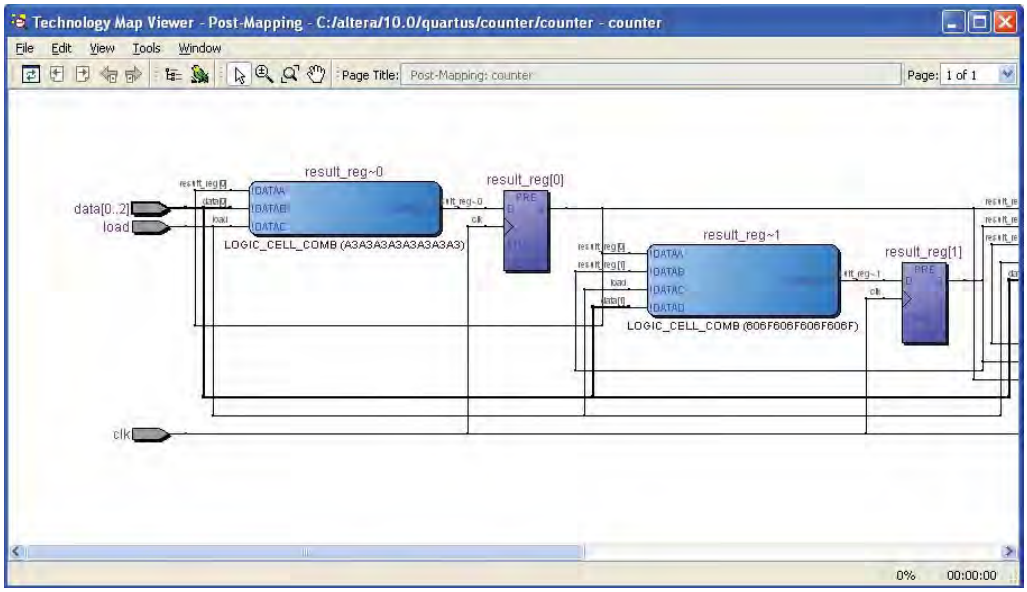


Table 13–1 lists and describes the primitives and basic symbols that you can display in the schematic view of the RTL Viewer and Technology Map Viewer. Table 13–3 on page 13–12 lists and describes the additional higher-level operator symbols used in the RTL Viewer schematic view.

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)

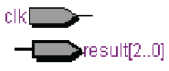
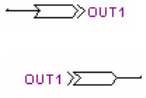
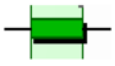
Symbol	Description
I/O Ports 	An input, output, or bidirectional port in the current level of hierarchy. A device input, output, or bidirectional pin when viewing the top-level hierarchy. The symbol can also represent a bus. Only one wire is shown connected to the bidirectional symbol, representing the 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 13–29). 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 13–30).
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.

Table 13–1. Symbols in the Schematic View (Part 2 of 3)

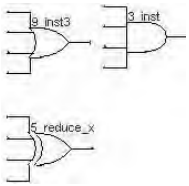
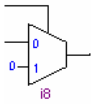
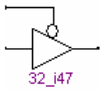
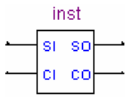
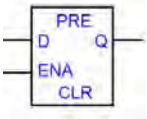
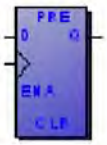
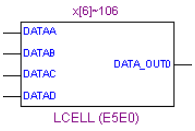
Symbol	Description
OR, AND, XOR Gates 	An OR, AND, or XOR gate primitive (the number of ports can vary). A small circle (bubble symbol) on an input or output port indicates the port is inverted.
MULTIPLEXER 	A multiplexer primitive with a selector port that selects between port `0` and port `1`. A multiplexer 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 – latch enable input ■ Q – data output ■ PRE – preset ■ CLR – clear
DFFE/DFFEA/DFFEAS 	A DFFE (data flipflop with clock 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 synchronous and asynchronous load), which has ASDATA as the secondary data port
Atom Primitive	An atom primitive. The symbol displays the atom name, the port names, and the atom type. The blue shading indicates an atom primitive for which you can view the internal details. For more information, refer to "Viewing Contents of Atom Primitives" on page 13–22.
Other Primitive 	Any primitive that does not fall into the previous categories. 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)

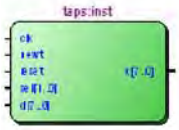
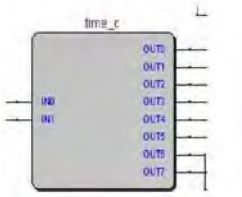
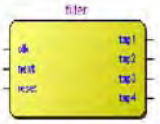
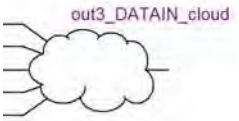
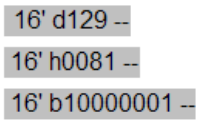
Symbol	Description
Instance 	An instance in the design that does not correspond to a primitive or operator (generally a user-defined hierarchy block). The symbol displays the port name and the instance name. For more information about opening the schematic for the lower-level hierarchy, refer to "Traversing and Viewing the Design Hierarchy" on page 13–21.
Encrypted Instance 	A user-defined encrypted instance in the design. 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. For more information, refer to "State Machine Viewer" on page 13–15.
RAM	A synchronous memory instance with registered inputs and optionally registered outputs. The symbol shows the device family and the type of memory block. This figure shows a true dual-port memory block in a Stratix M-RAM block.
Logic Cloud 	A combinational logic cloud in the design. For more information, refer to "Grouping Combinational Logic into Logic Clouds" on page 13–25.
Constant 	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 13–27.

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

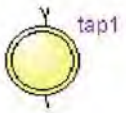
Symbol	Description
State Node 	The node representing a state in a finite state machine. State transitions are indicated with arcs between state nodes. The double circle border indicates the state connects to logic outside the state machine, and a single circle border indicates the state node does not 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)

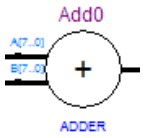
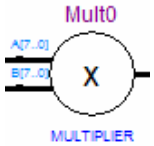
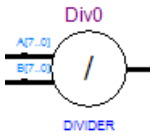
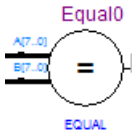
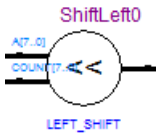
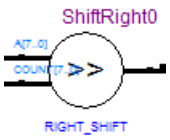
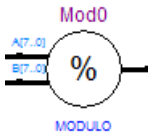
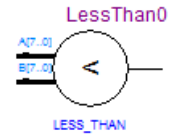
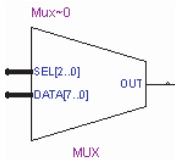
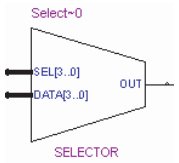
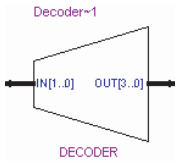
Symbol	Description
	An adder operator: `OUT = A + B`
	A multiplier operator: `OUT = A ¥ B`
	A divider operator: `OUT = A / B`
	Equals
	A left shift operator: `OUT = (A << COUNT)`

Table 13–3. Operator Symbols in the RTL Viewer Schematic View (Part 2 of 2)

Symbol	Description
	A right shift operator: `OUT = (A >> COUNT)`
	A modulo operator: `OUT = (A % B)`
	A less than comparator: `OUT = (A < B : A > B)`
	A multiplexer: `OUT = DATA [SEL]` The data range size is $2^{\text{sel range size}}$
	A selector: A multiplexer with one-hot select input and more than two input signals
	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 netlist viewer toolbar (this tool is enabled by default). Click 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, left-click at one corner of the area you want to select and drag the mouse to the diagonally opposite corner. By default, 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**. Under **Net Selection**, turn on the **Select entire net when segment is selected** option.

If you enable the **Enable auto hierarchy expansion** option, items selected in the schematic view are automatically selected in the **Netlist Navigator** pane (for more information about how to enable the auto hierarchy expansion option, refer to "Netlist Navigator Pane" on page 13–14). The folder then expands automatically if it is required to show the selected entry; however, the folder does not collapse automatically when you are not using or you have deselected the entries.

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 are also 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 or wheel button to move and pan in the schematic view. Click the middle-mouse or wheel button once to enable the feature. Move the mouse or scroll the wheel to move around the schematic view. Click the middle-mouse or wheel button again to turn the feature off.

Netlist Navigator Pane

The **Netlist Navigator** pane displays the entire netlist in a tree format based on the hierarchical levels of the design. In each level, similar elements are grouped into subcategories. You can use the **Netlist Navigator** pane to traverse through the design hierarchy to view the logic schematic for each level. You can also select an element in the netlist navigator to highlight in the schematic view.

Nodes inside atom primitives are not listed in the **Netlist Navigator** pane.

For each module in the design hierarchy, the **Netlist Navigator** pane displays the applicable elements listed in Table 13–4. Click the "+" icon to expand an element.

Table 13–4. Netlist Navigator Pane Elements

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.
Primitives	Low-level nodes that cannot be expanded to any lower hierarchy level. These primitives 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.

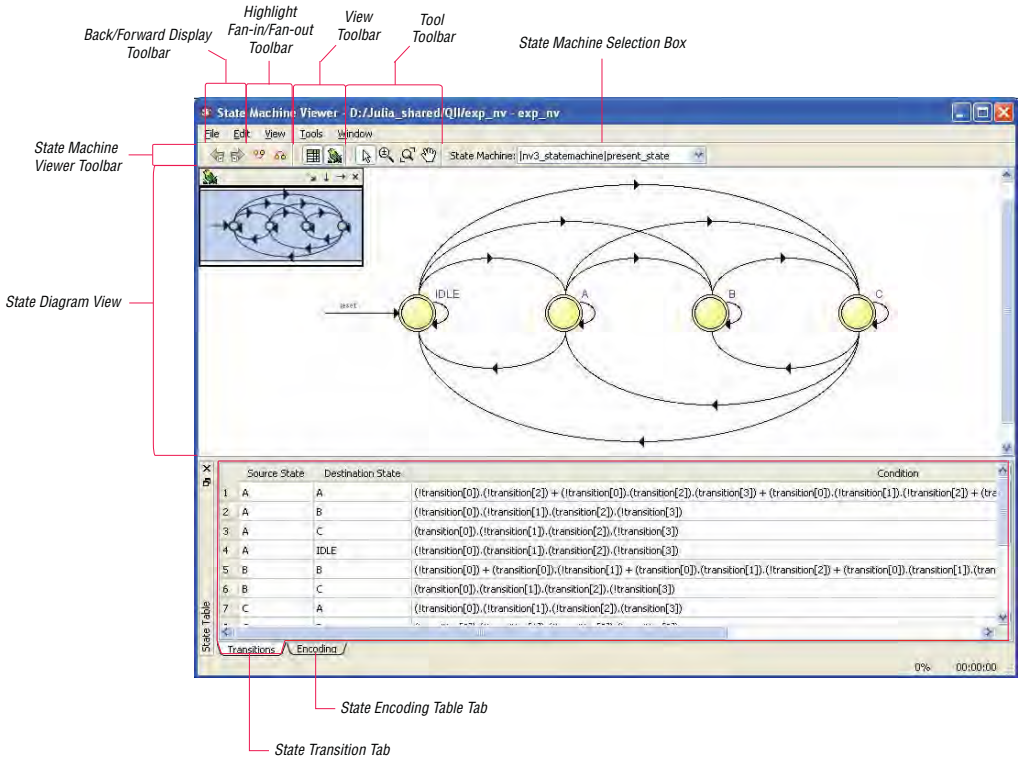
State Machine Viewer

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 a state machine instance in the RTL Viewer
- Right-click 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.

Figure 13–5. The State Machine Viewer



State Diagram View

The state diagram view appears at the top of the State Machine Viewer. 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, and 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 displays the condition equation for each state transition. Each row in the table represents a transition (each arc in the state diagram view). The table has the following 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 to make it easier to see the relevant part of the text, right-click the column and click **Align Right**. To revert 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 displays encoding information for each state transition.

To view state encoding information in the State Machine Viewer, you must synthesize your design with the **Start Analysis & Synthesis** command. If you have only elaborated your design with the **Start Analysis & Elaboration** command, 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 node or transition arc in the state diagram view, the matching state node or equation conditions in the state transition table are highlighted; conversely, when you select a state node or equation condition in the state transition table, the corresponding state node or transition arc is 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 necessary state machine.

Global Options

The **Options** dialog box allows you to customize the following settings:

- Display Settings
- Colors
- Fonts
- Tracing
- Customize View
- Shortcut Commands

Display Settings

If you want to customize the display settings for your preferred viewing, you can direct the RTL Viewer and Technology Map Viewer to adjust the settings in the schematic view. To adjust the display settings, on the Tools menu, click **Options**. Under **Display Settings**, you can select the options to customize your display.

You can divide the schematic representation of a large design into multiple pages. Dividing the display of the schematic into multiple pages does not affect your design, and only controls the number of elements per page. Under **Display Settings**, the **Nodes per page** option allows you to specify the number of nodes displayed per page. The default value is 50 nodes; however, you can view from one to 1,000 nodes per page. The **Ports per page** option allows you to specify the number of ports (or pins) displayed per page. The default value is 1,000 ports (or pins); the range is 1 to 2,000 ports (or pins). The netlist viewers partition your design into a new page if either the number of nodes or the number of ports exceeds the limit you specified. Occasionally, the number of ports displayed on the page might exceed the limit you specified, depending on the configuration of nodes on the page. If you turned on the **Display boundary around hierarchy levels** option and the total number of nodes or ports in the hierarchy exceeds the value you specified for the **Nodes per page** or **Ports per page** options, the netlist viewers display the boundary as a hierarchy port connector (refer to Table 13–1 on page 13–9).

To display net names in your schematic, turn on the **Show Net Name** option. If you turn on this option, the schematic view refreshes automatically to display the net names. To show node names in the schematic view, turn on the **Show node name** option. To change the value formatting, select the necessary format in the **Constant signal format** list.

To view highlighting around the design element in range of the mouse pointer, turn on the **Enable rollover** option. To more easily navigate through your design hierarchy, you can direct the netlist viewers to automatically expand the hierarchy list in **Netlist Navigator** pane and highlight the design element you selected in the schematic view. This automatic expansion and selection of corresponding design elements is useful when you have a complex schematic that spans multiple display pages in the netlist viewers. To use the automatic expansion and selection feature, turn on the **Enable auto hierarchy expansion** option.

To enable the radial menu, turn on the **Enable radial menu** option. The radial menu is an octagonal menu with eight commands from which you can choose. This menu provides a quick way to perform any of the commands with a single click whenever you are in the schematic view.

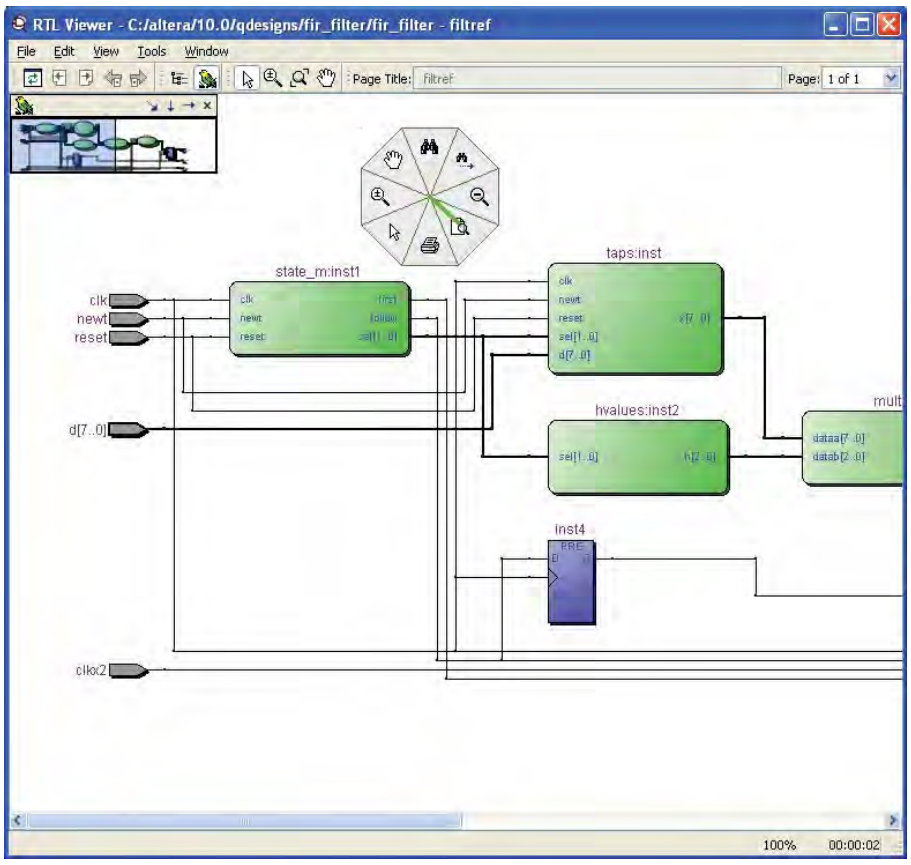
To open the radial menu, right-click and hold anywhere in the schematic view and wait for the menu to appear. By default, the menu appears after 0.2 seconds. The radial menu appears with the mouse pointer always at the center point. The center point of the menu is a non-trigger boundary in which no command is started.

To invoke the necessary command, hold down the right mouse button, drag the mouse onto the command, and then press the left 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 center point and release the mouse button.

To change the delay time before the radial menu appears, select the necessary interval time in the drop-down list for **Delay showing radial menu for**. The default delay is 0.2 seconds.

Figure 13–6 shows the radial menu.

Figure 13–6. Radial Menu



Tracing

If you want to filter information from the schematic view of your design to isolate specific design elements for further inspection, or if you want to expand specific design elements, you can direct the RTL Viewer and the Technology Map Viewer to adjust the elements the viewers show in the schematic view. To adjust the filtering and expansion settings in the RTL Viewer and Technology Map Viewer, on the Tools menu, click **Options**. Under **Tracing**, you can select the options to control filtering and expansion settings.

For all filtering commands, the netlist viewers stop tracing through the netlist when they reach one of the following objects:

- A pin
- A specified number of filtering levels, counting from the selected node or port
- A register

To specify the number of filtering levels, set the **Number of filtering levels** option to specify the number of levels to expand. You can specify a value from one to 100.

To enable the **Stop filtering at register** option, turn on the **Stop filtering at register** option. You can filter across hierarchies when you turn on the **Filter across hierarchy** option.

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 **Show all connections between nodes** option.

To set the amount of logic you want to expand, set the **Number of expansion levels** option to specify the number of levels to expand. You can specify a range from one to 100 levels. You can also set the **Stop expanding at register** option to specify whether netlist expansion should stop when a register is reached.

Customize View

If you want to customize the schematic display for better viewing and to speed up your debugging process, you can direct the RTL Viewer and the Technology Map Viewer to remove fan-out free nodes, show simplify logic, group or ungroup related nodes, and group combinational logic into a logic cloud. To adjust the options that control the schematic display in the RTL Viewer and the Technology Map Viewer, on the Tools menu, click **Options**. Under **Customize View**, you can select the options to customize your view. These options are also 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**.

When you change settings, 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 the **Remove registers without fan-out** option. 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 the **Show simplified logic** option.

To group all related nodes into a single node, turn on the **Group all related nodes** option. You can manually group or ungroup any nodes by right-clicking the selected nodes in the schematic and selecting **Group Related Nodes** or **Ungroup Selected Nodes**. To group combinational logic into logic clouds, turn on the **Group combinational logic into logic cloud** option.

Customize Logic and Customize Group options are available for the RTL Viewer, whereas only the Customize Group option is available for the Technology Map Viewer.

Shortcut Commands

You can choose eight commands to appear on the radial menu from a list of 18 available commands. To customize the command list on the menu, first launch the RTL Viewer or the Technology Map Viewer. On the Tools menu, select **Options**. On the **Shortcut Commands** category list, drag and drop the icon under **Shortcut buttons** into any region under **Shortcut commands popup**. You can click the icon under **Shortcut buttons** to see its description.

Navigating the Schematic View

This section describes methods to navigate through the pages and hierarchy levels in the schematic view of the RTL Viewer and the Technology Map Viewer.

Traversing and Viewing the Design Hierarchy

You can open different hierarchy levels in the schematic view from the **Netlist Navigator** pane (refer to "Netlist Navigator Pane" on page 13–14), or the **Hierarchy Up** and **Hierarchy Down** commands (Shortcut menu) in the schematic view.

Use the **Hierarchy Down** command to go down in 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 13–7). 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 13–7. 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 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 in the Current Schematic

You can use the **Display Content** and **Hide Content** (Shortcut menu) commands to show or hide a lower hierarchy level for a specific instance in 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**.

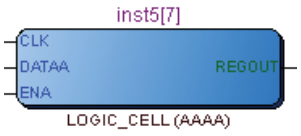
Viewing Contents of Atom Primitives

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 Arria® GX, Cyclone® series, MAX® II, and Stratix series of devices, you can view LUTs, registers, and logic gates. For I/O atoms in the Arria GX, Cyclone series, HardCopy® IV, and Stratix series of 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 Arria GX, Cyclone series, and Stratix series of devices. You can view the implementation of DSP blocks only in Arria GX and Stratix series of devices.

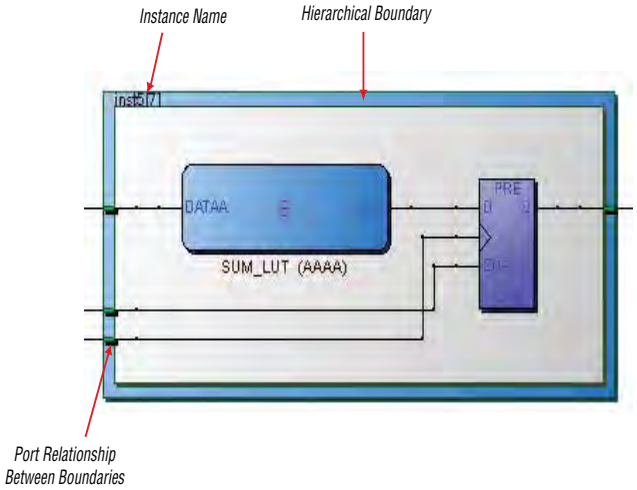
If you can view the contents of an atom instance, the internal contents are shown in blue in the schematic view (Figure 13–8).

Figure 13–8. Instance That Can Be Expanded to View Internal Contents



To view the contents of one or more atom primitive instances, select the necessary atom instances. Right-click a selected instance and click **Display Content**. You can also double-click the necessary atom instance to view the contents. Figure 13–9 shows an expanded version of the instance in Figure 13–8.

Figure 13–9. Internal Contents of the Atom Instance in Figure 13–8.



To hide the contents (and revert to the compact format), select and right-click the atom instance or instances, and click **Hide Content**.

In the schematic view, the internal details in 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 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 a LUT by right-clicking the selected LUT and clicking **Properties**. This feature is supported for the Arria GX, Cyclone series, MAX II, and Stratix series of devices only. You can view the LUT representations in the following three tabs in the **Properties** dialog box:

- The **Schematic** tab (Figure 13–10) shows the equivalent gate representations of the LUT.
- The **Truth Table** tab (Figure 13–11) shows the truth table representations.
- The **Karnaugh Map** tab (Figure 13–12) shows the Karnaugh map representations of the LUT. The Karnaugh map supports up to 6 input LUTs.

For more information about the **Ports** tab, refer to "Viewing the Properties of Instances and Primitives" on page 13–23.

Figure 13–10. Schematic Tab

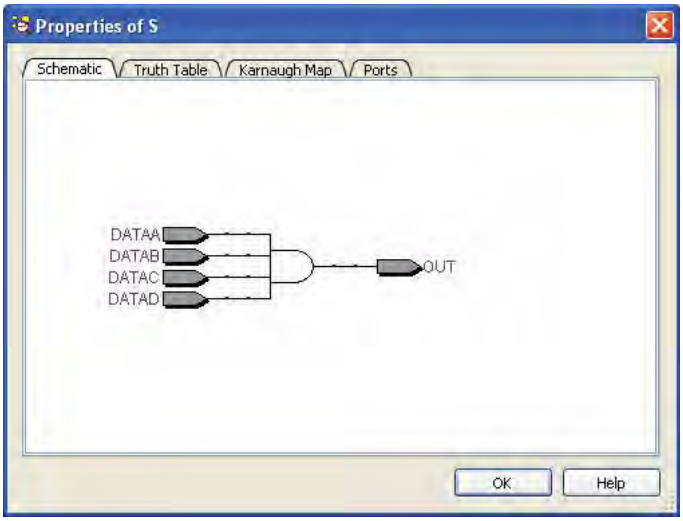


Figure 13–11. Truth Table Tab

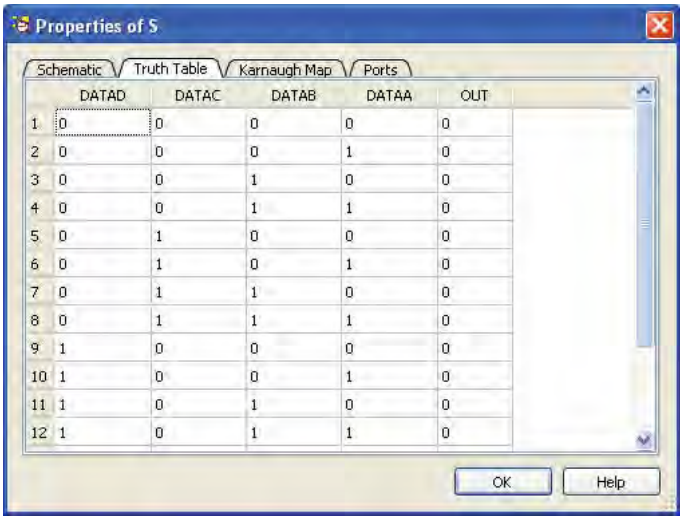
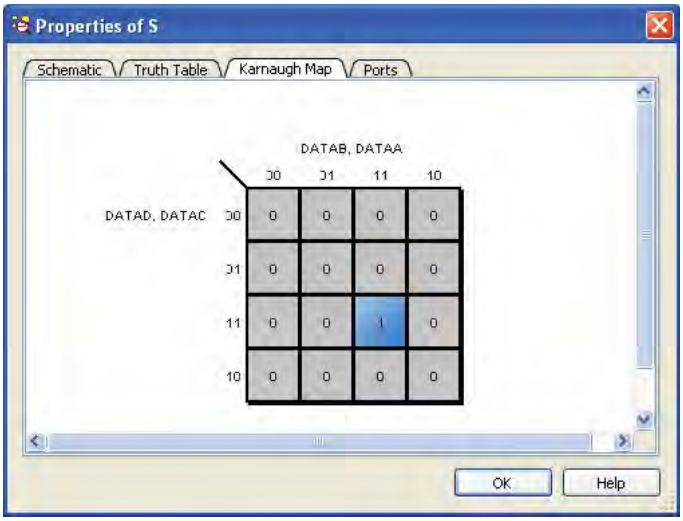


Figure 13–12. Karnaugh Map Tab



Grouping Combinational Logic into Logic Clouds

The following sections describe 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. For more information about how to group combinational logic clouds, refer to "Customize View" on page 13–20. Figure 13–13 and Figure 13–14 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 Cyclone II, HardCopy, and Stratix II devices only. For more information about how to group combinational logic clouds, refer to "Customize View" on page 13–20.

Figure 13–13. Schematic Before Combinational Logic Grouping

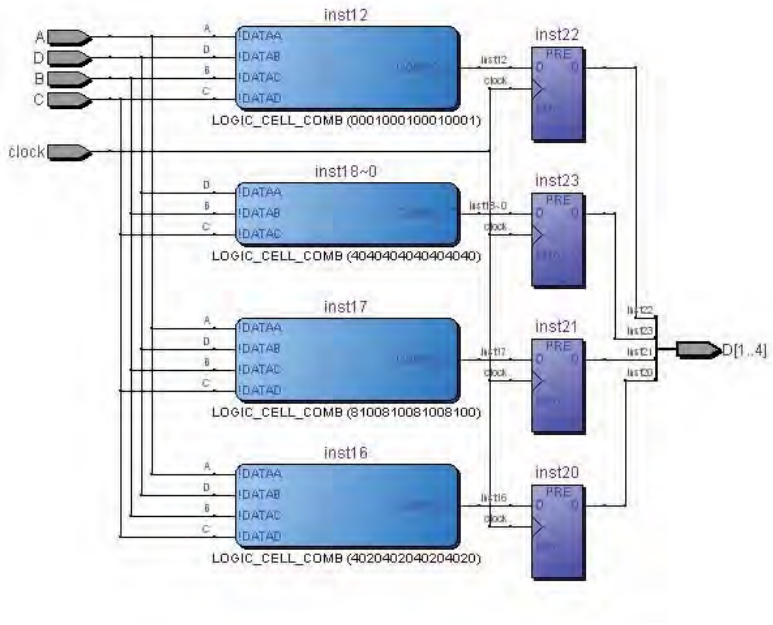
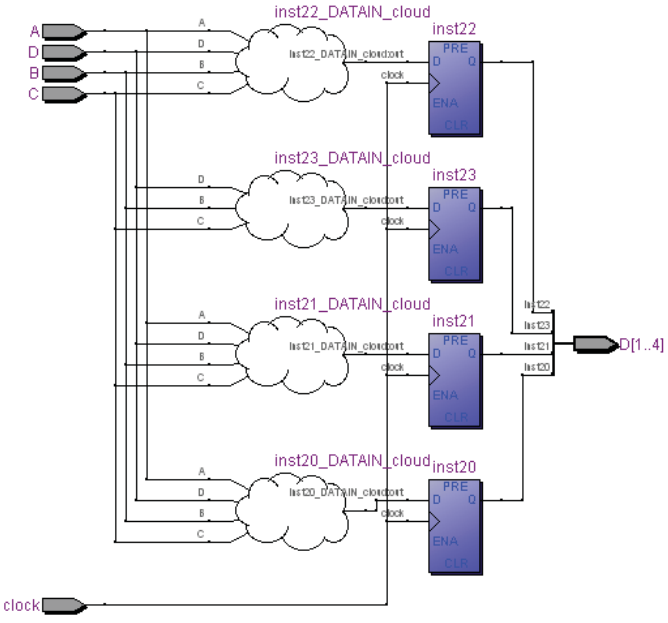


Figure 13–14. Schematic After Combinational Logic Grouping



Grouping and Ungrouping Logic Clouds

To group logic nodes into a logic cloud manually, right-click the selected node or input port and click **Group source logic into logic cloud**. To ungroup a logic cloud manually, right-click the selected logic cloud and click **Ungroup source logic from logic cloud**. You can also ungroup a logic cloud manually by double-clicking 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. For more information about changing the value formatting, refer to "Display Settings" on page 13–18.

Changing the format affects all constant signal values throughout the schematic. For descriptions of constant signal values in the schematic, refer to Table 13–3 on page 13–12.

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.

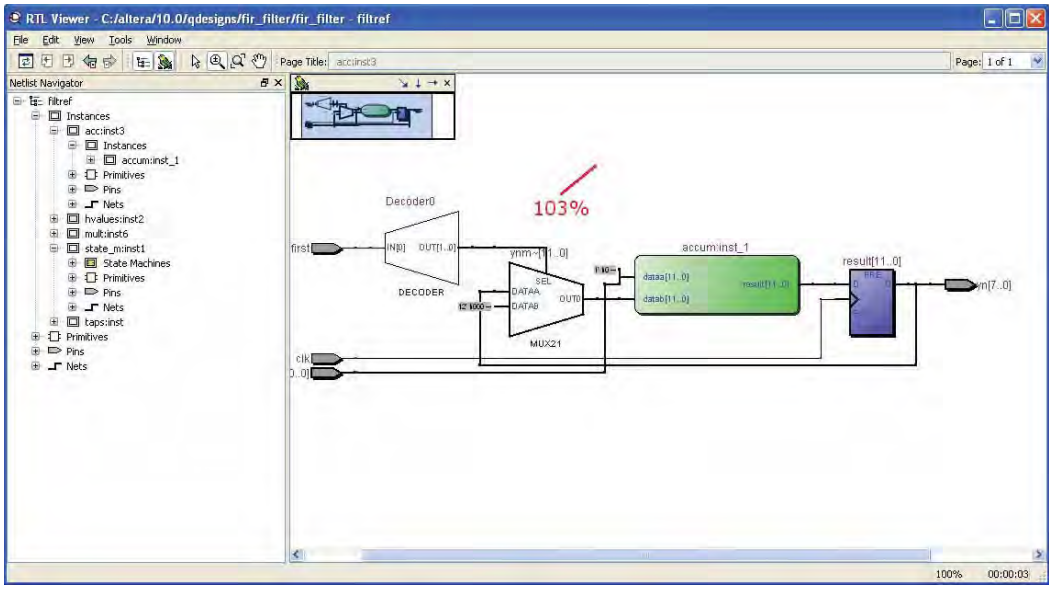
By default, the netlist viewer displays most pages sized to fit in the window. If the schematic page is very large, the schematic is displayed at the minimum zoom level, and the view is centered on the first node. Click **Zoom In** to view the image at a larger size, and click **Zoom Out** to view the 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).

The **Fit Selection in Window** command zooms in on the selected nodes in a schematic to fit in the window. Use the Selection Tool to select one or more nodes (instances, primitives, pins, and nets), then click **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 temporarily enlarge a portion of the schematic with the magnifying glass tool in the toolbar.

You can also use the Zoom Tool on the netlist 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 in 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 the necessary area and dragging the mouse to the right to zoom in or to the left to zoom out with the Zoom Tool. You will see a red line with the zoom percentage above it. The zoom percentage is proportional to the length of the green line (Figure 13–15). Release the mouse button at the necessary zoom percentage.

Figure 13–15. Dragging the Mouse Pointer to Change Zoom Percentage



By default, the netlist viewers maintain the zoom level when filtering on the schematic (refer to "Filtering in the Schematic View" on page 13–32). 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

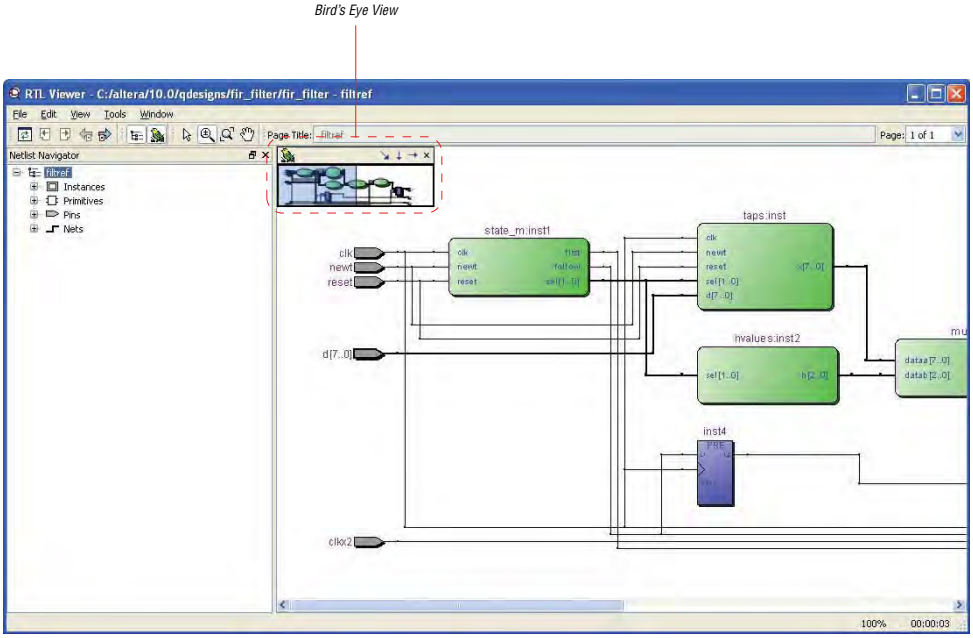
Viewing the entire schematic can be useful when debugging and tracing through a large netlist. The Quartus II software allows you to quickly navigate to a specific section of the schematic using the Bird's Eye View feature, which is available in the RTL Viewer and Technology Map Viewer.

The Bird's Eye View shows the current area of interest. Select the necessary area by clicking and dragging the indicator or right-clicking to form a rectangular box around the necessary 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 toolbar (Figure 13–16).

Figure 13–16. Bird's Eye View Icon

Figure 13–17 shows the Bird's Eye View window in the schematic diagram.

Figure 13–17. Bird's Eye View Window

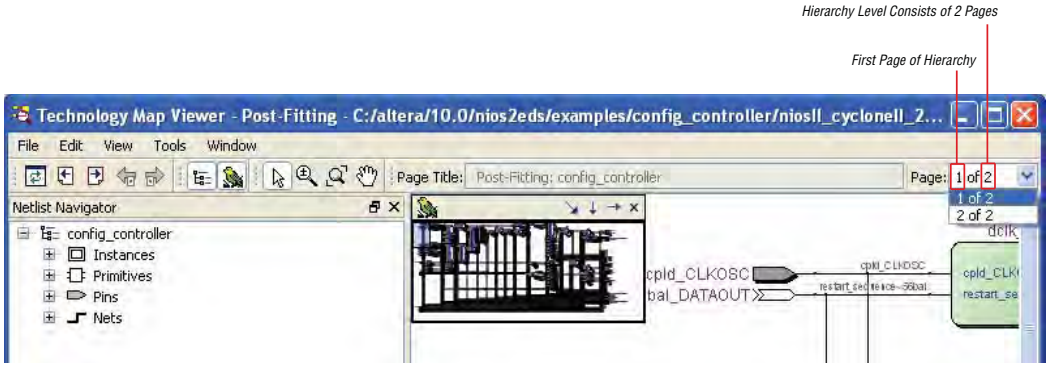


Partitioning the Schematic into Pages

For large design hierarchies, the RTL Viewer and Technology Map Viewer partition your netlist into multiple pages in the schematic view. For more information about controlling how much of the design is visible on each page, refer to "Display Settings" on page 13–18.

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 13–18.

Figure 13–18. 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 netlist viewer. To refresh the current page so that it displays the changed number of nodes or ports, click the **Refresh** button on 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 on the netlist 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 necessary page number. You can also go to a particular page by selecting the necessary page number from the pull-down list on the top right of the netlist viewer.

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 netlist viewer toolbar. To go to the next view, click **Forward** on the View menu, or click the **Forward** icon on the netlist 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

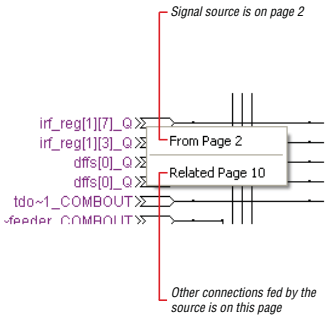
Input and output connectors indicate nodes that connect across pages of the same hierarchy. Right-click a connector to display a menu of commands that trace the net through the pages of the hierarchy.

After you right-click to follow a connector port, the netlist viewer opens a new page, which centers the view on the particular source or destination net using the same zoom factor as the previous page. To trace a specific net to the new page of the hierarchy, Altera recommends that you first select the necessary net, which highlights it in red, before you right-click to traverse pages.

Input Connectors

Figure 13–19 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.

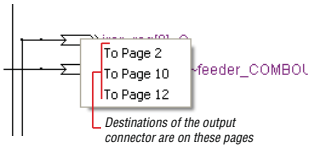
Figure 13–19. Input Connector Shortcut Menu



Output Connectors

Figure 13–20 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 13–20. Output Connector Shortcut Menu



Go to Net Driver

To locate the source of a particular net in the schematic view, right-click the net, point to **Go to Net Driver** and click **Current page**, **Current hierarchy**, or **Across hierarchies**. Table 13–6 lists the **Go to Net Driver** commands.

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 in 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 unfiltered, so no filtering results are kept.

Filtering in the Schematic View

Filtering allows you to filter out nodes and nets in your netlist to view only the logic elements of interest to you.

You can 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 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

To filter your netlist, 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 netlist 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, as described in "Moving Back and Forward Through Schematic Pages" on page 13–30.

When viewing a filtered netlist, clicking an item in the **Netlist Navigator** pane causes the schematic view to display an unfiltered view of the appropriate hierarchy level. You cannot use the **Netlist Navigator** pane 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 listed in Table 13–7 and shown in Figure 13–21.

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 13–21.
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 13–21, right-click the node or port, point to **Filter**, and click **Destinations**.

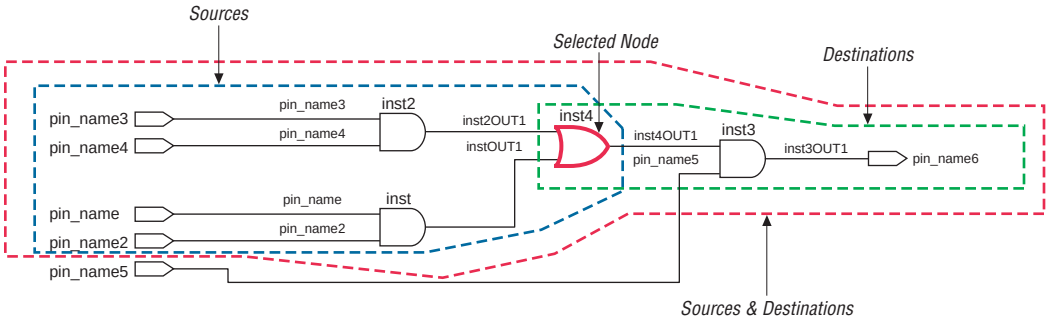
Table 13–8. Selected Objects Determine Filter Destinations Display

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 13–21.
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).

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 the sources and the destinations of the selected item. To select this option, right-click the necessary object, point to **Filter**, and click **Sources & Destinations**. Refer to the example in Figure 13–21.

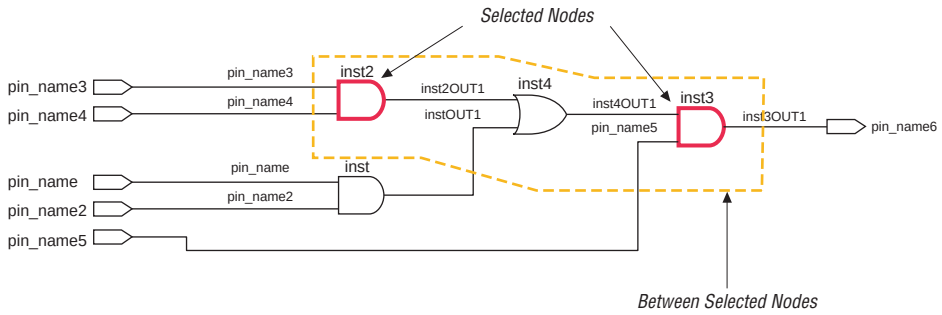
Figure 13–21. Sources, Destinations, and Sources and Destinations Filtering for inst4



Filter Between Selected Nodes Command

To show the nodes in the path between two or more selected nodes or hierarchy boxes, right-click the necessary object, 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 13–22.

Figure 13–22. 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 the necessary object, point to **Filter**, and click **Selected Nodes & Nets**. Figure 13–23 shows a schematic with several nodes selected.

Figure 13–23. Using Selected Nodes and Nets to Select Nodes

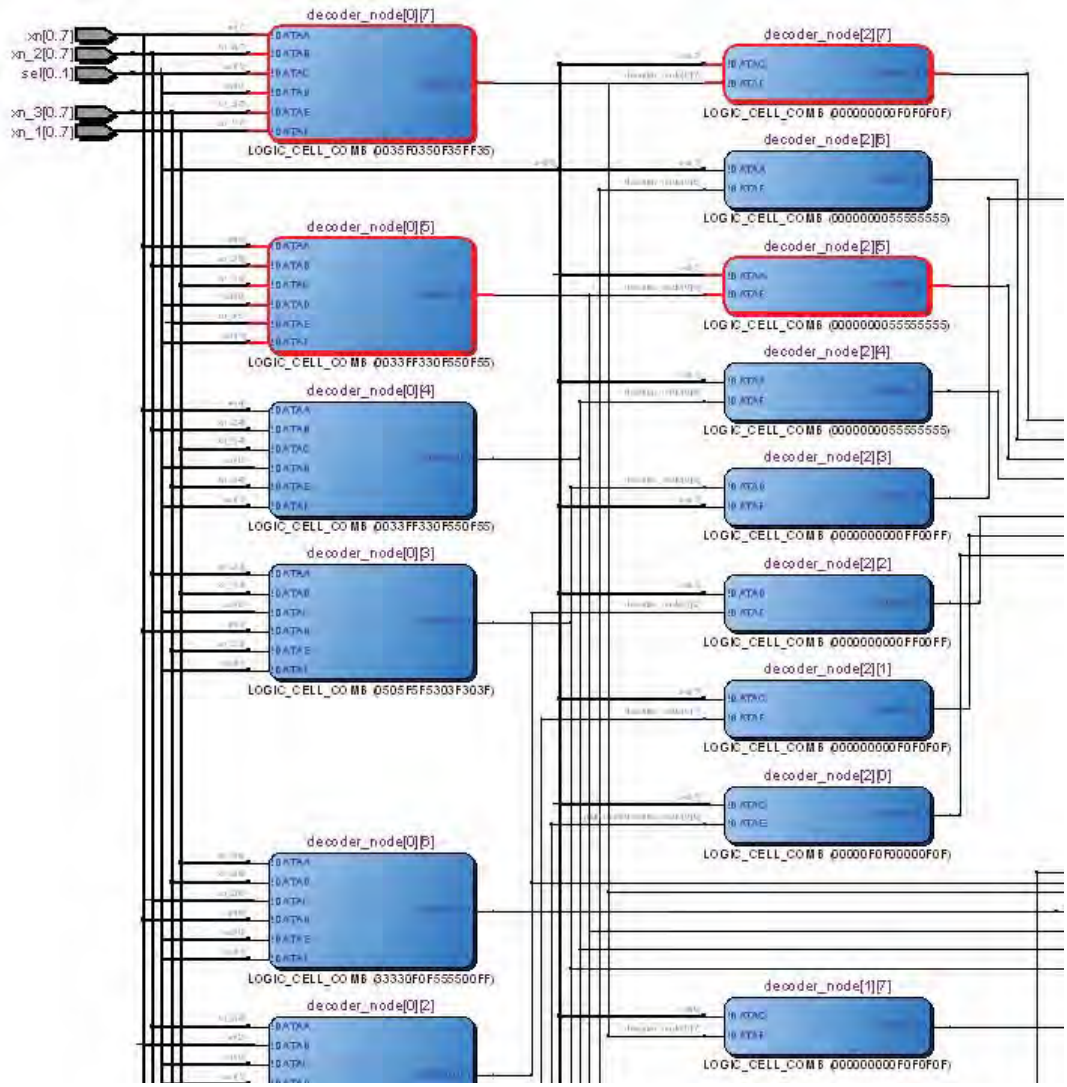
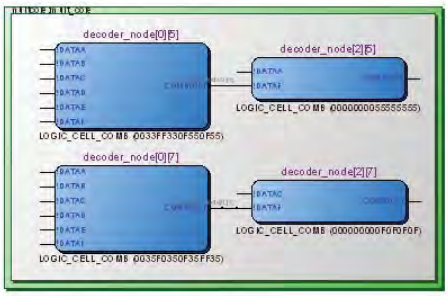


Figure 13–24 shows the schematic after filtering. If you select a net, the filtered page shows the immediate sources and destinations of the selected net.

Figure 13–24. Selected Nodes and Nets Filtering on Figure 13–23 Schematic



New Schematic Created After Applying Selected Nodes and Nets Filtering

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** tab of the **RTL/Technology Map Viewer Options** dialog box. Right-click in the schematic view and click **Viewer Options** to open the dialog box. For more information about filter command processing, refer to "Tracing" on page 13–19.

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 that appears on the border represents the port relationship between two different hierarchies (Figure 13–25 and Figure 13–26).

For more information about how to control filtering, refer to "Tracing" on page 13–19.

For more information about how to disable the box hierarchy display, refer to "Display Settings" on page 13–18.

Netlists of the same hierarchy 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 13–25 and Figure 13–26 show examples of filtering across hierarchical boundaries. Figure 13–25 shows a smaller example of an input port of the `taps` instance after the **Sources** filter is applied, in which the input port of the lower-level hierarchical block connects directly to an input pin of the design. The name of the instance appears in the green border and as a tooltip when you move your mouse pointer over the instance.

Figure 13–25. Filtering Across Hierarchical Boundaries

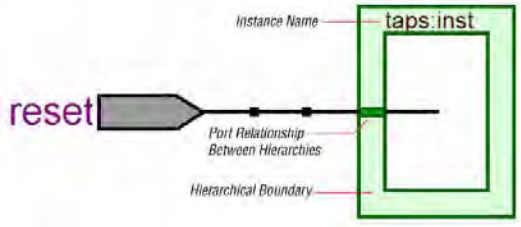
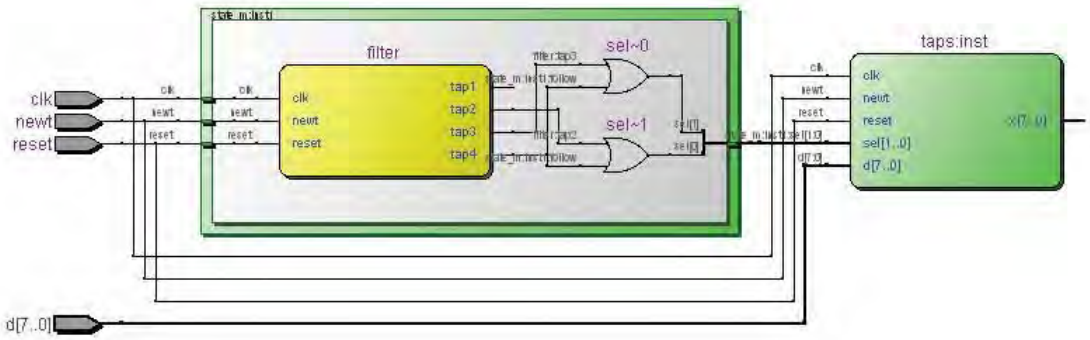


Figure 13–26 shows a larger example of an input port of an instance after the **Sources** filter is applied, in which the source comes from input pins that are fed through another level of hierarchy.

Figure 13–26. Filtering Across Hierarchical Boundaries



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. These options are also available under **Tracing** in the **Options** dialog box (refer to "Tracing" on page 13–19).

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 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 you want to expand was generated with the **Filter across hierarchy** option turned on (for details about this option, refer to "Filtering in the Schematic View" on page 13–32). 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 (for details about these schematics, refer to "Viewing a Timing Path" on page 13–40).

Reducing a Filtered Netlist

In some cases, removing logic from a filtered schematic or state diagram makes the schematic view easier to read and minimizes distracting logic in the schematic that you do not need to view.

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 a Source Design File and Other Quartus II Windows

The RTL Viewer, Technology Map Viewer, and State Machine Viewer allows you to cross-probe to the source design file and to various other windows in 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 netlist 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 netlist viewer in another window, right-click the items of interest in the schematic or state diagram, 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 an item 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 netlist 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 netlist 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, prompting 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 in 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 in 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 (**.stp**), right-click the selected node in the schematic diagram or in the netlist navigator, and then click **Add Node to SignalTap II Logic Analyzer**. If the node cannot be tapped, the option is unavailable.

Probing to the Netlist Viewers from Other Quartus II Windows

You can cross-probe to the RTL Viewer and Technology Map Viewer from other windows in the Quartus II software. You can select one or more nodes or nets in another window and locate them in one of the netlist 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 and 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 netlist 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 click this command, the netlist viewer opens, or is brought to the foreground if the netlist viewer is already open.

The first time the window opens after a compilation, the preprocessor stage runs before the netlist viewer opens.

The netlist 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 netlist 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 a report panel in the TimeQuest analyzer.

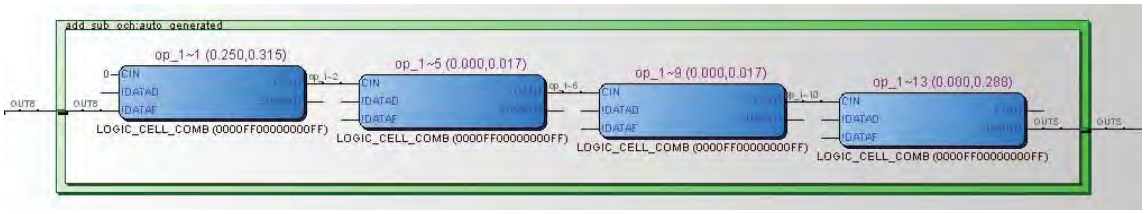
To take advantage of this feature, you must complete a full compilation of your design, including the timing analyzer stage. To see the timing results for your design, on the Processing menu, click **Compilation Report**. On the left side of the Compilation Report, select **Timing Analyzer** or **TimeQuest 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 analyzer report panels. To view a particular timing path in the Technology Map Viewer or RTL Viewer, right-click the appropriate row in the table, point to **Locate**, and click **Locate in Technology Map Viewer** or **Locate in RTL Viewer**.

To locate a path, on the **Tasks** pane, in the **Custom Reports** folder, double-click **Report Timing**. In the **Report Timing** dialog box, make any necessary settings, and then click the **Report Timing** button. After the TimeQuest analyzer generates the report, right-click on the node in the table and select **Locate Path**. In the Technology Map Viewer, the schematic page displays the nodes along the timing path with a summary of the total delay.

When you locate the timing path from the TimeQuest 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 13–42.

Figure 13–27 shows a portion of a timing path represented in the Technology Map Viewer.

Figure 13–27. 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 fitting 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, or 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 in 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 in 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 more information about understanding equations, refer to the Quartus II Help.

To copy tooltips into the clipboard for use in other applications, right-click the necessary node or netlist and click **Copy Tooltip**.

To turn off tooltips or change the duration of time that a tooltip appears in the view, in the main Quartus II window, select **Options**. In the **Tooltip Settings** pane, turn on the **Enable tooltips** option.

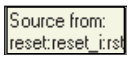
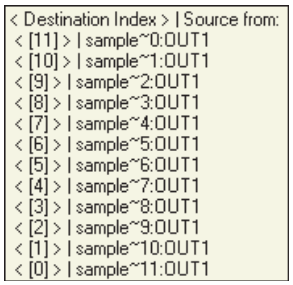
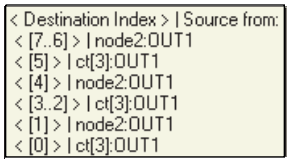
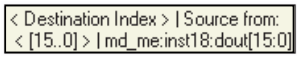
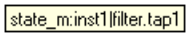
The **Show names in tooltip for** option specifies the number of seconds to display the names of assigned nodes and pins in a tooltip when the pointer is over the assigned nodes and pins. Selecting **Unlimited** displays the tooltip as long as the pointer remains over the node or pin. Selecting **0** 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 the display of tooltips. The default value is 1 second.

Table 13–9. Tooltip Information (Part 1 of 2)

Tooltip Format	Description	Example Tooltips
Instance	Format: <*instance name*>, <*instance type*> <*primitive type*>, <*number of primitives*>... <*primitive type*>, <*number of primitives*>	taps:inst, INST DFF 32 OPERATOR(SELECTOR) 8 OPERATOR(DECODER) 1
Atom Primitive	Format: <*instance name*>, <*primitive name*> (<*LUT Mask Value*>) {(r | c <*Register or Combinational equation*>)} ... An r (as in the first example) represents the equation for a register, and a c (as in the second example) represents the equation for combinational logic.	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
Primitive	Format:<*primitive name*>, <*primitive type*>	clocks:inst7|Mux~1, OPER (MUX) md_me:inst18|data[3..3], DFFE
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

Table 13–9. Tooltip Information (Part 2 of 2)

Tooltip Format	Description	Example Tooltips
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).	 *(1)*  *(2)*  *(3)* *(4)* *(5)*  *(6)*
State Machine Node	Format: <*node name*>	
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*>)	

Finding Design Elements in the Netlist Viewers

You can narrow the range of the search process by setting the following options in the **Find** pane:

- Click **Browse** in the **Find** pane to specify the hierarchy level of the search. In the **Select Hierarchy Level** dialog box, select the particular instance 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 the **List** button, a progress bar appears below the **Find** box.

All results that match the criteria you set are listed in a table. When you double-click an item in the table, the related node is highlighted in red in the schematic view.

Exporting and Copying a Schematic Image

You can export the schematic view of the RTL Viewer or Technology Map Viewer into various 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**), and 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 necessary 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 entire image or a portion of the image. To copy the entire image, right-click on the schematic, point to **Copy**, and then click **Full Image**. To copy a portion of the image, right-click on the schematic, point to **Copy**, and then click **Partial Image**. The cursor changes to a "+" 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, split the design into multiple pages to export or to copy smaller portions of the design. For more information about controlling how much of your design is shown on each schematic page, refer to "Partitioning the Schematic into Pages" on page 13–29. As an alternative, use the Partial Image feature to copy a portion of the image.

Printing

To print your schematic page, on the File menu, click **Print**. You can print each schematic page onto one page, or you can print selected parts of your schematic onto one page with the **Selection** option. To control how much of your design is shown on each schematic page, refer to "Partitioning the Schematic into Pages" on page 13–29.

You cannot print the **Netlist Navigator** pane in the RTL Viewer and Technology Map Viewer and the table view of the State Machine Viewer. You can use the State Machine Viewer **Copy** command to copy the table to a text editor and print from the text editor.

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 netlist viewers provide a number of features in the **Netlist Navigator** pane 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, and 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

Date	Document Version	Changes Made
July 2010	10.0.0	■ Updated screenshots ■ Updated chapter for the Quartus II software version 10.0, including major user interface changes
November 2009	9.1.0	■ Updated devices ■ Minor text edits
March 2009	9.0.0	■ 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
November 2008	8.1.0	Changed page size to 8.5” × 11”
May 2008	8.0.0	■ 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

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

Take an online survey to provide feedback about this handbook chapter.

About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 10.0.

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 10.0 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 10.0

Volume 2: Design Implementation and Optimization

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

QII5V2-10.0.1

Copyright © 2010 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. Scripting and Constraint Entry

Chapter 1. Constraining Designs

Constraining Designs with the Quartus II GUI 1-1
Global Constraints 1-2
Node, Entity, and Instance-Level Constraints 1-3
Probing Between Components of the Quartus II GUI 1-4
SDC and the TimeQuest Timing Analyzer 1-4
Constraining Designs with Tcl 1-4
Quartus II Settings Files and Tcl 1-5
Timing Analysis with Synopsys Design Constraints and Tcl 1-8
A Fully Iterative Scripted Flow 1-9
Document Revision History 1-9

Chapter 2. Command-Line Scripting

Benefits of Command-Line Executables 2-1
Introductory Example 2-2
Command-Line Scripting Help 2-3
Command-Line Option Details 2-4
Option Precedence 2-5
Compilation with quartus_sh --flow 2-7
Text-Based Report Files 2-7
Makefile Implementation 2-9
The MegaWizard Plug-In Manager 2-11
Command-Line Support 2-12
Module and Wizard Names 2-13
Ports and Parameters 2-14
Invalid Configurations 2-15
Strategies to Determine Port and Parameter Values 2-15
Optional Files 2-15
Parameter File 2-16
Working Directory 2-17
Variation File Name 2-17
Command-Line Scripting Examples 2-17
Create a Project and Apply Constraints 2-17
Check Design File Syntax 2-18
Create a Project and Synthesize a Netlist Using Netlist Optimizations 2-19
Archive and Restore Projects 2-20
Perform I/O Assignment Analysis 2-20
Update Memory Contents Without Recompiling 2-20
Create a Compressed Configuration File 2-21
Fit a Design as Quickly as Possible 2-21
Fit a Design Using Multiple Seeds 2-22
The QFlow Script 2-23
Document Revision History 2-23

Chapter 3. Tcl Scripting

Introduction . . . 3-1
What is Tcl? . . . 3-2
Quartus II Tcl Packages . . . 3-2
Loading Packages . . . 3-3
Quartus II Tcl API Help . . . 3-3
Command-Line Options: -t, -s, and --tcl_eval . . . 3-5
Run a Tcl Script . . . 3-6
Interactive Shell Mode . . . 3-6
Evaluate as Tcl . . . 3-6
Using the Quartus II Tcl Console Window . . . 3-6
End-to-End Design Flows . . . 3-7
Creating Projects and Making Assignments . . . 3-8
HardCopy Device Design . . . 3-8
Compiling Designs . . . 3-9
The flow Package . . . 3-9
Compile All Revisions . . . 3-9
Reporting . . . 3-10
Creating .csv Files for Excel . . . 3-11
Timing Analysis . . . 3-12
Classic Timing Analysis . . . 3-12
Advanced Classic Timing Analysis . . . 3-14
TimeQuest Timing Analysis . . . 3-16
Automating Script Execution . . . 3-16
Execution Example . . . 3-17
Controlling Processing . . . 3-17
Displaying Messages . . . 3-18
Other Scripting Features . . . 3-18
Natural Bus Naming . . . 3-18
Short Option Names . . . 3-19
Using Collection Commands . . . 3-19
The foreach_in_collection Command . . . 3-19
The get_collection_size Command . . . 3-19
Using the post_message Command . . . 3-20
Accessing Command-Line Arguments . . . 3-20
Using the cmdline Package . . . 3-21
Using the Quartus II Tcl Shell in Interactive Mode . . . 3-22
Using the tclsh Shell . . . 3-23
Tcl Scripting Basics . . . 3-24
Hello World Example . . . 3-24
Variables . . . 3-24
Substitutions . . . 3-24
Variable Value Substitution . . . 3-24
Nested Command Substitution . . . 3-25
Backlash Substitution . . . 3-25
Arithmetic . . . 3-25
Lists . . . 3-25
Arrays . . . 3-26
Control Structures . . . 3-27
Procedures . . . 3-27
File I/O . . . 3-28
Syntax and Comments . . . 3-29
External References . . . 3-29
Document Revision History . . . 3-30

Chapter 4. Managing Quartus II Projects

Managing Your Quartus II Projects . . . 4-1
Creating a New Project . . . 4-1
File Association . . . 4-2
Editing Text-Based Designs with the Quartus II Text Editor . . . 4-3
Creating Assignments . . . 4-3
Quartus II Settings File . . . 4-3
Preserving QSF Format . . . 4-4
Quartus II Default Settings File . . . 4-4
Using Revisions with Your Design . . . 4-5
Creating a Revision, Specifying the Current Revision, and Comparing Revisions . . . 4-5
Deleting a Revision . . . 4-6
Creating New Copies of Your Design . . . 4-6
Archiving Projects . . . 4-7
Archiving a Project . . . 4-8
Restoring an Archived Project . . . 4-10
Exporting and Importing Version-Compatible Database Files . . . 4-11
Migrating to a New Version . . . 4-12
Saving the Database in a Version-Compatible Format . . . 4-12
Quartus II Project Platform Migration . . . 4-13
File Names and Hierarchies . . . 4-13
Specifying Libraries . . . 4-15
Quartus II Search Path Precedence Rules . . . 4-16
Quartus II-Generated Files for Third-Party EDA Tools . . . 4-16
Migrating Database Files Between Platforms . . . 4-17
Working with Messages . . . 4-18
Messages Window . . . 4-18
Hiding Messages . . . 4-19
Message Suppression . . . 4-20
Managing Projects in a Team-Based Design Environment . . . 4-23
Scripting Support . . . 4-24
Managing Revisions . . . 4-24
Creating Revisions . . . 4-24
Setting the Current Revision . . . 4-25
Getting a List of Revisions . . . 4-25
Deleting Revisions . . . 4-25
Archiving Projects . . . 4-25
Restoring Archived Projects . . . 4-26
Importing and Exporting Version-Compatible Databases . . . 4-26
Specifying Libraries Using Scripts . . . 4-27
Conclusion . . . 4-28
Document Revision History . . . 4-28

Section II. I/O and PCB Tools

Chapter 5. I/O Management

Understanding Altera Pin Terminology . . . 5-1
Package Pins . . . 5-2
Pads . . . 5-2
I/O Banks . . . 5-3
VREF Groups . . . 5-4
I/O Planning Overview . . . 5-5
Selecting a Device . . . 5-7

Working with Third-Party PCB Tools . . . 5-7
Performing Early I/O Planning with the Pin Planner . . . 5-8
Instantiating or Importing IP Cores in the Pin Planner . . . 5-9
Adding and Connecting Nodes . . . 5-9
Setting Up and Creating the Top-Level Design File . . . 5-10
Importing and Exporting Pin Assignments . . . 5-11
Importing and Exporting Assignments with the Quartus II Software . . . 5-12
Importing and Exporting Assignments with Third-Party PCB Tools . . . 5-12
Creating Pin-Related Assignments . . . 5-13
Creating Pin Assignments With the Pin Planner . . . 5-14
Finding Compatible Pin Locations with the Pin Finder . . . 5-15
Verifying Pin Migration Compatibility . . . 5-15
Viewing Simultaneous Switching Noise (SSN) Results . . . 5-17
Creating Location Assignments . . . 5-17
Creating Exclusive I/O Group Assignments . . . 5-18
Changing the Slew Rate and Drive Strength . . . 5-18
Assigning Locations for Differential Pins . . . 5-19
Creating Pin Assignments with the Chip Planner . . . 5-20
Creating Pin Assignments with Tcl Scripts . . . 5-20
Creating Pin Assignments in HDL Code . . . 5-21
Synthesis Attributes . . . 5-21
chip_pin and useioff . . . 5-21
altera_attribute . . . 5-22
Creating Pin Assignments with Low-Level I/O Primitives . . . 5-22
Validating Pin Assignments . . . 5-23
Validating Pin Assignments with the Live I/O Check Feature . . . 5-25
Validating Pin Assignments with I/O Assignment Analysis . . . 5-26
Running I/O Assignment Analysis without Design Files . . . 5-27
Running I/O Assignment Analysis with Design Files . . . 5-28
Optimizing I/O Assignment Analysis with Output Enable Group Logic Option Assignments . . . 5-30
Validating Pin Assignments with Full Compilation . . . 5-32
Performing I/O Timing Analysis . . . 5-33
Enabling and Configuring Advanced I/O Timing . . . 5-34
Defining Overall Board Trace Models . . . 5-34
Customizing the Board Trace Model in the Pin Planner . . . 5-35
Configuring Board Trace Models . . . 5-35
Specifying Near-End vs Far-End Timing Analysis . . . 5-37
Understanding Advanced I/O Timing Analysis Reports . . . 5-37
Adjusting I/O Timing and Power with Capacitive Loading . . . 5-38
Incorporating PCB Design Tools . . . 5-38
Scripting Support . . . 5-39
Running I/O Assignment Analysis . . . 5-39
Generating a Mapped Netlist . . . 5-39
Reserving Pins . . . 5-39
Creating Location Assignments . . . 5-40
Creating Exclusive I/O Group Assignments . . . 5-40
Changing the Slew Rate and Drive Strength . . . 5-40
Conclusion . . . 5-41
Document Revision History . . . 5-41

Chapter 6. Simultaneous Switching Noise (SSN) Analysis and Optimizations

Definitions . . . 6-1
Understanding SSN . . . 6-2

SSN Estimation Tools . . . 6-4
SSN Analysis Overview . . . 6-5
Performing Early Pin-Out SSN Analysis . . . 6-6
Performing Early Pin-Out SSN Analysis with the ESE Tool . . . 6-6
Performing Early Pin-Out SSN Analysis with the SSN Analyzer . . . 6-6
Performing Final Pin-Out SSN Analysis . . . 6-7
Design Factors Affecting SSN Results . . . 6-8
Optimizing Your Design for SSN Analysis . . . 6-8
Optimizing Pin Placements for Signal Integrity . . . 6-9
Specifying Board Trace Model Settings . . . 6-10
Defining PCB Layers and PCB Layer Thickness . . . 6-11
Specifying Signal Breakout Layers . . . 6-13
Creating I/O Assignments . . . 6-13
Decreasing Pessimism in SSN Analysis . . . 6-14
Excluding Pins as Aggressor Signals . . . 6-15
Performing SSN Analysis and Viewing Results . . . 6-15
Understanding the SSN Reports . . . 6-15
Summary Report . . . 6-16
Output Pins and Input Pins Reports . . . 6-16
Unanalyzed Pins Report . . . 6-16
Confidence Metric Details Report . . . 6-16
Viewing SSN Analysis Results in the Pin Planner . . . 6-16
Decreasing Processing Time for SSN Analysis . . . 6-17
Scripting Support . . . 6-17
Optimizing Pin Placements for Signal Integrity . . . 6-18
Defining PCB Layers and PCB Layer Thickness . . . 6-18
Specifying Signal Breakout Layers . . . 6-18
Decreasing Pessimism in SSN Analysis . . . 6-19
Performing SSN Analysis . . . 6-19
Conclusion . . . 6-20
Document Revision History . . . 6-20

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 . . . 7-13
Running and Interpreting LineSim Simulations . . . 7-15
Simulation with HSPICE Models . . . 7-16
Supported Devices and Signaling . . . 7-16
Accessing HSPICE Simulation Kits . . . 7-17
The Double Counting Problem in HSPICE Simulations . . . 7-17

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 . . . 7-21
Invoking HSPICE Writer from the Command Line . . . 7-21
Customizing Automatically Generated HSPICE Decks . . . 7-22
Running an HSPICE Simulation . . . 7-23
Interpreting the Results of an Output Simulation . . . 7-23
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
Document Revision History . . . 7-42

Chapter 8. Mentor Graphics PCB Design Tools Support

FPGA-to-PCB Design Flow . . . 8-2
Performing Simultaneous Switching Noise (SSN) Analysis of Your FPGA . . . 8-3
Setting Up the Quartus II Software . . . 8-4
Generating a .pin File . . . 8-5
Generating an .fx File . . . 8-6
Creating a Backup .qsf . . . 8-6

FPGA-to-Board Integration with the I/O Designer Software 8-7
I/O Designer Database Wizard 8-8
Updating Pin Assignments from the Quartus II Software 8-11
Sending Pin Assignment Changes to the Quartus II Software 8-13
Protecting Assignments in the Quartus II Software 8-15
Generating Symbols for the DxDesigner Software 8-15
Setting Up the I/O Designer Software to Work with the DxDesigner Software 8-16
Creating Symbols with the Symbol Wizard 8-17
Exporting Symbols to the DxDesigner Software 8-19
Scripting Support 8-19
FPGA-to-Board Integration with the DxDesigner Software 8-20
DxDesigner Project Settings 8-21
DxDesigner Symbol Wizard 8-23
Conclusion 8-25
Document Revision History 8-25

Chapter 9. Cadence PCB Design Tools Support

Product Comparison 9-2
FPGA-to-PCB Design Flow 9-3
Performing Simultaneous Switching Noise (SSN) Analysis of Your FPGA 9-5
Setting Up the Quartus II Software 9-5
Generating a .pin File 9-6
FPGA-to-Board Integration with the Cadence Allegro Design Entry HDL Software 9-6
Creating Symbols 9-6
Cadence Allegro PCB Librarian Part Developer Tool 9-7
Instantiating the Symbol in the Cadence Allegro Design Entry HDL Software 9-12
FPGA-to-Board Integration with Cadence Allegro Design Entry CIS Software 9-13
Creating a Cadence Allegro Design Entry CIS Project 9-13
Generating a Part 9-14
Splitting a Part 9-15
Instantiating a Symbol in a Design Entry CIS Schematic 9-17
Altera Libraries for the Cadence Allegro Design Entry CIS Software 9-18
Conclusion 9-20
Document Revision History 9-20

Chapter 10. Reviewing Printed Circuit Board Schematics with the Quartus II Software

Reviewing Quartus II Software Settings 10-1
Device and Pins Options Dialog Box Settings 10-2
Configuration Page Settings 10-2
Unused Pin Page Settings 10-2
Dual-Purpose Pins Page Settings 10-3
Voltage Page Settings 10-3
Error Detection CRC Page Settings 10-3
Voltage Page Settings 10-3
Reviewing Device Pin-Out Information in the Fitter Report 10-4
Reviewing Compilation Error and Warning Messages 10-5
Running the HardCopy Design Readiness Check 10-6
Using Additional Quartus II Software Features 10-6
Using Additional Quartus II Software Tools 10-6
Pin Planner 10-6
SSN Analyzer 10-7
Conclusion 10-7
Document Revision History 10-7

Section III. Area, Timing, Power, and Compilation Time Optimization

Chapter 11. Design Optimization Overview

Introduction . . . 11-1
Physical Implementation . . . 11-1
Trade-Offs and Limitations . . . 11-1
Preserving Results and Enabling Teamwork . . . 11-2
Reducing Area . . . 11-2
Reducing Critical Path Delay . . . 11-3
Reducing Power Consumption . . . 11-3
Reducing Runtime . . . 11-3
Using Quartus II Tools . . . 11-4
Design Analysis . . . 11-4
Advisors . . . 11-4
Design Space Explorer . . . 11-5
Conclusion . . . 11-5
Document Revision History . . . 11-5

Chapter 12. Reducing Compilation Time

Compilation Time Optimization Techniques . . . 12-1
Compilation Time Advisor . . . 12-2
Strategies to Reduce the Overall Compilation Time . . . 12-2
Using Parallel Compilation with Multiple Processors . . . 12-2
Using Incremental Compilation . . . 12-3
Using the Smart Compilation Setting . . . 12-4
Using Rapid Recompile . . . 12-4
Reducing Synthesis Time and Synthesis Netlist Optimization Time . . . 12-5
Settings to Reduce Synthesis Time and Synthesis Netlist Optimization Time . . . 12-5
Use Appropriate Coding Style to Reduce Synthesis Time . . . 12-5
Using Early Timing Estimation . . . 12-6
Reducing Placement Time . . . 12-6
Fitter Effort Setting . . . 12-7
Placement Effort Multiplier Settings . . . 12-7
Final Placement Optimization Levels . . . 12-7
Physical Synthesis Effort Settings . . . 12-7
Limit to One Fitting Attempt . . . 12-8
Preserving Placement, Incremental Compilation, and LogicLock Regions . . . 12-8
Reducing Routing Time . . . 12-8
Identifying Routing Congestion in the Chip Planner . . . 12-8
Placement Effort Multiplier Setting . . . 12-9
Preserving Routing Incremental Compilation and LogicLock Regions . . . 12-9
Setting Process Priority . . . 12-9
Conclusion . . . 12-9
Document Revision History . . . 12-9

Chapter 13. Area and Timing Optimization

Optimizing Your Design . . . 13-2
Initial Compilation: Required Settings . . . 13-3
Device Settings . . . 13-3
I/O Assignments . . . 13-3
Timing Requirement Settings . . . 13-4
Device Migration Settings . . . 13-5
Partitions and Floorplan Assignments for Incremental Compilation . . . 13-5

Initial Compilation: Optional Settings . . . 13-6
Design Assistant . . . 13-6
Optimize Hold Timing . . . 13-6
Limit to One Fitting Attempt . . . 13-7
Optimize Multi-Corner Timing . . . 13-7
Fitter Effort Setting . . . 13-8
Design Analysis . . . 13-9
Error and Warning Messages . . . 13-9
Ignored Timing Constraints . . . 13-10
Resource Utilization . . . 13-10
I/O Timing (Including t_{PD}) . . . 13-11
Register-to-Register Timing . . . 13-12
Timing Analysis with the TimeQuest Timing Analyzer . . . 13-12
Timing Analysis with the Classic Timing Analyzer . . . 13-13
Tips for Analyzing Failing Paths . . . 13-14
Tips for Analyzing Failing Clock Paths that Cross Clock Domains . . . 13-15
Global Routing Resources . . . 13-16
Resource Utilization Optimization Techniques (LUT-Based Devices) . . . 13-16
Using the Resource Optimization Advisor . . . 13-16
Resolving Resource Utilization Issues Summary . . . 13-17
I/O Pin Utilization or Placement . . . 13-17
Use I/O Assignment Analysis . . . 13-17
Modify Pin Assignments or Choose a Larger Package . . . 13-17
Logic Utilization or Placement . . . 13-18
Optimize Synthesis for Area, Not Speed . . . 13-18
Restructure Multiplexers . . . 13-18
Perform WYSIWYG Primitive Resynthesis with Balanced or Area Setting . . . 13-19
Use Register Packing . . . 13-19
Remove Fitter Constraints . . . 13-21
Change State Machine Encoding . . . 13-21
Flatten the Hierarchy During Synthesis . . . 13-22
Retarget Memory Blocks . . . 13-22
Use Physical Synthesis Options to Reduce Area . . . 13-23
Retarget or Balance DSP Blocks . . . 13-24
Optimize Source Code . . . 13-25
Use a Larger Device . . . 13-25
Routing . . . 13-25
Set Auto Packed Registers to Sparse or Sparse Auto . . . 13-25
Set Fitter Aggressive Routability Optimizations to Always . . . 13-26
Increase Placement Effort Multiplier . . . 13-26
Increase Router Effort Multiplier . . . 13-27
Remove Fitter Constraints . . . 13-27
Optimize Synthesis for Area, Not Speed . . . 13-27
Optimize Source Code . . . 13-28
Use a Larger Device . . . 13-28
Timing Optimization Techniques (LUT-Based Devices) . . . 13-29
Timing Optimization Advisor . . . 13-29
I/O Timing Optimization . . . 13-30
Improving Setup and Clock-to-Output Times Summary . . . 13-30
Timing-Driven Compilation . . . 13-31
Fast Input, Output, and Output Enable Registers . . . 13-32
Programmable Delays . . . 13-32
Use PLLs to Shift Clock Edges . . . 13-33
Use Fast Regional Clock Networks and Regional Clocks Networks . . . 13-34

Change How Hold Times are Optimized for MAX II Devices . . . 13-34
Register-to-Register Timing Optimization Techniques (LUT-Based Devices) . . . 13-34
Improving Register-to-Register Timing Summary . . . 13-34
Physical Synthesis Optimizations . . . 13-35
Turn Off Extra-Effort Power Optimization Settings . . . 13-38
Optimize Synthesis for Speed, Not Area . . . 13-38
Flatten the Hierarchy During Synthesis . . . 13-39
Set the Synthesis Effort to High . . . 13-40
Change State Machine Encoding . . . 13-40
Duplicate Logic for Fan-Out Control . . . 13-40
Prevent Shift Register Inference . . . 13-40
Use Other Synthesis Options Available in Your Synthesis Tool . . . 13-41
Fitter Seed . . . 13-41
Set Maximum Router Timing Optimization Level . . . 13-42
Enable Beneficial Skew Optimization . . . 13-42
Optimize Source Code . . . 13-42
LogicLock Assignments . . . 13-43
Hierarchy Assignments . . . 13-43
Path Assignments . . . 13-44
Location Assignments and Back-Annotation . . . 13-45
Custom Regions . . . 13-46
Metastability Analysis and Optimization Techniques . . . 13-46
Resource Utilization Optimization Techniques (Macrocell-Based CPLDs) . . . 13-46
Use Dedicated Inputs for Global Control Signals . . . 13-47
Reserve Device Resources . . . 13-47
Pin Assignment Guidelines and Procedures . . . 13-47
Control Signal Pin Assignments . . . 13-48
Output Enable Pin Assignments . . . 13-48
Estimate Fan-In When Assigning Output Pins . . . 13-48
Outputs Using Parallel Expander Pin Assignments . . . 13-48
Resolving Resource Utilization Problems . . . 13-49
Resolving Macrocell Usage Issues . . . 13-49
Resolving Routing Issues . . . 13-50
Using LCELL Buffers to Reduce Required Resources . . . 13-51
Timing Optimization Techniques (Macrocell-Based CPLDs) . . . 13-52
Improving Setup Time . . . 13-53
Improving Clock-to-Output Time . . . 13-53
Improving Propagation Delay (t_{PD}) . . . 13-54
Improving Maximum Frequency (f_{MAX}) . . . 13-55
Optimizing Source Code—Pipelining for Complex Register Logic . . . 13-55
Other Optimization Resources . . . 13-56
Design Space Explorer . . . 13-56
Other Optimization Advisors . . . 13-56
Scripting Support . . . 13-56
Initial Compilation Settings . . . 13-57
Resource Utilization Optimization Techniques (LUT-Based Devices) . . . 13-57
I/O Timing Optimization Techniques (LUT-Based Devices) . . . 13-58
Register-to-Register Timing Optimization Techniques (LUT-Based Devices) . . . 13-59
Duplicate Logic for Fan-Out Control . . . 13-59
Conclusion . . . 13-59
Document Revision History . . . 13-60

Chapter 14. Power Optimization

Power Dissipation . . . 14-2

Design Space Explorer . . . 14-3
Power-Driven Compilation . . . 14-4
Power-Driven Synthesis . . . 14-4
Power-Driven Fitter . . . 14-8
Area-Driven Synthesis . . . 14-10
Gate-Level Register Retiming . . . 14-10
Design Guidelines . . . 14-11
Clock Power Management . . . 14-11
LAB-Wide Clock Enable Example . . . 14-13
Reducing Memory Power Consumption . . . 14-13
Memory Power Reduction Example . . . 14-15
Pipelining and Retiming . . . 14-16
Architectural Optimization . . . 14-18
I/O Power Guidelines . . . 14-18
Dynamically Controlled On-Chip Terminations . . . 14-20
Power Optimization Advisor . . . 14-21
Power Optimization Advisor Example . . . 14-21
Conclusion . . . 14-23
Document Revision History . . . 14-23

Chapter 15. Analyzing and Optimizing the Design Floorplan

Chip Planner Overview . . . 15-1
Starting the Chip Planner . . . 15-2
Chip Planner Toolbar . . . 15-3
Chip Planner Tasks, Layers, and Editing Modes . . . 15-3
LogicLock Regions . . . 15-4
Creating LogicLock Regions . . . 15-5
Creating LogicLock Regions with the Project Navigator . . . 15-5
Creating LogicLock Regions with the Chip Planner . . . 15-5
Placing LogicLock Regions . . . 15-5
Placing Device Resources into LogicLock Regions . . . 15-5
LogicLock Regions Window . . . 15-6
Assigning LogicLock Region Content . . . 15-7
Hierarchical (Parent and Child) LogicLock Regions . . . 15-8
Reserved LogicLock Region . . . 15-8
Creating Non-Rectangular LogicLock Regions . . . 15-8
Excluded Resources . . . 15-9
Additional Quartus II LogicLock Design Features . . . 15-10
Analysis and Synthesis Resource Utilization by Entity . . . 15-10
Quartus II Revisions Feature . . . 15-10
LogicLock Assignment Precedence . . . 15-10
Virtual Pins . . . 15-11
Using LogicLock Regions in the Chip Planner . . . 15-12
Viewing Connections Between LogicLock Regions in the Chip Planner . . . 15-12
Using LogicLock Regions with the Design Partition Planner . . . 15-12
Design Floorplan Analysis Using the Chip Planner . . . 15-13
Chip Planner Floorplan Views . . . 15-13
Bird's Eye View . . . 15-13
Selected Elements Window . . . 15-14
Viewing Architecture-Specific Design Information . . . 15-14
Viewing Available Clock Networks in the Device . . . 15-15
Viewing Critical Paths . . . 15-16
Viewing Routing Congestion . . . 15-16
Viewing I/O Banks . . . 15-17

Generating Fan-In and Fan-Out Connections . . . 15-18
Generating Immediate Fan-In and Fan-Out Connections . . . 15-19
Highlight Routing . . . 15-20
Show Delays . . . 15-21
Exploring Paths in the Chip Planner . . . 15-21
Locate Path from the Timing Analysis Report to the Chip Planner . . . 15-21
Analyzing Connections for a Path . . . 15-22
Viewing Assignments in the Chip Planner . . . 15-22
Viewing Routing Channels for a Path in the Chip Planner . . . 15-23
Delay Information Table . . . 15-24
Viewing High-Speed and Low-Power Tiles in the Chip Planner . . . 15-25
Scripting Support . . . 15-26
Initializing and Uninitializing a LogicLock Region . . . 15-27
Creating or Modifying LogicLock Regions . . . 15-27
Obtaining LogicLock Region Properties . . . 15-27
Assigning LogicLock Region Content . . . 15-27
Save a Node-Level Netlist for the Entire Design into a Persistent Source File . . . 15-27
Setting LogicLock Assignment Priority . . . 15-28
Assigning Virtual Pins . . . 15-28
Conclusion . . . 15-28
Document Revision History . . . 15-29

Chapter 16. Netlist Optimizations and Physical Synthesis

WYSIWYG Primitive Resynthesis . . . 16-1
Performing Physical Synthesis Optimizations . . . 16-3
Automatic Asynchronous Signal Pipelining . . . 16-5
Physical Synthesis for Combinational Logic . . . 16-6
Physical Synthesis for Registers—Register Duplication . . . 16-6
Physical Synthesis for Registers—Register Retiming . . . 16-7
Preserving Your Physical Synthesis Results . . . 16-10
Physical Synthesis Options for Fitting . . . 16-11
Applying Netlist Optimization Options . . . 16-12
Scripting Support . . . 16-12
Synthesis Netlist Optimizations . . . 16-13
Physical Synthesis Optimizations . . . 16-13
Incremental Compilation . . . 16-14
Back-Annotating Assignments . . . 16-14
Conclusion . . . 16-14
Document Revision History . . . 16-15

Section IV. Engineering Change Management

Chapter 17. Engineering Change Management with the Chip Planner

Engineering Change Orders . . . 17-2
Performance Preservation . . . 17-2
Compilation Time . . . 17-3
Verification . . . 17-3
Change Modification Record . . . 17-3
ECO Design Flow . . . 17-4
The Chip Planner Overview . . . 17-5
Opening the Chip Planner . . . 17-5
The Chip Planner Tasks and Layers . . . 17-6
The Chip Planner Floorplan Views . . . 17-6

First-Level View . . . 17-6
Second-Level View . . . 17-8
Third-Level View . . . 17-9
Bird's Eye View . . . 17-9
Performing ECOs with the Chip Planner (Floorplan View) . . . 17-10
Creating Atoms . . . 17-10
Creating ALM Atoms . . . 17-10
Creating Logic Element Atoms . . . 17-12
Deleting Atoms . . . 17-14
Moving Atoms . . . 17-14
Check and Save Netlist Changes . . . 17-14
Performing ECOs in the Resource Property Editor . . . 17-15
Logic Element . . . 17-15
Logic Element Schematic View . . . 17-16
LE Properties . . . 17-16
Modes of Operation . . . 17-17
Sum and Carry Equations . . . 17-17
sload and sclr Signals . . . 17-17
Register Cascade Mode . . . 17-17
Cell Delay Table . . . 17-17
LE Connections . . . 17-18
Delete an LE . . . 17-18
Adaptive Logic Module . . . 17-18
ALM Schematic . . . 17-19
ALM Properties . . . 17-20
ALM Connections . . . 17-20
FPGA I/O Elements . . . 17-20
Arria GX, Stratix, Stratix II, and Stratix GX I/O Elements . . . 17-21
Arria II GX, Stratix III, and Stratix IV I/O Elements . . . 17-21
Cyclone and Cyclone II I/O Elements . . . 17-22
Cyclone III I/O Elements . . . 17-23
MAX II I/O Elements . . . 17-24
FPGA RAM Blocks . . . 17-25
FPGA DSP Blocks . . . 17-26
Change Manager . . . 17-27
Complex Changes in the Change Manager . . . 17-27
Managing SignalProbe Signals . . . 17-27
Exporting Changes . . . 17-27
Scripting Support . . . 17-28
Common ECO Applications . . . 17-28
Adjust the Drive Strength of an I/O with the Chip Planner . . . 17-28
Modify the PLL Properties With the Chip Planner . . . 17-29
PLL Properties . . . 17-30
Adjusting the Duty Cycle . . . 17-31
Adjusting the Phase Shift . . . 17-31
Adjusting the Output Clock Frequency . . . 17-32
Adjusting the Spread Spectrum . . . 17-32
Modify the Connectivity between Resource Atoms . . . 17-32
Post ECO Steps . . . 17-33
Performing Static Timing Analysis . . . 17-33
Conclusion . . . 17-33
Document Revision History . . . 17-34

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 10.0 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 Constraining Designs
Revised: *July 2010*
Part Number: *QII52001-10.0.0*

Chapter 2 Command-Line Scripting
Revised: *July 2010*
Part Number: *QII52002-10.0.0*

Chapter 3 Tcl Scripting
Revised: *July 2010*
Part Number: *QII52003-10.0.0*

Chapter 4 Managing Quartus II Projects
Revised: *July 2010*
Part Number: *QII52012-10.0.0*

Chapter 5 I/O Management
Revised: *July 2010*
Part Number: *QII52013-10.0.0*

Chapter 6 Simultaneous Switching Noise (SSN) Analysis and Optimizations
Revised: *July 2010*
Part Number: *QII52018-10.0.0*

Chapter 7 Signal Integrity Analysis with Third-Party Tools
Revised: *July 2010*
Part Number: *QII53020-10.0.0*

Chapter 8 Mentor Graphics PCB Design Tools Support
Revised: *July 2010*
Part Number: *QII52015-10.0.0*

Chapter 9 Cadence PCB Design Tools Support
Revised: *July 2010*
Part Number: *QII52014-10.0.0*

Chapter 10 Reviewing Printed Circuit Board Schematics with the Quartus II Software
Revised: *July 2010*
Part Number: *QII52019-10.0.0*

Chapter 11 Design Optimization Overview
Revised: *August 2010*
Part Number: *QII52021-10.0.1*

Chapter 12 Reducing Compilation Time
Revised: *July 2010*
Part Number: *QII52022-10.0.0*

Chapter 13 Area and Timing Optimization
Revised: *August 2010*
Part Number: *QII52005-10.0.1*

Chapter 14 Power Optimization
Revised: *July 2010*
Part Number: *QII52016-10.0.0*

Chapter 15 Analyzing and Optimizing the Design Floorplan
Revised: *July 2010*
Part Number: *QII52006-10.0.0*

Chapter 16 Netlist Optimizations and Physical Synthesis
Revised: *July 2010*
Part Number: *QII52007-10.0.0*

Chapter 17 Engineering Change Management with the Chip Planner
Revised: *July 2010*
Part Number: *QII52017-10.0.0*

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. After you create a project and 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 constrain designs, 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 projects.

This section includes the following chapters:

- Chapter 1, Constraining Designs

  This chapter discusses the ways to constrain designs in the Quartus II software, including the tools avaliable in the Quartus II software GUI, as well as Tcl scripting flows.

- Chapter 2, Command-Line Scripting

  This chapter discusses Quartus II command-line executables, which 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.

- Chapter 3, Tcl Scripting

  This chapter discusses developing and running Tcl scripts in the Quartus II software to allow you to perform a wide range of functions, such as compiling a design or automating common tasks. This chapter includes sample Tcl scripts for automating the Quartus II software. You can modify these example scripts for use with your own designs.

- Chapter 4, Managing Quartus II Projects

  This chapter discusses the best ways to manage Quartus II projects. This chapter also discusses how to migrate your projects from one computing platform to another; create and compare revisions; and copy, archive and restore projects.

1. Constraining Designs

QII52001-10.0.0

This chapter discusses the various tools and methods for constraining and re-constraining Quartus II designs in different design flows, both with the Quartus II GUI and with Tcl to facilitate a scripted flow.

Constraints, sometimes known as assignments or logic options, control the way the Quartus II software implements a design for an FPGA. Constraints are also central in the way that the TimeQuest Timing Analyzer and the PowerPlay Power Analyzer inform synthesis, placement, and routing. There are several types of constraints:

- Global design constraints and software settings, such as device family selection, package type, and pin count.
- Entity-level constraints, such as logic options and placement assignments.
- Instance-level constraints.
- Pin assignments and I/O constraints.

User-created constraints are contained in one of two files: the Quartus II Settings File (**.qsf**) or, in the case of timing constraints, the Synopsys Design Constraints file (**.sdc**). Constraints and assignments made with the **Device** dialog box, **Settings** dialog box, Assignment Editor, Chip Planner, and Pin Planner are contained in the Quartus II Settings File. The **.qsf** file contains project-wide and instance-level assignments for the current revision of the project in Tcl syntax. You can create separate revisions of your project with different settings, and there is a separate **.qsf** file for each revision.

The TimeQuest Timing Analyzer uses industry-standard Synopsys Design Constraints, also using Tcl syntax, that are contained in Synopsys Design Constraints (**.sdc**) files. The TimeQuest Timing Analyzer GUI is a tool for making timing constraints and viewing the results of subsequent analysis.

There are several ways to constrain a design, each potentially more appropriate than the others, depending on your tool chain and design flow. You can constrain designs for compilation and analysis in the Quartus II software using the GUI, as well as using Tcl syntax and scripting. By combining the Tcl syntax of the **.qsf** files and the **.sdc** files with procedural Tcl, you can automate iteration over several different settings, changing constraints and recompiling.

Constraining Designs with the Quartus II GUI

In the Quartus II GUI, the New Project Wizard, **Device** dialog box, and **Settings** dialog box allow you to make global constraints and software settings. The Assignment Editor and Pin Planner are spreadsheet-style interfaces for constraining your design at the instance or entity level. The Assignment Editor and Pin Planner make constraint types and values available based on global design characteristics such as the targeted device. These tools help you verify that your constraints are valid before compilation by allowing you to pick only from valid values for each constraint.

The TimeQuest Timing Analyzer GUI allows you to make timing constraints in SDC format and view the effects of those constraints on the timing in your design. Before running the TimeQuest timing analyzer, you must specify initial timing constraints that describe the clock characteristics, timing exceptions, and external signal arrival and required times. The Quartus II Fitter optimizes the placement of logic in the device to meet your specified constraints.

For more information about timing constraints and the TimeQuest Timing Analyzer, refer to *About TimeQuest Timing Analysis* in Quartus II Help.

Global Constraints

Global constraints affect the entire Quartus II project and all of the applicable logic in the design. Many of these constraints are simply project settings, such as the targeted device selected for the design. Synthesis optimizations and global timing and power analysis settings can also be applied with globally. Global constraints are often made when running the New Project Wizard, or in the **Device** dialog box or the **Settings** dialog box, early project development.

The following are the most common types of global constraints:

- Target device specification
- Top-level entity of your design, and the names of the design files included in the project
- Operating temperature limits and conditions
- Physical synthesis optimizations
- Analysis and synthesis options and optimization techniques
- Verilog HDL and VHDL language versions used in your project
- Fitter effort and timing driven compilation settings
- **.sdc** files for the TimeQuest timing analyzer to use during analysis as part of a full compilation flow

Settings that direct compilation and analysis flows in the Quartus II software are also stored in the Quartus II Settings File for your project, including the following global software settings:

- Early Timing Estimate mode
- Settings for EDA tool integration such as third-party synthesis tools, simulation tools, timing analysis tools, and formal verification tools.
- Settings and settings file specifications for the Quartus II Assembler, SignalTap II Logic Analyzer, PowerPlay power analyzer, and SSN Analyzer.

Global constraints and software settings stored in the Quartus II settings file are specific to each revision of your design, allowing you to control the operation of the software differently for different revisions. For example, different revisions can specify different operating temperatures and different devices, so that you can compare results.

Only the valid assignments made in the Assignment Editor are saved in the Quartus II Settings File, which is located in the project directory. When you make a design constraint, the new assignment is placed on a new line at the end of the file.

When you create or update a constraint in the GUI, 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 more information about specifying initial global constraints and software settings, refer to *Setting up and Running a Compilation* in Quartus II Help. For more information about how the Quartus II software uses Quartus II Settings Files, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

Node, Entity, and Instance-Level Constraints

Node, entity, and instance-level constraints constrain a particular segment of the design hierarchy, as opposed to the entire design. In the Quartus II software GUI, most instance-level constraints are made with the Assignment Editor, Pin Planner, and Chip Planner. Both the Assignment Editor and Pin Planner aid you in correctly constraining your design, both passively, through device-and-assignment-determined pick lists, and actively, through live I/O checking.

You can assign logic functions to physical resources on the device, using location assignments with the Assignment Editor or the Chip Planner. Node, entity, and instance-level constraints take precedence over any global constraints that affect the same sections of the design hierarchy. You can edit and view all node and entity-level constraints you created in the Assignment Editor, or you can filter the assignments by choosing to view assignments only for specific locations, such as DSP blocks.

The Pin Planner provides a graphical representation of the target device, which allows you to easily plan, view, create, and edit pin assignments in terms of where the pins actually exist on the targeted device package. With the Pin Planner, you can visually identify I/O banks, VREF groups, edges, and differential pin pairings to assist you in the pin planning process. You can verify the legality of new and existing pin assignments with the live I/O check feature and view the results in the Live I/O Check Status window.

The Chip Planner allows you to view the device from a variety of different perspectives, and you can make precise assignments to specific floorplan locations. With the Chip Planner, you can adjust existing assignments to device resources, such as pins, logic cells, and LABs using drag and drop features and a graphical interface. You can also view equations and routing information, and demote assignments by dragging and dropping assignments to various regions in the Regions window.

For more information about the Assignment Editor, refer to *About the Assignment Editor* in Quartus II Help. For more information about the Chip Planner, refer to *About the Chip Planner* in Quartus II Help. For more information about the Pin Planner, refer to *About the Pin Planner* in Quartus II Help.

Probing Between Components of the Quartus II GUI

The Assignment Editor, Chip Planner, and Pin Planner let you locate nodes and instances in the source files for your design in other Quartus II viewers. You can select a cell in the Assignment Editor spreadsheet and locate the corresponding item in another applicable Quartus II software window, such as the Chip Planner. To locate an item from the Assignment Editor in another window, right-click the item of interest in the spreadsheet, point to **Locate**, and click the appropriate command.

You can also locate nodes in the Assignment Editor and other constraint tools from other windows within the Quartus II software. First, select the node or nodes in the appropriate window. For example, select an entity in the **Entity** list in the **Hierarchy** tab in the Project Navigator, or select nodes in the Chip Planner. 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.

For more information about the Assignment Editor, refer to *About the Assignment Editor* in Quartus II Help. For more information about the Chip Planner, refer to *About the Chip Planner* in Quartus II Help. For more information about the Pin Planner, refer to *About the Pin Planner* in Quartus II Help.

SDC and the TimeQuest Timing Analyzer

You can make individual timing constraints for individual entities, nodes, and pins with the Constraints menu of the TimeQuest Timing Analyzer. The TimeQuest Timing Analyzer GUI provides easy access to timing constraints, and reporting, without requiring knowledge of SDC syntax. As you specify commands and options in the GUI, the corresponding SDC or Tcl command appears in the Console. This lets you know exactly what constraint you have added to your Synopsys Design Constraints file, and also enables you to learn SDC syntax for use in scripted flows. The GUI also provides enhanced graphical reporting features.

Individual timing assignments override project-wide requirements. You can also assign timing exceptions to nodes and paths to avoid reporting of incorrect or irrelevant timing violations. The TimeQuest timing analyzer supports point-to-point timing constraints, wildcards to identify specific nodes when making constraints, and assignment groups to make individual constraints to groups of nodes.

For more information about timing constraints and the TimeQuest Timing Analyzer, refer to *About TimeQuest Timing Analysis* in Quartus II Help.

Constraining Designs with Tcl

Because **.sdc** files and **.qsf** files are both in Tcl syntax, you can modify these files to be part of a scripted constraint and compilation flow. With Quartus II Tcl packages, Tcl scripts can open projects, make the assignments procedurally that would otherwise be specified in a **.qsf** file, compile a design, and compare compilation results against known goals and benchmarks for the design. Such a script can further automate the iterative process by modifying design constraints and recompiling the design.

For more information about controlling the Quartus II software with Tcl, refer to *About Quartus II Tcl Scripting* in Quartus II Help.

Quartus II Settings Files and Tcl

QSF files use Tcl syntax, but, unmodified, are not executable scripts. However, you can embed QSF constraints in a scripted iterative compilation flow, where the script that automates compilation and custom results reporting also contains the design constraints. Example 1–1 shows an example QSF file with boilerplate comments removed.

Example 1–1. Quartus II Settings File

```
set_global_assignment -name FAMILY "Cyclone II"
set_global_assignment -name DEVICE EP2C35F672C6
set_global_assignment -name TOP_LEVEL_ENTITY chiptrip
set_global_assignment -name ORIGINAL_QUARTUS_VERSION 10.0
set_global_assignment -name PROJECT_CREATION_TIME_DATE "11:45:02  JUNE 08, 2010"
set_global_assignment -name LAST_QUARTUS_VERSION 10.0
set_global_assignment -name MIN_CORE_JUNCTION_TEMP 0
set_global_assignment -name MAX_CORE_JUNCTION_TEMP 85
set_instance_assignment -name PARTITION_HIERARCHY root_partition -to | -section_id Top
set_global_assignment -name PARTITION_NETLIST_TYPE SOURCE -section_id Top
set_global_assignment -name PARTITION_FITTER_PRESERVATION_LEVEL PLACEMENT_AND_ROUTING \
-section_id Top
set_global_assignment -name PARTITION_COLOR 16764057 -section_id Top
set_global_assignment -name LL_ROOT_REGION ON -section_id "Root Region"
set_global_assignment -name LL_MEMBER_STATE LOCKED -section_id "Root Region"
set_global_assignment -name STRATIX_DEVICE_IO_STANDARD "3.3-V LVTTL"
set_location_assignment PIN_P2 -to clk2
set_location_assignment PIN_AE4 -to ticket[0]
set_location_assignment PIN_J23 -to ticket[2]
set_location_assignment PIN_Y12 -to timeo[1]
set_location_assignment PIN_N2 -to reset
set_location_assignment PIN_R2 -to timeo[7]
set_location_assignment PIN_P1 -to clk1
set_location_assignment PIN_M3 -to ticket[1]
set_location_assignment PIN_AE24 -to ~LVDS150p/nCEO~
set_location_assignment PIN_C2 -to accel
set_location_assignment PIN_K4 -to ticket[3]
set_location_assignment PIN_B3 -to stf
set_location_assignment PIN_T9 -to timeo[0]
set_location_assignment PIN_M5 -to timeo[6]
set_location_assignment PIN_J8 -to dir[1]
set_location_assignment PIN_C5 -to timeo[5]
set_location_assignment PIN_F6 -to gt1
set_location_assignment PIN_P24 -to timeo[2]
set_location_assignment PIN_B2 -to at_altera
set_location_assignment PIN_P3 -to timeo[4]
set_location_assignment PIN_M4 -to enable
set_location_assignment PIN_E3 -to ~ASDO~
set_location_assignment PIN_E5 -to dir[0]
set_location_assignment PIN_R25 -to timeo[3]
set_location_assignment PIN_D3 -to ~nCSO~
set_location_assignment PIN_G4 -to gt2
set_global_assignment -name MISC_FILE "D:/altera/chiptrip/chiptrip.dpf"
set_global_assignment -name USE_TIMEQUEST_TIMING_ANALYZER ON
set_global_assignment -name POWER_PRESET_COOLING_SOLUTION \
"23 MM HEAT SINK WITH 200 LFPM AIRFLOW"
set_global_assignment -name POWER_BOARD_THERMAL_MODEL "NONE (CONSERVATIVE)"
set_global_assignment -name SDC_FILE chiptrip.sdc
```

Example 1–1 shows the way that the `set_global_assignment` Quartus II Tcl command makes all global constraints and software settings, with `set_location_assignment` constraining each I/O node in the design to a physical pin on the device.

However, after you initially create the Quartus II Settings File for your design, you can export the contents to a procedural, executable Tcl (**.tcl)** file. You can then use that generated script to restore certain settings after experimenting with other constraints. You can also use the generated Tcl script to archive your assignments instead of archiving the Quartus II Settings file itself.

To export your constraints as an executable Tcl script, on the Project menu, click **Generate Tcl File for Project**. Example 1–2 shows the constraints in Example 1–1 converted to an executable Tcl script.

Example 1–2. Generated Tcl Script for a Quartus II Project (Part 1 of 2)

```
# Quartus II: Generate Tcl File for Project
# File: chiptrip.tcl
# Generated on: Tue Jun 08 13:08:48 2010

# Load Quartus II Tcl Project package
package require ::quartus::project

set need_to_close_project 0
set make_assignments 1

# Check that the right project is open
if {[is_project_open]} {
    if {[string compare $quartus(project) "chiptrip"]} {
        puts "Project chiptrip is not open"
        set make_assignments 0
    }
} else {
    # Only open if not already open
    if {[project_exists chiptrip]} {
        project_open -revision chiptrip chiptrip
    } else {
        project_new -revision chiptrip chiptrip
    }
    set need_to_close_project 1
}

# Make assignments
if {$make_assignments} {
set_global_assignment -name FAMILY "Cyclone II"
set_global_assignment -name DEVICE EP2C35F672C6
set_global_assignment -name TOP_LEVEL_ENTITY chiptrip
set_global_assignment -name ORIGINAL_QUARTUS_VERSION 10.0
set_global_assignment -name PROJECT_CREATION_TIME_DATE "11:45:02  JUNE 08, 2010"
set_global_assignment -name LAST_QUARTUS_VERSION 10.0
set_global_assignment -name MIN_CORE_JUNCTION_TEMP 0
set_global_assignment -name MAX_CORE_JUNCTION_TEMP 85
set_instance_assignment -name PARTITION_HIERARCHY root_partition -to | -section_id Top
set_global_assignment -name PARTITION_NETLIST_TYPE SOURCE -section_id Top
set_global_assignment -name PARTITION_FITTER_PRESERVATION_LEVEL PLACEMENT_AND_ROUTING \
-section_id Top
```

Example 1–2. Generated Tcl Script for a Quartus II Project (Part 2 of 2)

```
set_global_assignment -name PARTITION_COLOR 16764057 -section_id Top
set_global_assignment -name LL_ROOT_REGION ON -section_id "Root Region"
set_global_assignment -name LL_MEMBER_STATE LOCKED -section_id "Root Region"
set_global_assignment -name STRATIX_DEVICE_IO_STANDARD "3.3-V LVTTL"
set_location_assignment PIN_P2 -to clk2
set_location_assignment PIN_AE4 -to ticket[0]
set_location_assignment PIN_J23 -to ticket[2]
set_location_assignment PIN_Y12 -to timeo[1]
set_location_assignment PIN_N2 -to reset
set_location_assignment PIN_R2 -to timeo[7]
set_location_assignment PIN_P1 -to clk1
set_location_assignment PIN_M3 -to ticket[1]
set_location_assignment PIN_AE24 -to ~LVDS150p/nCEO~
set_location_assignment PIN_C2 -to accel
set_location_assignment PIN_K4 -to ticket[3]
set_location_assignment PIN_B3 -to stf
set_location_assignment PIN_T9 -to timeo[0]
set_location_assignment PIN_M5 -to timeo[6]
set_location_assignment PIN_J8 -to dir[1]
set_location_assignment PIN_C5 -to timeo[5]
set_location_assignment PIN_F6 -to gt1
set_location_assignment PIN_P24 -to timeo[2]
set_location_assignment PIN_B2 -to at_altera
set_location_assignment PIN_P3 -to timeo[4]
set_location_assignment PIN_M4 -to enable
set_location_assignment PIN_E3 -to ~ASDO~
set_location_assignment PIN_E5 -to dir[0]
set_location_assignment PIN_R25 -to timeo[3]
set_location_assignment PIN_D3 -to ~nCSO~
set_location_assignment PIN_G4 -to gt2
set_global_assignment -name MISC_FILE "D:/altera/chiptrip/chiptrip.dpf"
set_global_assignment -name USE_TIMEQUEST_TIMING_ANALYZER ON
set_global_assignment -name POWER_PRESET_COOLING_SOLUTION \
"23 MM HEAT SINK WITH 200 LFPM AIRFLOW"
set_global_assignment -name POWER_BOARD_THERMAL_MODEL "NONE (CONSERVATIVE)"
set_global_assignment -name SDC_FILE chiptrip.sdc

   # Commit assignments
   export_assignments

   # Close project
   if {$need_to_close_project} {
      project_close
   }
}
```

After setting initial values for variables to control constraint creation and whether or not the project needs to be closed at the end of the script, the generated script checks to see if a project is open. If a project is open but it is not the correct project, in this case, **chiptrip**, the script prints `Project chiptrip is not open` to the console and does nothing else.

If no project is open, the script determines if **chiptrip** exists in the current directory. If the project exists, the script opens the project. If the project does not exist, the script creates a new project and opens the project.

The script then creates the constraints. After creating the constraints, the script writes the constraints to the Quartus II Settings File and then closes the project.

Timing Analysis with Synopsys Design Constraints and Tcl

Timing constraints used in analysis by the Quartus II TimeQuest Timing Analyzer are stored in **.sdc** files. Because they use Tcl syntax, the constraints in **.sdc** files can be incorporated into other scripts for iterative timing analysis. Example 1–3 shows a basic **.sdc** file for the **chiptrip** project.

Example 1–3. Initial .sdc file for the chiptrip Project

```
# ----------------------------------------

set_time_unit ns
set_decimal_places 3

# ----------------------------------------
#
create_clock -period 10.0 -waveform { 0 5.0 } clk2 -name clk2
create_clock -period 4.0 -waveform { 0 2.0 } clk1 -name clk1

# clk1 -> dir* : INPUT_MAX_DELAY = 1 ns
set_input_delay -max 1ns -clock clk1  [get_ports dir*]
# clk2 -> time* : OUTPUT_MAX_DELAY = -2 ns
set_output_delay -max -2ns -clock clk2  [get_ports time*]
```

Similar to the constraints in the Quartus II Settings File, you can make the SDC constraints in Example 1–3 part of an executable timing analysis script, as shown in example Example 1–4.

Example 1–4. Tcl Script Making Basic Timing Constraints and Performing Mult-Corner Timing Analysis

```
project_open chiptrip
create_timing_netlist

#
# Create Constraints
#
create_clock -period 10.0 -waveform { 0 5.0 } clk2 -name clk2
create_clock -period 4.0 -waveform { 0 2.0 } clk1 -name clk1

# clk1 -> dir* : INPUT_MAX_DELAY = 1 ns
set_input_delay -max 1ns -clock clk1  [get_ports dir*]
# clk2 -> time* : OUTPUT_MAX_DELAY = -2 ns
set_output_delay -max -2ns -clock clk2  [get_ports time*]

#
# Perform timing analysis for several different sets of operating conditions
#
foreach_in_collection oc [get_available_operating_conditions] {
    set_operating_conditions $oc
    update_timing_netlist

    report_timing -setup -npaths 1
    report_timing -hold -npaths 1
    report_timing -recovery -npaths 1
    report_timing -removal -npaths 1
    report_min_pulse_width -nworst 1
}

delete_timing_netlist
project_close
```

The script in Example 1–4 opens the project, creates a timing netlist, then constrains the two clocks in the design and applies input and output delay constraints. The clock settings and delay constraints are identical to those in the **.sdc** file shown in Example 1–3. The next section of the script updates the timing netlist for the constraints and performs multi-corner timing analysis on the design.

A Fully Iterative Scripted Flow

You can use the **::quartus::flow** Tcl package and other packages in the Quartus II Tcl API to add flow control to modify constraints and recompile your design in an automated flow. You can combine your timing constraints with the other constraints for your design, and embed them in an executable Tcl script that also iteratively compiles your design as different constraints are applied.

Each time such a modified generated script is run, it can modify the **.qsf** file and **.sdc** file for your project based on the results of iterative compilations, effectively replacing these files for the purposes of archiving and version control using industry-standard source control methods and practices.

This type of scripted flow can include automated compilation of a design, modification of design constraints, and recompilation of the design, based on how you foresee results and pre-determine next-step constraint changes in response to those results.

For more information about the Quartus II Tcl API, refer to *API Functions for Tcl* in Quartus II Help. For more information about controlling the Quartus II software with Tcl scripts, refer to *About Quartus II Tcl Scripting* in Quartus II Help.

Document Revision History

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

Table 1–1. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Rewrote chapter to more broadly cover all design constraint methods. Removed procedural steps and user interface details, and replaced with links to Quartus II Help.
November 2009	9.1.0	■ Added two notes. ■ Minor text edits.
March 2009	9.0.0	■ Revised and reorganized the entire chapter. ■ Added section “Probing to Source Design Files and Other Quartus II Windows” on page 1–2. ■ Added description of node type icons (Table 1–3). ■ Added explanation of wildcard characters.
November 2008	8.1.0	Changed to 8½” × 11” page size. No change to content.
May 2008	8.0.0	Updated Quartus II software 8.0 revision and date.

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

Take an online survey to provide feedback about this handbook chapter.

2. Command-Line Scripting

QII52002-10.0.0

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. This chapter includes the following topics:

- "Benefits of Command-Line Executables"
- "Introductory Example" on page 2–2
- "Compilation with quartus_sh --flow" on page 2–7
- "The MegaWizard Plug-In Manager" on page 2–11
- "Command-Line Scripting Examples" on page 2–17

Benefits of Command-Line Executables

The Quartus II command-line executables provide 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. These features enhance the ease of integration between the Quartus II software and other EDA synthesis, simulation, and verification software.

Command-line executables add 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, the executables themselves are relatively compact, both in file size and the amount of memory used when running. This memory usage reduction improves performance, and is particularly beneficial in design environments where computer networks or workstations are heavily used with reduced memory.

For a complete list of the Quartus II command-line executables, refer to *Using the Quartus II Executables in Shell Scripts* in Quartus II Help.

Introductory Example

The following introduction to command-line executables demonstrates how to create a project, fit the design, 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 ↵
quartus_asm filtref ↵
quartus_sta filtref ↵
```

The `quartus_map filtref --source=filtref.bdf --family=CYCLONE` command creates a new Quartus II project called **filtref** with **filtref.bdf** 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` 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_sta filtref` command performs basic timing analysis on the **filtref** project using the Quartus II TimeQuest Timing Analyzer, reporting worst-case setup slack, worst-case hold slack, and other measurements.

The TimeQuest Timing Analyzer employs Synopsys Design Constraints to fully analyze the timing of your design. For more information about using all of the features of **quartus_sta.exe**, refer to the *TimeQuest Timing Analyzer Quick Start Tutorial*.

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
quartus_map $PROJECT --source=$TOP_LEVEL_FILE --family=$FAMILY
quartus_fit $PROJECT --part=$PART --fmax=$FMAX
quartus_asm $PROJECT
quartus_sta $PROJECT
```

Edit the script as necessary and compile your project.

Command-Line Scripting Help

Help for 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.

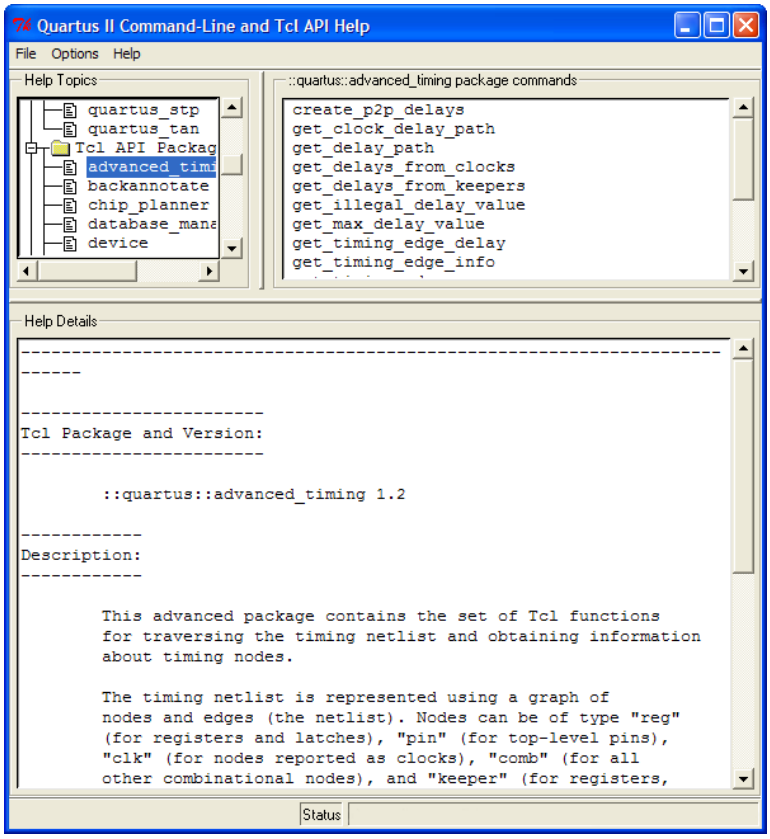
To use the Quartus II Command-Line and Tcl API Help browser, type the following command:

```
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 for 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. For more information refer to "Option Precedence" on page 2–5. 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.

For more information about the Quartus II Tcl scripting API, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about Quartus II project settings and assignments, refer to the *QSF Reference Manual*.

Option Precedence

If you use command-line executables, you must 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 the 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, 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, which reflects the assignments that existed when the project was compiled
- Command-line options

Table 2–1 lists the precedence for reading assignments depending on the value of the `--read_settings_files` option.

Table 2–1. 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 2–2 lists the locations to which assignments are written, depending on the value of the `--write_settings_files` command-line option.

Table 2–2. 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_sta fir_filter ↵
mv fir_filter_sta.rpt fir_filter_1_sta.rpt ↵
quartus_fit fir_filter --fmax=100MHz --write_settings_files=off ↵
quartus_sta fir_filter ↵
mv fir_filter_sta.rpt fir_filter_2_sta.rpt ↵
```

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_sta fir_filter`, runs timing analysis for the results of the previous fit.

The third command uses the UNIX `mv` command to copy the report file output from **quartus_sta** to a file with a new name, so that the results are not overwritten by subsequent timing analysis.

The fourth command reruns the Fitter with a global f_{MAX} requirement of 100 MHz. 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.

The fifth command reruns timing analysis, and the sixth command renames the report file, so that it is not overwritten by subsequent timing anlysis.

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

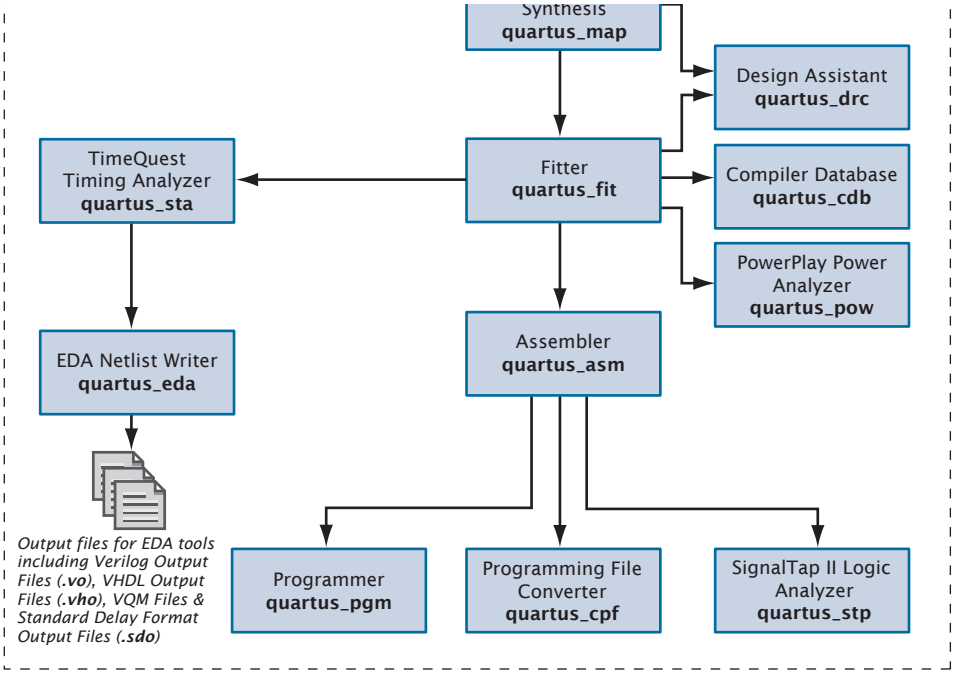
```
quartus_map filtref --source=filtref --part=EP1C12F256C6 ↵
quartus_fit filtref --fmax=100MHz --read_settings_files=off ↵
quartus_asm filtref --read_settings_files=off --write_settings_files=off ↵
```

In Example 2–4, the `quartus_asm` executable does not read or write settings files because they do not change any settings in the project.

Compilation with quartus_sh --flow

Figure 2–2 shows a typical Quartus II FPGA design flow.

Figure 2–2. Typical Design 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.

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_sta` executable to perform timing analysis on a project with the revision name **fir_filter** generates a report file named **fir_filter.sta.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 2–3.

You can use 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 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 *About Quartus II Tcl Scripting* in Quartus II Help.

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

```
#!/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_sta executable to do fast and slow-model
# timing analysis
quartus_sta myproject --model=slow
quartus_sta myproject --model=fast

# 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


# 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
###################################################################
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
```

Example 2–6. Sample Makefile (Part 2 of 2)

```
#####################################################################
# 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 are 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 provides a GUI-based flow to configure megafunction and IP variation files. However, you can use command-line options for the **qmegawiz** executable 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 for configuring the variation, and performs validation and error checking of your selected ports and parameters. When you create or update a variation with the GUI, the parameters and values are entered through the GUI provided by the Plug-In. When you create a Plug-In variation with the command line, you provide the parameters and values as command-line options.

Example 2–7 shows how to create a new variation file at a system command prompt.

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 versions, the variation files must be updated. To do this, use `qmegawiz -silent <variation file name>`.

Table 2–3 describes the supported options.

Table 2–3. 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 2–16.
`-p:<working directory>`	Sets the default working directory. Refer to"Working Directory" on page 2–17.

For information about specifying the module name or wizard name, refer to "Module and Wizard Names" on page 2–13.

For information about specifying ports and parameters, refer to "Ports and Parameters" on page 2–14.

For information about generating optional files, refer to "Optional Files" on page 2–15.

For information about specifying the variation file name, refer to "Variation File Name" on page 2–17.

Command-Line Support

Only the MegaWizard Plug-Ins listed in Table 2–4 support creation and update in command-line mode. For Plug-Ins not listed in the table, you must use the MegaWizard Plug-In Manager GUI for creation and updates.

Table 2–4. MegaWizard Plug-Ins with Command Line Support (Part 1 of 2)

MegaWizard Plug-In	Wizard Name	Module Name
alt2gxb	ALT2GXB	alt2gxb
alt4gxb	ALTGX	alt4gxb
altasmi_parallel	ALTASMI_PARALLEL	altasmi_parallel
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_abs	ALTFP_ABS	altfp_abs
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_exp	ALTFP_EXP	altfp_exp

Table 2–4. MegaWizard Plug-Ins with Command Line Support (Part 2 of 2)

MegaWizard Plug-In	Wizard Name	Module Name
altfp_inv_sqrt	ALTFP_INV_SQRT	altfp_inv_sqrt
altfp_inv	ALTFP_INV	altfp_inv
altfp_log	ALTFP_LOG	altfp_log
altfp_matrix_inv	ALTFP_MATRIX_INV	altfp_matrix_inv
altfp_matrix_mult	ALTFP_MATRIX_MULT	altfp_matrix_mult
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
altotp	ALTOTP	altotp
altpll_reconfig	ALTPLL_RECONFIG	altpll_reconfig
altpll	ALTPLL	altpll
altremote_update	ALTREMOTE_UPDATE	altremote_update
altshift_taps	ALTSHIFT_TAPS	altshift_taps
altsyncram	RAM: 2-PORT	altsyncram
	RAM: 1-PORT	
	ROM: 1-PORT	
alttemp_sense	ALTTEMP_SENSE	alttemp_sense
alt_c3gxb	ALT_C3GXB	alt_c3gxb
dcfifo	FIFO	dcfifo
scfifo		scfifo

Module and Wizard Names

You must specify the wizard or module name, shown in Table 2–4, 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 the Altera 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 Quartus II Help and other megafunction documentation. For example, 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 might 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 that 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 2–5 lists valid arguments for the `OPTIONAL_FILES` parameter.

Table 2–5. Arguments for the OPTIONAL_FILES Parameter

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.
`BB`	Controls the generation of the <*variation*>**_bb.v** file.
`SIM_NETLIST`	Controls the generation of the 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 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 files you subsequently specify are generated.

Table 2–6 shows examples for the OPTIONAL_FILES parameter and describes the result of each example.

Table 2–6. 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 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, the following command specifies a parameter file named **rom_params.txt**:

```
qmegawiz -silent module=altsyncram -f:rom_params.txt myrom.v ↵
```

The **rom_params.txt** parameter file can include options similar to the following:

```
RAM_BLOCK_TYPE=M4K DEVICE_FAMILY=Stratix WIDTH_A=5 WIDTHAD_A=5
    NUMWORDS_A=32 INIT_FILE=rom.hex OPERATION_MODE=ROM
```

If you use the `-f` option and the `-p` option together, the MegaWizard Plug-In Manager sources the parameter file in a directory specified with the `-p` option, or in a directory relative to that directory. For example, if you specify **C:\project\work** with the `-p` option and **work\params.txt** with the `-f` option, the MegaWizard Plug-In Manager attempts to source the file **params.txt** in **C:\project\work\work**.

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 2–7 shows the file extension that corresponds to supported HDL types.

Table 2–7. Variation File Extensions

Variation File HDL Type	Required File Extension
Verilog HDL	**.v**
VHDL	**.vhd**
AHDL	**.tdf**

Command-Line Scripting Examples

This section 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 yourself, 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, **filtref_sta_1.rpt** and **filtref_sta_2.rpt**.

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_sta filtref --model=fast --export_settings=off ↵
mv filtref_sta.rpt filtref_sta_1.rpt ↵
quartus_sta filtref --export_settings=off ↵
mv filtref_sta.rpt filtref_sta_2.rpt ↵
```

Type the following commands to create the design, apply constraints, and compile the design, without performing timing analysis:

```
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 information about register retiming, WYSIWYG primitive resynthesis, and other netlist optimization options, refer to 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.

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 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 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.

For more information about archiving and restoring projects, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

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 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 **.mif** or **.hexout** files. Then, rerun 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 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 the following command at a command prompt:

```
quartus_cpf -w <filename>.opt ↵
```

This interactive command guides you through some questions, 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 **.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 a command prompt, or refer to *Design Space Explorer* in 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` (Classic Timing Analyzer)
- `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.

Document Revision History

Table 2–8 shows the revision history for this chapter.

Table 2–8. Document Revision History (Part 1 of 2)

Date	Version	Changes Made
July 2010	10.0.0	Updated script examples to use quartus_sta instead of quartus_tan, and other minor updates throughout document.
November 2009	9.1.0	Updated Table 2–1 to add quartus_jli and quartus_jbcc executables and descriptions, and other minor updates throughout document.
March 2009	9.0.0	No change to content.

Table 2–8. Document Revision History (Part 2 of 2)

Date	Version	Changes Made
November 2008	8.1.0	Added the following sections: ■ "The MegaWizard Plug-In Manager" on page 2–11 ■ "Command-Line Support" on page 2–12 ■ "Module and Wizard Names" on page 2–13 ■ "Ports and Parameters" on page 2–14 ■ "Invalid Configurations" on page 2–15 ■ "Strategies to Determine Port and Parameter Values" on page 2–15 ■ "Optional Files" on page 2–15 ■ "Parameter File" on page 2–16 ■ "Working Directory" on page 2–17 ■ "Variation File Name" on page 2–17 ■ "Create a Compressed Configuration File" on page 2–21 Updated "Option Precedence" on page 2–5 to clarify how to control precedence Corrected Example 2–5 on page 2–8 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
May 2008	8.0.0	Updated "Referenced Documents" on page 2–20. Updated references in document.

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

Take an online survey to provide feedback about this handbook chapter.

3. Tcl Scripting

QII52003-10.0.0

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 3–3
- "Command-Line Options: -t, -s, and --tcl_eval" on page 3–5
- "End-to-End Design Flows" on page 3–7
- "Creating Projects and Making Assignments" on page 3–8
- "Compiling Designs" on page 3–9
- "Reporting" on page 3–10
- "Timing Analysis" on page 3–12
- "Automating Script Execution" on page 3–16
- "Other Scripting Features" on page 3–18
- "Using the Quartus II Tcl Shell in Interactive Mode" on page 3–22
- "Using the tclsh Shell" on page 3–23
- "Using the tclsh Shell" on page 3–23
- "Tcl Scripting Basics" on page 3–24

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 3–29.

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 3–24 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
insystem_source_probe	interact with the In-System Sources and Probes tool in an Altera device
jtag	Control the JTAG chain
logic_analyzer_interface	Query and modify the logic analyzer interface output pin state
misc	Perform miscellaneous tasks
project	Create and manage projects and revisions, make any project assignments including timing assignments
rapid_recompile	Manipulate Quartus II Rapid Recompile features
report	Get information from report tables, create custom reports
rtl	Traversing and querying the RTL netlist of your design
sdc_ext	Altera-specific SDC commands
sdc	Specifies constraints and exceptions to the TimeQuest Timing Analyzer
simulator	Configure and perform simulations

Table 3–1. Tcl Packages (Part 2 of 2)

Package Name	Package Description
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_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
timing	Annotate timing netlist with delay information, compute and report timing paths

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.

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.

To determine which executables load or make available each Tcl package, refer to *API Functions for Tcl* in Quartus II Help.

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 3–2 on page 3–4), 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.

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 3–2 summarizes the help options available in the Tcl environment.

Table 3–2. 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` ↵
<*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` ↵

Table 3–2. Help Options Available in the Quartus II Tcl Environment (Part 2 of 2)

Help Command	Description
`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*.

Command-Line Options: -t, -s, and --tcl_eval

Table 3–3 lists three command-line options you can use with executables that support Tcl.

Table 3–3. 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 3–20 for more information.

Interactive Shell Mode

Running an executable with the `-s` option starts an interactive Tcl shell. For example, to open the Quartus II TimeQuest Timing Analyzer executable in interactive shell mode, type the following command:

```
quartus_sta -s ↵
```

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.

If your design uses the Quartus II TimeQuest Timing Analyzer, you should perform scripted timing analysis in the TimeQuest Tcl console.

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 3–6 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. Run this script in the **fir_filter** directory, with the `quartus_sh` executable.

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.

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. Use the **flow** package instead of using system calls to run Quartus II executables from scripts.

Another way to run a Quartus II executable from Tcl environment is with the `qexec` Tcl command, a Quartus II implementation of the Tcl `exec` command. For example, to run Quartus II Analysis and Synthesis on a given project, use:

```
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 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 3–3 in a file called **compile_revisions.tcl** and type the following to run it:

```
quartus_sh -t compile_revisions.tcl <project name> ↵
```

Example 3–3. 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 Compilation Report to evaluate results. 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.

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 3–5.

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 3–4 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 3–4. 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 3–5 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 3–5. 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 3–6 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 3–6. 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 3–7.

Example 3–7. 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 3–8 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 3–8. 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 3–9 at a system command prompt to run this script.

Example 3–9.

```
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 **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.

Automating Script Execution

You can configure scripts to run automatically at various points during compilation. Use this capability to automatically run scripts that perform custom reporting, make specific assignments, and perform many other tasks.

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

A module is another term for 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.

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 Quartus II software runs the scripts as shown in Example 3–10.

Example 3–10.

```
<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 3–11 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 3–11.

```
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
```

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 3–12. 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 3–12. 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 3–16.

Refer to "Using the post_message Command" on page 3–20 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

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 3–13.

Example 3–13. 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

You can use short versions of command options, as long as they are unambiguous. For example, the `project_open` command supports two options: `-current_revision` and `-revision`. You can use any of the following abbreviations of the `-revision` option: `-r`, `-re`, `-rev`, `-revi`, `-revis`, and `-revisio`. You can use an option as short as `-r` because in the case of the `project_open` command no other option starts with the letter r. However, the `report_timing` command includes the options `-recovery` and `-removal`. You cannot use `-r` or `-re` to shorten either of those options, because the abbreviation would not be unique to only one option.

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. 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, refer to foreach_in_collection in Quartus II Help.

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 3–14 prints all instance assignments in an open project.

Example 3–14. 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 3–15 prints the number of global assignments in an open project.

Example 3–15. 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 3–16 shows how to use the `post_message` command.

Example 3–16. 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 3–17 shows code that prints all of the arguments in the `quartus(args)` variable.

Example 3–17. 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 3–18 at a command prompt.

Example 3–18. 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
```

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 3–19 uses the **cmdline** package.

Example 3–19. 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 3–26 at a command prompt.

Example 3–20. 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 3–21 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 3–21. 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 3–22.

Example 3–22. Simple Method to Open Projects

```
set proj_name [lindex $argv 0]
project_open $proj_name
```

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, type the following at the system command prompt to run the interactive Tcl shell:

```
quartus_sh -s ↵
```

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
```

To perform a full compilation of the FIR filter design, use the `execute_flow` command with the `-compile` option:

```
exectue_flow -compile ↵
```

This command compiles the FIR filter tutorial project, exporting the project assignments and running `quartus_map`, `quartus_fit`, `quartus_asm`, and `quartus_sta`. 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 3–23.

Example 3–23.

```
project_close ↵
```

To exit the interactive Tcl shell, type `exit ↵` at a Tcl prompt.

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 the examples in this chapter) in files and run them with the Quartus II executables or with the tclsh shell.

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 3–24. 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 3–24 assigns the value `1` to the variable named `a`.

Example 3–24. Assigning Variables

```
set a 1
```

To access the contents of a variable, use a dollar sign before the variable name. Example 3–25 prints `"Hello world"` in a different way.

Example 3–25. 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 3–25, 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 3–26 sets `a` to the length of the string `foo`.

Example 3–26. 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 3–27 shows how to use the backslash character for line continuation.

Example 3–27. 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 3–28 sets `b` to the sum of the value in the variable `a` and the square root of `2`.

Example 3–28. 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 3–29 sets `a` to a list with three numbers in it.

Example 3–29. 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 3–30 prints the second element (at index 1) in the list stored in `a`.

Example 3–30. Accessing List Elements

```
puts [lindex $a 1]
```

The `llength` command returns the length of a list. Example 3–31 prints the length of the list stored in `a`.

Example 3–31. 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 3–32 appends some elements to the list stored in `a`.

Example 3–32. 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 3–33 shows how to access the value for a particular index.

Example 3–33. 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 3–34 shows one way to iterate over all the values in an array.

Example 3–34. 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 3–35 prints whether the value of variable `a` positive, negative, or zero.

Example 3–35. 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 3–36 uses a `for` loop to print each element in a list.

Example 3–36. 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 3–37 uses a `foreach` loop to print each element in a list.

Example 3–37. foreach Loop

```
set a { 1 2 3 }
foreach element $a {
   puts "The list element is $element"
}
```

Example 3–38 uses a `while` loop to print each element in a list.

Example 3–38. 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 3–39 defines a procedure that multiplies two numbers and returns the result.

Example 3–39. Simple Procedure

```
proc multiply { x y } {
   set product [expr { $x * $y }]
   return $product
}
```

Example 3–40 shows how to use the `multiply` procedure in your code. You must define a procedure before your script calls it.

Example 3–40. 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 3–41 defines a procedure that prints an element in a global list of numbers, then calls the procedure.

Example 3–41. 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 3–42.

Example 3–42. 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 3–43.

Example 3–43. 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 3–44 uses the `gets` command to read each line of the file and then prints it out with its line number.

Example 3–44. 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 3–24, 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 3–45 is a valid line of code that includes a `set` command and a comment.

Example 3–45. 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 3–46 causes an error because of unbalanced curly braces.

Example 3–46. 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

Document Revision History

Table 3–4 shows the revision history for this chapter.

Table 3–4. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	Minor updates throughout document.
November 2009	9.1.0	■ Removed LogicLock example. ■ Added the incremental_compilation, insystem_source_probe, and rtl packages to Table 3-1 and Table 3-2. ■ Added quartus_map to table 3-2.
March 2009	9.0.0	■ Removed the "EDA Tool Assignments" section ■ Added the section "Compile All Revisions" on page 3–9 ■ Added the section "Using the tclsh Shell" on page 3–23
November 2008	8.1.0	Changed to 8½" × 11" page size. No change to content.
May 2008	8.0.0	Updated references.

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

Take an online survey to provide feedback about this handbook chapter.

4. Managing Quartus II Projects

QII52012-10.0.0

A Quartus II project contains all your design files, setting files, and other files necessary for the successful compilation of your design. This chapter discusses how to create and manage projects, and how to migrate them from one computing platform to another.

Today's larger and more sophisticated FPGA designs are often developed by several engineers and are constantly changing throughout the project. Designers must track their project changes to ensure efficient design coordination.

This chapter discusses the following topics:

- "Managing Your Quartus II Projects" on page 4–1
- "Exporting and Importing Version-Compatible Database Files" on page 4–11
- "Managing Projects in a Team-Based Design Environment" on page 4–23
- "Scripting Support" on page 4–24

Managing Your Quartus II Projects

To help you manage your FPGA designs, the Quartus® II software provides tools that allow you to perform the following tasks:

- "Creating a New Project"
- "Editing Text-Based Designs with the Quartus II Text Editor" on page 4–3
- "Creating Assignments" on page 4–3
- "Creating a Revision, Specifying the Current Revision, and Comparing Revisions" on page 4–5
- "Creating New Copies of Your Design" on page 4–6
- "Archiving Projects" on page 4–7
- "Working with Messages" on page 4–18

Creating a New Project

You can create a new project with the **New Project** wizard. The **New Project** wizard allows you to specify the working directory for the project, assign the project name, and designate the name of the top-level design entity. When you name the project, the same name is automatically assigned to the top-level design entity. If you want the name of the top-level design entity to be unique, you can rename the top-level design entity. You can also specify design files, other source files, libraries, and other EDA tools for your project, as well as the target device family and device. If you have no files to add when creating a new project, or you want to add files at a later time, or you do not wish to specify other EDA tools at this time, you can bypass these steps in the **New Project** wizard.

For more information about creating and opening a project, adding files to and removing files from a project, modifying project settings, saving project changes, and specifying the top-level entity, refer to *Managing Files in a Project* in Quartus II Help. For more information about libraries, refer to "Specifying Libraries" on page 4–15 and "Specifying Libraries Using Scripts" on page 4–27.

On the **General** page of the **Options** dialog box, you can also specify a default directory that automatically stores all project files.

After you create a new project, the Quartus II software automatically generates various settings files necessary for successful compilation, including:

- Quartus II Project File (**.qpf**)—Contains the name of your project and all the revisions of your project.
- 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**, refer to "Quartus II Settings File" on page 4–3.

If you create timing commands with the TimeQuest Timing Analyzer or Tcl commands, the Quartus II software creates a Synopsys Design Constraints File (**.sdc**) that contains your SDC commands.

For more information about SDC constraints, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

File Association

Quartus II project files are files that are associated with a Quartus II project, but are not design files in the project hierarchy. Most project-related files do not contain design logic. The Quartus II software supports project files such as Quartus II Project File (**.qpf**), Quartus II IP File (**.qip**), and Quartus II Settings File (**.qsf**), among others.

Quartus II design files are files that contain logic for a Quartus II project and are compiled by the Compiler. The Quartus II software also supports designs created from EDIF Input Files (**.edf**) or Verilog Quartus Mapping Files (**.vqm**) generated by EDA design entry and synthesis tools. You can also create Verilog HDL or VHDL designs in the Quartus II software and EDA design entry tools and either generate EDIF Input Files and VQM Files, or use the Verilog HDL or VHDL design files directly in Quartus II projects. The Quartus II software also supports use of Quartus II Exported Partition Files (**.qxp**) as source files containing entities you can add to your design.

For a list of supported Quartus II project files and design file types, refer to *Introduction to the Quartus II Software* manual.

When you run the Quartus II software version 9.1 or earlier, all file types are associated to the active Quartus II software. Beginning from the Quartus II software version 10.0, file type associations are not set for every run of the Quartus II software, because it has been set when you install the software.

If your Quartus II software files are associated to a different version of the Quartus II software, and you want to associate the files to the Quartus II software version 10.0, you can manually associate the files to the Quartus II software version 10.0 by typing the following command, as shown in Example 4–1:

Example 4–1.

```
<path to install directory>\quartus\bin\qreg.exe --file_assoc ↵
```

Editing Text-Based Designs with the Quartus II Text Editor

The Quartus II Text Editor is a flexible tool for entering text-based designs in AHDL, VHDL, and Verilog HDL design languages and Tcl scripting language.

You can use any text editor with the Quartus II software; however, the Quartus II Text Editor allows you to take advantage of features available only in the Quartus II software, including syntax coloring, error location, and predefined templates to help you with coding.

Using templates can increase the speed and accuracy of your design. The Quartus II software provides templates that allow you to insert predefined code directly into your design file; you can choose from several design languages, and you also can directly add TimeQuest analyzer design constraints and megafunction information. You can also create and save your own templates.

For more information about editing Quartus II Text Editor files, refer to *Editing Quartus II Text Editor Files* in Quartus II Help. For more information about text editors, refer to *About the Quartus II Text Editor* in Quartus II Help. For more information about the Quartus II Text Editor options and setting a preferred text editor, refer to *Setting Quartus II Text Editor Options* in Quartus II Help. For more information about the Quartus II language template feature, refer to the "Quartus II Language Templates" section in the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Creating Assignments

The **.qsf** stores all assignments created in the Quartus II software as Tcl commands. The Quartus II Default Settings File (**.qdf**) contains all the project and assignment default settings from the current version of the Quartus II software.

Quartus II Settings File

The **.qsf** is a text-based file containing Tcl commands and comments. The **.qsf** is not a Tcl script and does not support the full Tcl scripting language.

As you create assignments in the Quartus II software, you can choose either to store the assignments in memory temporarily or write the assignment out to the **.qsf** with the **Update assignments to disk during design processing only** option located on the **Processing** page of the **Options** dialog box. You can open the **Options** dialog box by clicking **Options** on the Tools menu. If you turn on the **Update assignments to disk**

during design processing only option, all assignments are stored in memory and are written to the **.qsf** when a compilation starts or when you save or close the project. By saving assignments to memory, the performance of the software improves because unnecessary reading and writing to the **.qsf** on the disk is not done. This performance improvement is seen when the project files are stored on a remote data disk.

You can add lines of comments in the **.qsf**, as shown in Example 4–2:

Example 4–2.

```
# 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
```

You can source another **.qsf** by typing the Tcl command shown in Example 4–3:

Example 4–3.

```
source <filename>.qsf ↵
```

For more information, refer to the *Quartus II Settings File Reference Manual*.

Preserving QSF Format

The Quartus II software maintains the order of assignments in the **.qsf**. When you create new assignments, the assignments are appended to the end of the **.qsf**. If you modify an assignment, the corresponding line in the **.qsf** changes to maintain the order of assignments in the **.qsf**, 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 appended to the end of the **.qsf**. For example, if you add a new design file to the project, the list of all your design files are appended to the end of the **.qsf**.

The header at the beginning of the **.qsf** is written only if the **.qsf** is newly created.

The Quartus II software preserves all spaces and tabs for all unmodified assignments and comments. When you create a new assignment or modify an existing assignment, the assignment is written using the default formatting.

Quartus II Default Settings File

You can ensure consistent results when defaults change between versions of the Quartus II software with the **assignment_defaults.qdf**, located in the **bin** directory of the Quartus II installation path.

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 the project directory
3. <*revision name*>**_assignment_defaults.qdf** from the project directory
4. <*revision name*>**.qsf** from the 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.

Using Revisions with Your Design

In the Quartus II software, a revision is a set of assignments and settings. A project may have multiple revisions, and each revision has its own set of assignments and settings. You can create multiple revisions in a project, and you can create a unique revision based on an existing revision. Creating a unique revision allows you to optimize a design for different results; creating a revision based on an existing revision allows you to try new settings and assignments and then compare the revisions.

Creating a revision of your design 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 perform the following tasks with revisions:

- Create a unique revision not based on a previous revision. Creating a unique revision allows you to optimize a design for different fundamental reasons, such as to optimize by area in one revision and then optimize for f_{MAX} in another revision. When you create a unique 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 includes all the assignments and settings in the existing 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 features in the Quartus II software.

Creating a Revision, Specifying the Current Revision, and Comparing Revisions

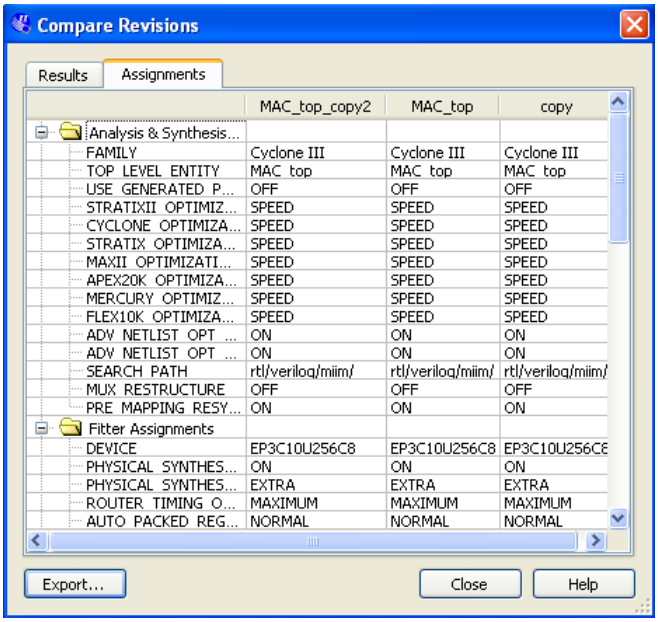
You can manage revisions with the **Revisions** dialog box that allows you to create and delete a revision, specify the current revision, and compare revisions.

The **.qsf** stores all Quartus II software assignments and settings. The Quartus II software creates a new **.qsf** and adds the name of the new revision to the list of revisions in the **.qsf** each time you create a new revision of a project. The name of a new **.qsf** matches the revision name.

You can compare the compilation results of multiple revisions side by side with the **Compare Revisions** (Figure 4–1) dialog box. The **Compare Revisions** dialog box compares the compilation results of each revision in three assignment categories:

- Analysis & Synthesis
- Fitter
- Timing Analyzer

Figure 4–1. Compare Revisions Dialog Box



In addition to viewing the compilation results of each revision, you can also view the assignments for each revision. Viewing the compilation results and assignments for each revision allows you to gain a better understanding of how different optimization options affect your design.

For more information about the steps to create a revision, specify the current revision, and compare revisions, refer to *Creating Project Revisions* in Quartus II Help.

Deleting a Revision

To delete a revision, follow these steps:

1. On the Project menu, click **Revisions**.
2. 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.

Creating New Copies of Your Design

Managing different copies of design files in a large project is 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 with the **Copy Project** command directs the Quartus II software to copy all your design files, your **.qsf**, and all your associated revisions (all assignments and settings).

If your design requires that you have two separate copies of your project, rather than just a separate revision, you can create a second copy of your project with the **Copy Project** command. For example, if you have a design that is compatible with a 32-bit data bus and you require a new copy of your design to interface with a 64-bit data bus, you may want a separate copy of the project. To create a new copy of your project, follow these steps:

1. On the Project menu, click **Copy Project**. The **Copy Project** dialog box appears.
2. In the **Destination directory** box, specify the path to your new project.
3. In the **New project** name box, type the new project name.
4. To open the new project immediately, turn on the **Open new project. (This option closes the current project)** option.
5. Click **OK**.

If you are creating a new copy of a project that contains an EDIF Input File (**.edf**) or a Verilog Quartus Mapping File (.**vqm**) from a third-party EDA synthesis tool, first create a copy of your project and then replace any **.edf** or **.vqm** files with the newly generated **.edf** or **.vqm**.

The **Copy Project** feature copies files listed in the **Source Control** file set for the Archiver. If you cannot find your source file in the **Source Control** file set, Altera recommends adding the source file into your project before copying.

A version is a Quartus II project that includes one set of design files and one or more revisions. The **Copy Project** command allows you to create multiple versions of a project. Creating a new version of your project allows you to edit a copy of your design files while preserving the original functionality of your design in another copy.

Archiving Projects

A single project can contain hundreds of files in many directories, which can make transferring a project between engineers difficult. To more easily share projects between engineers or to transfer your project to a new version of the Quartus II software, you can archive your project with the tools in the Quartus II software. Archiving your project creates a single compressed Quartus II Archive File (**.qar**) that contains all your design, project, and settings files. The **.qar** contains all the files, including the Quartus II Default Settings File (**.qdf**), required to compile your design and restore the original compilation results. The **.qdf** contains all the project and assignment default settings from the current version of the Quartus II software. When you restore the archive in a different version of the Quartus II software, you must include the **.qdf** in the archive to preserve your results. For more information about the **.qdf**, refer to "Quartus II Default Settings File" on page 4–4.

For more information on archiving a project and restoring an archived project, refer to *Archiving Projects* in Quartus II Help.

Archiving a Project

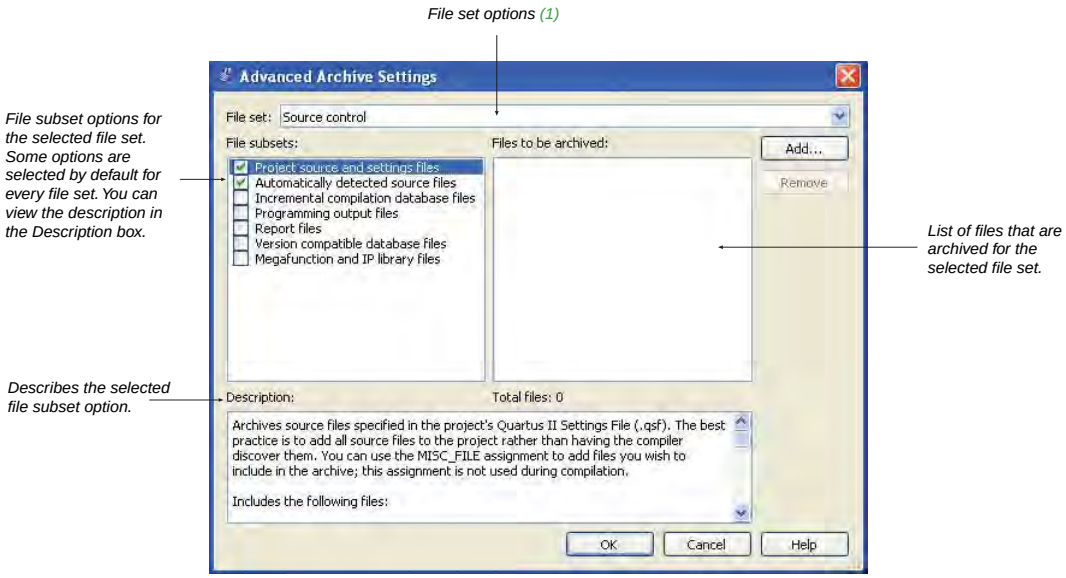
To archive a project, follow these steps:

1. On the Processing menu, point to **Start,** and then 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.

2. On the Project menu, click **Archive Project**.
3. In the **Archive file name** box, specify the name of the **.qar**.
4. Click the **Advanced** button for more archiving options. You can choose the type of files you want to archive by selecting the file set from the list in the **Advanced Archive Settings** dialog box (Figure 4–2). The default **Source control** file set consists all project source files and settings files. The default archive does not include the compilation results, or post-compilation netlists for incremental compilation. However, all file set options include the imported Quartus II Exported Partition Files (**.qxp**) if you perform incremental compilation.

Figure 4–2. Advanced Archive Settings Dialog Box



Note to Figure 4–2:

(1) For more information about file set options, refer to Table 4–1 on page 4–9.

Altera recommends that you include all source files in your Quartus II project rather than directing the Compiler to find them. Include all your source files in the **.qsf**. If all source files are not specified in your project, you can add them manually to the archive.

5. Click **OK**.
6. Click **Archive**.

Beginning with the Quartus II software version 9.0, the Quartus II software does not support archiving a full database (**\db**) folder.

If you are using a new device family with advanced support, a version-compatible database might not be available, and the archive option does not include a full compilation database. To archive your Quartus II project so that you can reproduce the compilation results in the same Quartus II version, type the following command-line option to archive a full database, as shown in Example 4–4:

Example 4–4.

```
quartus_sh --archive -use_file_set full_db [-revision <revision name>] <project name> ↵
```

Table 4–1 shows the predefined file sets available for archiving.

Table 4–1. File Sets (Part 1 of 2)

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 **.qxp** files for incremental compilation, because top-level projects do not include source files for imported partitions.
Source control with incremental compilation database	This file set includes files from the **Source control** settings, plus the incremental compilation database files when you compile your design with the **Full incremental compilation** option turned on. This file set includes the incremental compilation results, so you can preserve post-synthesis or post-fitting netlists from the original project database. When you restore an archive containing these post-compilation netlists, the subsequent compilation is much faster because the compilation only stitches the netlist partitions. Performing a compilation is necessary to analyze or view full compilation results in tools such as the Chip Planner. The archived file size for this setting is large. If the **Full incremental compilation** option is turned off, this file subset does not archive any files.
Service request	This file set includes files from the **Source control** settings, the report files, and the programming output files. This file set is useful when you attach your design files to a service request with Altera technical support, and for debugging purposes.

Table 4–1. File Sets (Part 2 of 2)

File Set	Description
Service request with incremental compilation database	This file set includes files from the **Service request** settings and the incremental compilation database files when you compile your design with the **Full incremental compilation** option turned on. This file set includes the incremental compilation results, so you can preserve post-synthesis or post-fitting netlists from the original project database. When you restore an archive containing these post-compilation netlists, the subsequent compilation is much faster because the compilation only stitches the netlist partitions. Performing a compilation is necessary to analyze or view full compilation results in tools such as the Chip Planner. The archived file size for this setting is large. If the **Full incremental compilation** option is turned off, this file subset does not archive any files.
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 predefined file set, the file set selection changes to **Custom**.

If you use external version control tools, you can refer to the **Source control** file set to determine the files required for a full compilation. If you want to add additional files to or exclude certain files from the archive, click the **Add** and **Remove** buttons in the **Advanced Archive Settings** dialog box. You can also modify the files added to the archive with the command shown in Example 4–5:

Example 4–5.

```
quartus_sh --archive -list_files [-ascii <output file name>]-revision <revision> <project name> ↵
```

Use the `-ascii` option if you want to output the file list to a file other than the standard output file. Use the `-ascii` option followed by a file name to write a list of the contents to a file.

For Quartus II Command-Line and Tcl API Help details about archiving or restoring a project, you can use the commands shown in Example 4–6:

Example 4–6.

```
quartus_sh --help=archive ↵
quartus_sh --help=restore ↵
```

Restoring an Archived Project

To restore an archived project, follow these steps:

1. On the Project menu, click **Restore Archived Project**.
2. In the **Archive file name** box, specify the **.qar** you wish to restore.
3. In the **Destination folder** box, specify the directory path in which you wish to restore the contents of the **.qar**.
4. Click **OK**.
5. If you did not include the compilation database files in the project archive, 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, follow these steps:

a. Delete the megafunction files located in the *<drive>:<restored_project>***/megafunctions** directory.

b. Delete the ***.cbx.xml** file located in the *<drive>:<restored_project>* directory. The ***.cbx.xml** files are revision-based. Altera uses the XML format to describe data in different file types, including revision reports, design metadata, component declarations, ports and nodes assignments, and device utilization.

To view the content of the **.qar** without restoring the archived file, type the command shown in Example 4–7:

Example 4–7.

```
quartus_sh --restore -content <.qar name> ↵
```

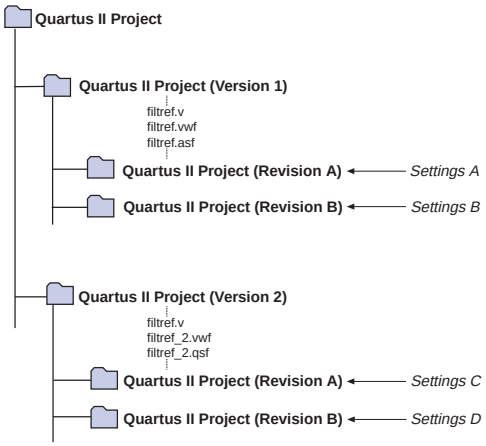
Exporting and Importing Version-Compatible Database Files

The Quartus II software generates version-compatible database files that are a representation of the internal database files. The Quartus II software allows you to export and import version-compatible database files for a project to use compilation databases in different versions of the Quartus II software. If you export version-compatible database files for a project, you can import these files in a future version of the Quartus II software. By importing version-compatible database files and rerunning timing analysis, you can check a project's fitting and timing results in newer versions of the Quartus II software.

Using version-compatible databases allows you to continue to use your same project database when you upgrade to a newer version of the Quartus II software, eliminating the need to recompile your project, which saves design time.

Figure 4–3 shows the Quartus II software version-compatible database structure.

Figure 4–3. Quartus II Version Compatible Database Structure



The version-compatible database feature is not available for all device families. For a list of supported device families, refer to *Exporting and Importing Version-Compatible Database Files* in Quartus II Help. If you require the database files to reproduce the compilation results in the same Quartus II software version, you can use the command-line option to archive a full non-version compatible database. For more information, refer to "Archiving Projects" on page 4–7.

Migrating to a New Version

To migrate a design from one Quartus II software version to a newer version, follow these steps:

1. On the File menu, click **Open Project** and browse to select the Quartus II project file to open the older version of the Quartus II software project.
2. On the Project menu, click **Copy Project** to create a new copy of the project. The older version closes and the copied project opens.
3. Before exporting the database, you must run Analysis and Synthesis for the new version. On the Project menu, click **Export Database**. By default, the database is exported to the **export_db** directory of the copied project. You can also create a new directory.
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.

Saving the Database in a Version-Compatible Format

To save the database in a version-compatible format during a full compilation, follow these 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**.

You can also export a project database as version-compatible database files during a full compilation. For more information on importing and exporting version-compatible databases, refer to *Exporting and Importing Version-Compatible Database Files* in Quartus II Help.

Quartus II Project Platform Migration

When moving your project from one computing platform to another, you must consider the following cross-platform issues:

- "File Names and Hierarchies"
- "Specifying Libraries"
- "Quartus II Search Path Precedence Rules"
- "Quartus II-Generated Files for Third-Party EDA Tools"
- "Migrating Database Files Between Platforms"

File Names and Hierarchies

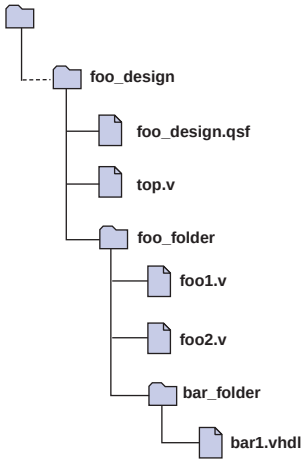
To ensure a successful migration across platforms, you must consider a few basic differences between operating systems when naming source files, especially when interacting with these from the command prompt or a Tcl script:

- 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. Use lowercase letters when naming files.
- Use a character set common to all the used platforms.
- Do not change the forward-slash (/) and back-slash (\) path separators in the **.qsf** 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 using spaces in the name of the project directory. 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 example, for a project titled **foo_design** with a directory structure shown in Figure 4–4, specify the source files as: **top.v, foo_folder/foo1.v**, **foo_folder/foo2.v**, and **foo_folder/bar_folder/bar1.vhdl**.

Figure 4–4. All-Inclusive Project Directory Structure

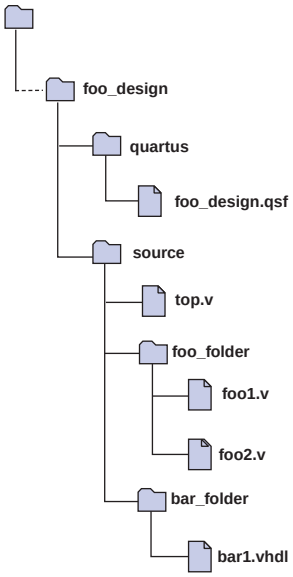


If the **.qsf** is in a directory that is separate from the source files, you can specify paths using the relative and absolute paths and libraries options.

Relative Paths

If the source files are very near to the Quartus II project directory, you can express relative paths using the . . notation. For example, in the directory structure shown in Figure 4–5, you can specify **top.v** as **../source/top.v** and **foo1.v** as **../source/foo_folder/foo1.v**.

Figure 4–5. Quartus II Project Directory Separate from Design Files



When you copy a directory structure to a different platform, ensure that all the 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:

- Project libraries—Apply to a specific project
- Global libraries—Apply to all projects

Use the procedures in this section to specify project or global libraries. The project directory takes precedence over the project libraries (for example, **foo1.v** is located in the project directory).

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 project library in the project, the **foo1.v** file can be specified in the **.qsf** as **foo1.v**. The Quartus II software searches directories that are specified as libraries and finds the file.

Specifying Project Libraries

To specify project 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. The **.qsf** of the current revision stores project libraries.

You can also specify project libraries in the **Libraries** page in the **General** category in the **Options** dialog box.

Specifying Global Libraries

To specify global libraries from the GUI, on the Tools menu, click **Options** and select **Libraries.** Type the name of the directory in the **Global Library name** box, or browse to the name of the directory. The **quartus2.ini** file stores global libraries.

To specify libraries from the GUI, on the Assignments menu, click **Settings** and select **Libraries.**

For PC users, the Quartus II software searches for the **quartus2.ini** file in the following directories and order:

1. USERPROFILE, for example, **C:\Documents and Settings**<*user name*>
2. Directory specified by the **TMP** environmental variable
3. Directory specified by the **TEMP** environmental variable
4. 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** 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. Alternatively, you can also select from the **Libraries** list and double-click to edit the path.

When copying projects that specify project libraries, you must either copy your project library files along with the project directory or ensure that your project library files exist in the target platform.

Quartus II Search Path Precedence Rules

If two files have the same file name, the Quartus II software's search path precedence rules determine the found file. 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**.
2. The project's database (**db)** directory.
3. Project libraries are searched in the order specified by the **SEARCH_PATH** setting of the **.qsf** 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 project 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 4–24 on page 4–27.

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 4–27.

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:

- 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**)
- IBIS Output Files (**.ibs**)

When you archive your 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 new device families, a version-compatible database might not be available. In this case, the archive does not include the compilation database. If you require the database files to reproduce the compilation results in the same Quartus II software version, you can use the command-line option to archive a full database. For more information, refer to "Archiving Projects" on page 4–7.

For more information about saving the database in a version-compatible format and archiving projects, refer to "Saving the Database in a Version-Compatible Format" on page 4–12 and "Archiving Projects" on page 4–25.

To copy your project to another platform, you can regenerate the output netlist or output files by following these steps:

1. Import the version-compatible database. For more information, refer to "Migrating to a New Version" on page 4–12.
2. From the Processing menu, run the Classic Timing Analyzer or TimeQuest analyzer.
3. Run the EDA Netlist Writer.

To restore your project, you can regenerate the output netlist or output files by performing the following steps:

1. Restore your design project. For more information about restoring an archived project, refer to "Restoring an Archived Project" on page 4–10.
2. Import the version-compatible database. For more information about migrating to a new version, refer to "Migrating to a New Version" on page 4–12.
3. From the Processing menu, run the Classic Timing Analyzer or TimeQuest analyzer.
4. Run the EDA Netlist Writer.

When you create version-compatible databases, you do not need to recompile your design as you move across platforms.

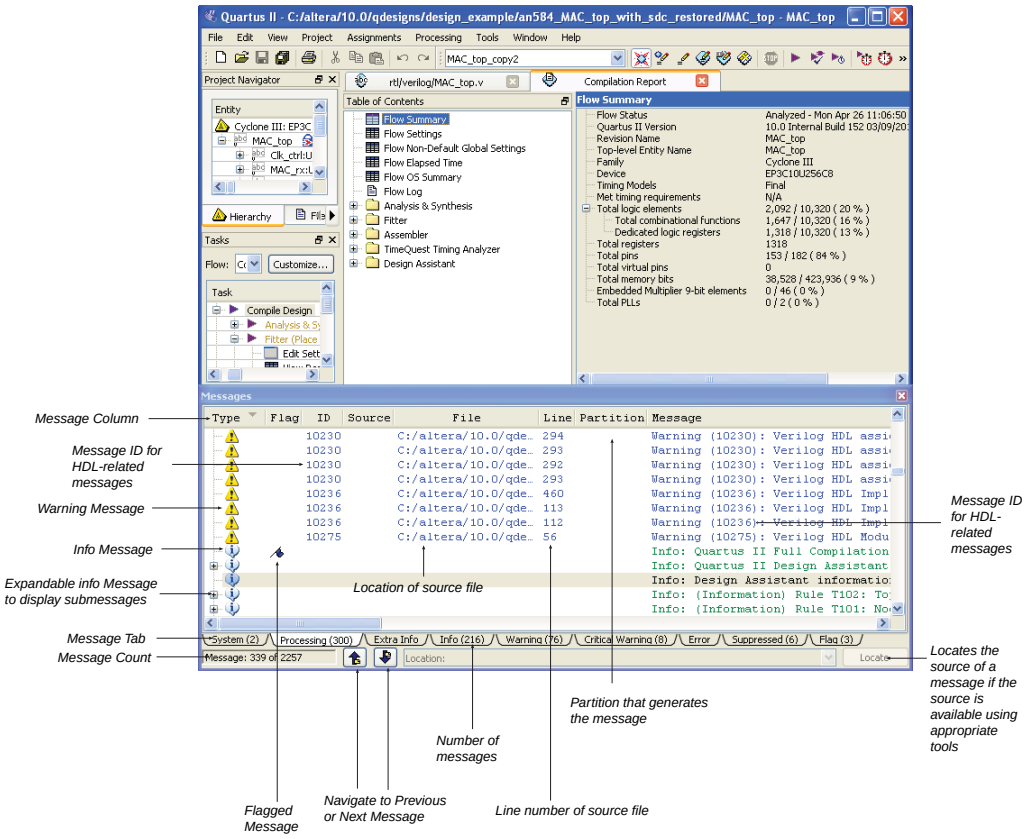
Migrating Database Files Between Platforms

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, Extra Info, Critical Warning, and Error messages. Some messages include information about software status during a compilation and alert you to possible problems with your design. The Messages box in the Quartus II GUI (Figure 4–6) displays messages, and these messages are written to `stdout` when you use command-line executables. In both cases, Quartus II report files write messages.

Figure 4–6. Viewing Quartus II Messages



You can right-click 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 messages. The following sections describe the Quartus II software features to help you manage messages.

Messages Window

The Messages window displays nine 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 generated when the Quartus II software processes your most recent compilation and simulation; timing analysis messages appear as part of the compilation messages.
Extra Info		Displays detailed informational messages for designers; for example, extra fitting information.
Info		Displays general informational messages generated during a compilation, and simulation; for example, legality messages and successful compilation messages.
Warning		Displays strong warning messages generated during a compilation or simulation.
Critical Warning		Displays critical warning messages generated during a compilation or simulation.
Error		Displays processing and compilation error messages generated during a compilation or simulation. Error messages can sometimes stop processing and cannot be disabled.
Suppressed		Displays suppressed messages during the last processing operation.
Flag		Displays flagged messages. 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.

For more information about the Messages window, refer to *About the Messages Window* in Quartus II Help.

Hiding Messages

In the Messages window, you can hide all messages of a particular type. For example, to hide Critical Warning messages, follow these 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 (for example, point to **Hide** and click **Hide Critical Warning Messages**).

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.

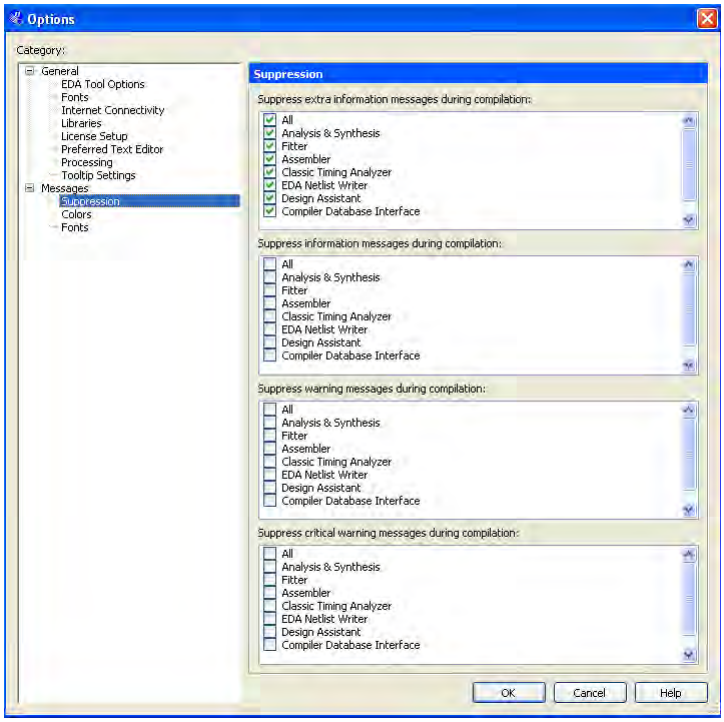
Message Suppression

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 displaying. 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.

Adding a message to be suppressed creates a suppression rule. 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** in the **Messages** section (Figure 4–7).

Figure 4–7. 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 the Message Suppression Manager, refer to "Message Suppression Manager" on page 4–22.

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 **Suppress Exact Selected Messages** option, only messages that match the exact text are suppressed during subsequent compilations. The **Suppress All Similar Messages** option behaves like a wildcard pattern on variable fields in messages and the **Suppress All Flagged Messages** option only suppresses flagged messages.

Example 4–8 shows an example of suppressing all similar messages:

Example 4–8.

```
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.

Example 4–9 shows an example of this message in Help:

Example 4–9.

```
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 suppression rule shown in Example 4–10:

Example 4–10.

```
Info: Found * design units, including * entities, in source file *.
```

As a result, all similar messages (ones that match the pattern) are suppressed.

Message Suppression Details and Limitations

The following rules describe which messages can be suppressed and how to suppress them:

- 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.

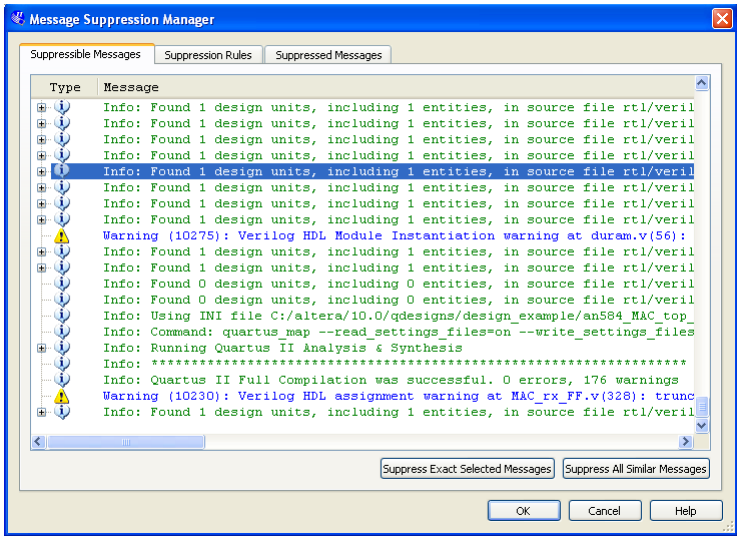
- Messages are suppressed on a per-revision basis, not for an entire project. Information about which messages to suppress is stored in a file called *<revision>*.**srf**. If you create a revision based on a revision for which messages are suppressed, the suppression rules file is copied to the new revision. You cannot make all revisions in one project using 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 the Message Suppression Manager to view and suppress messages, view and delete suppression rules, and view suppressed messages.

Right-click in the Messages window and click **Message Suppression Manager** from the Suppress sub-menu. The Message Suppression Manager has three tabs labeled **Suppressible Messages**, **Suppression Rules**, and **Suppressed Messages** (Figure 4–8).

Figure 4–8. Message Suppression Manager Dialog Box



Suppressible Messages

The **Suppressible Messages** tab lists messages that are 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 4–21. You can select all similar messages to view suppressed messages 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. On 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 an **.srf** containing the suppression rules listed in the **Suppression Rules** tab for the current revision

An entry in the **Suppression Rules** tab that includes a message with submessages indicates the submessage is suppressed only when all its parent messages match.

You can stop suppressing messages by deleting the suppression rules that match them (causing them to be suppressed). Deleting suppression rules does not cause the suppressed messages to be added to the messages generated during the previous compilation; you must recompile your 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.

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, a **Suppressed Messages** entry in the Quartus II compilation report—viewable in the GUI—includes suppressed messages. 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**.

Managing Projects in a Team-Based Design Environment

The Quartus II software provides several methods to help you manage efficient design coordination across multiple designers and design iterations. You can use the following features to preserve and track project changes:

- Creating Revisions
- Managing Different Design Versions
- Creating Version-Compatible Databases
- Archiving Projects

For more information about creating revisions, managing different design versions, creating version-compatible databases, and archiving projects in a team-based design environment, refer to *About Project Management* in Quartus II Help.

The Quartus II software supports top-down incremental compilation flows. 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. However, the project lead compiles and optimizes the top-level project as a whole. Completed parts of the design can have fitting results and performance fixed as other parts of the design change.

For more information about incremental compilation for 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*. For more information about best practices for incremental compilation partitions and floorplan assignments, refer to *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 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 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 command shown in Example 4–11 at the command prompt:

Example 4–11.

```
quartus_sh --qhelp ↵
```

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook* and *API Functions for Tcl* in Quartus II Help. 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*.

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 "Using Revisions with Your Design" on page 4–5.

Creating Revisions

The `-based_on` and `-set_current` options are optional. You can also use `-copy_results` option to copy results from the "based_on" revision. Example 4–12 shows a Tcl command creating a new revision called **speed_ch**, based on a revision called **chiptrip**, and sets the new revision as the current revision:

Example 4–12.

```
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. Type the Tcl command shown in Example 4–13 to specify the current revision:

Example 4–13.

```
set_current_revision -force <revision name> ↵
```

Getting a List of Revisions

Type the Tcl command shown in Example 4–14 to get a list of revisions in the opened project:

Example 4–14.

```
get_project_revisions <project_name> ↵
```

Deleting Revisions

Type the Tcl command shown in Example 4–15 to delete a revision:

Example 4–15.

```
delete_revision <revision name> ↵
```

Archiving Projects

You can archive projects with a Tcl command or a command run at the system command prompt.

The Tcl command shown in Example 4–16 creates a project archive with the default settings and overwrites the specified archived file if it exists:

Example 4–16.

```
project_archive archive.qar -overwrite ↵
```

You can change default settings with the `project_archive` command with options such as:

- `-all_revisions`
- `-include_libraries`
- `-include_outputs`
- `-use_file_set` *<file_set>*
- `-version_compatible_database`

For new device families, a version-compatible database might not be available. In this case, the archive does not include the compilation database. If you require the database files to reproduce the compilation results in the same Quartus II software version, you can use the `-use_file_set full_db` command-line option to archive a full database. For more information, refer to "Archiving Projects" on page 4–7.

To create a project archive called **top,** type the command shown in Example 4–17 at a command prompt:

Example 4–17.

```
quartus_sh --archive top ↵
```

You can overwrite the existing archive file with the `-overwrite` option.

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 "Restoring an Archived Project" on page 4–10.

The Tcl command in Example 4–18 restores the project archive named **archive.qar** in the **restored** subdirectory and overwrites existing files:

Example 4–18.

```
project_restore archive.qar -destination restored -overwrite ↵
```

To restore a project archive, type the command shown in Example 4–19 at a command prompt:

Example 4–19.

```
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 "Exporting and Importing Version-Compatible Database Files" on page 4–11.

The `flow` and `database_manager` packages contain commands to manage version-compatible databases.

Type the Tcl commands shown in Example 4–20 from the `database_manager` package to import or export version-compatible databases.

Example 4–20.

```
export_database <directory> ↵
import_database <directory> ↵
```

Example 4–21 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 4–21.

```
set_global_assignment -name VER_COMPATIBLE_DB_DIR <directory>
execute_flow –flow export_database
execute_flow –flow import_database
```

Example 4–22 shows the Tcl commands to generate version-compatible databases after every compilation.

Example 4–22.

```
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 4–23).

Example 4–23.

```
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 project libraries. To specify project libraries, use the `set_global_assignment` command. To specify global libraries, use the `set_user_option` command. Example 4–24 shows the typical usage of the `set_global_assignment` and `set_user_option` commands.

Example 4–24. Commands to Specify Project 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 project 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. In Example 4–25, the Tcl script outputs the user paths and global libraries for an open Quartus II project.

Example 4–25. Commands to Report Specified Project 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. The 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 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.

Document Revision History

Table 4–3 shows the revision history for this chapter.

Table 4–3. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Major reorganization done to this chapter. ■ Updated "Working with Messages" on page 4–18. Added a link to Help. ■ Removed Figure 4–2 on page 4–7, Figure 4–11 on page 23, and Figure 4–12 on page. ■ Updated "Specifying Libraries" on page 4–15 section. Changed "User Libraries" to "Libraries". ■ Removed "Reducing Compilation Time" on page 4–26. ■ Added "Managing Projects in a Team-Based Design Environment" on page 4–23 and "File Association" on page 4–2. ■ Updated Figure 4–1 on page 4–6, Figure 4–2 on page 4–8, Figure 4–6 on page 4–18, Figure 4–7 on page 4–20, and Figure 4–8 on page 4–22.
November 2009	9.1.0	■ Updated "Creating a New Project" on page 4–4, "Archiving a Project" on page 4–9, "Restoring an Archived Project" on page 4–11. ■ Added "Quartus II Text Editor" on page 4–2, "Reducing Compilation Time" on page 4–32. ■ Updated Table 4–1 on page 4–10, Table 4–2 on page 4–20. ■ Updated Figure 4–4 on page 4–9, Figure 4–7 on page 4–19.
April 2009	9.0.0	Updated to fix "Document Revision History" for version 9.0.0.

Table 4–3. Document Revision History

Date	Version	Changes Made
March 2009	9.0.0	■ Updated "Managing Quartus II Projects" on page 4–1, "Creating a New Project" on page 4–2, "Using Revisions with Your Design" on page 4–3, "Creating and Deleting Revisions" on page 4–4, "Creating New Copies of Your Design" on page 4–6, "Version-Compatible Databases" on page 4–11, "Quartus II Project Platform Migration" on page 4–12, "Filenames and Hierarchies" on page 4–12, "Quartus II Search Path Precedence Rules" on page 4–15, "Quartus II-Generated Files for Third-Party EDA Tools" on page 4–15, "Migrating Database Files between Platforms" on page 4–16, "Message Suppression" on page 4–20, "Quartus II Settings File" on page 4–24, "Quartus II Default Settings File" on page 4–25, "Managing Revisions" on page 4–26, "Archiving Projects" on page 4–26 and "Archiving Projects with the Quartus II Archive Project Feature" on page 4–7, "Importing and Exporting Version-Compatible Databases" on page 4–27, "Specifying Libraries Using Scripts" on page 4–28, "Conclusion" on page 4–30. ■ Updated Figure 4–1, Figure 4–7, Figure 4–8, and Figure 4–11. ■ Updated Table 4–1 and Table 4–2. ■ Updated Example 4–3, Example 4–4, Example 4–5, and Example 4–6.
November 2008	8.1.0	Changed to 8½" × 11" page size. No change to content.
May 2008	8.0.0	Updated references.

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

Take an online survey to provide feedback about this handbook chapter.

Section II. I/O and PCB Tools

This section provides an overview of the I/O planning process, Altera 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 ways to use the Quartus II software to analyze signal integrity, including simultaneous switching noise, as well as interfaces with third-party PCB design tools.

This section includes the following chapters:

- Chapter 5, I/O Management

  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.

- Chapter 6, Simultaneous Switching Noise (SSN) Analysis and Optimizations

  This chapter describes the tools in the Quartus II software that allow you to estimate the SSN performance of your design both early in the design cycle and when your PCB is complete.

- Chapter 7, Signal Integrity Analysis with Third-Party Tools

  This chapter is intended for logic designers and board designers, and describes simulation and how to adjust designs to improve board-level timing and signal integrity. Also included is information about how to create accurate models of your design with the Quartus II software for use in simulation software.

- Chapter 8, Mentor Graphics PCB Design Tools Support

  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.

- Chapter 9, Cadence PCB Design Tools Support

  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.

5. I/O Management

QII52013-10.0.0

The process of managing I/O assignments involves more than 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. Both I/O capabilities of the target device and board layout guidelines influence pin location and other types of assignments. Therefore, it is necessary to begin I/O planning and PCB development even before starting to design for the target device.

Altera provides many resources for I/O planning. This chapter provides information on how to make create assignments, how to enter I/O interface information in the Pin Planner, how to create I/O-based top-level HDL files, how to validate your pin assignments, and how to generate a valid pin-out file for use with third-party PCB tools. 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.

For more information about the Altera resources available for I/O planning, refer to the I/O Management, Board Development Support, and Signal Integrity Analysis Resource Center of the Altera website.

For more information about PCB designs for Altera high-speed FPGAs, refer to *AN 315: Guidelines for Designing High-Speed FPGA PCBs*, and the Board Design Resource Center of the Altera website.

This chapter includes the following topics:

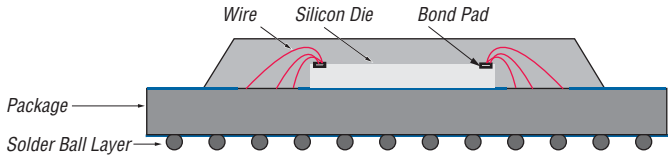
- "Understanding Altera Pin Terminology" on page 5–1
- "I/O Planning Overview" on page 5–5
- "Performing Early I/O Planning with the Pin Planner" on page 5–8
- "Importing and Exporting Pin Assignments" on page 5–11
- "Creating Pin-Related Assignments" on page 5–13
- "Validating Pin Assignments" on page 5–23
- "Performing I/O Timing Analysis" on page 5–33
- "Incorporating PCB Design Tools" on page 5–38

Understanding Altera Pin Terminology

Altera devices are available in a variety of package types. To describe 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 silicon to the I/O pins. 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 more information about the BGA packages available for each Altera device, refer to the *Altera Device Package Information Data Sheet*.

Figure 5–1. Wire Bond BGA

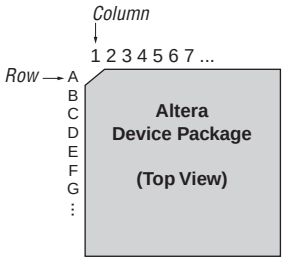


Package Pins

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 identified with pin numbers. The pin numbers are determined by pin locations using a coordinate system, with letters and numbers identifying the row and column of the pins, respectively.

Figure 5–2 shows the coordinate system used to identify pin locations. The top row of pins is labeled *A* and continues alphabetically as you move down. The left-most column of pins is labeled *1* and increments by one as you move right. For example, pin number *B4* represents the pin located in row *B* and column *4*. The letters I, O, Q, S, X, and Z are never used in pin numbers. If the device contains more rows than letters of the alphabet, the alphabet is repeated, prefixed with the letter *A*.

Figure 5–2. Row and Column Labeling

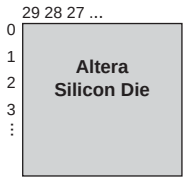


For more information about the pin numbers for your Altera device, refer to the *Pin-Out Files for Altera Devices* page of the Altera website.

Pads

Package pins are connected to pads located on the perimeter of the top metal layer of the silicon die (refer to Figure 5–1). Figure 5–3 shows the numbering scheme for pads on the device. Each pad is identified by a pad ID, which is numbered starting at 0, and increments by one in a counterclockwise direction around the device.

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. You must understand the pad locations to which your pins were assigned, because some pin placement rules describe pad placement restrictions. For example, to ensure signal integrity, in certain devices there is a restriction on the number of I/O pins supported by a VREF pad. 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 the 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 appropriate device handbook available on the Literature and Technical Documentation page of the Altera website.

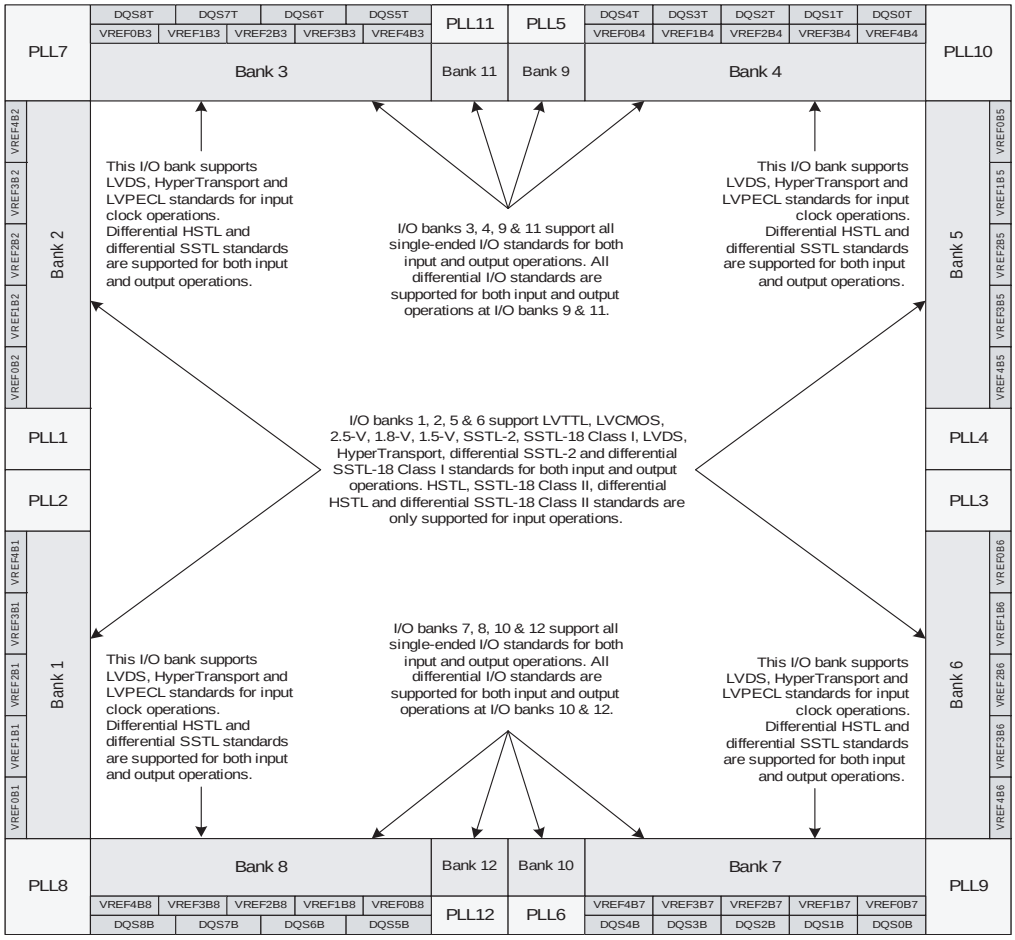
I/O Banks

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` pins, to offer the highest I/O performance. Depending on the device and I/O standards for the pins in 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, however the pins must use the same V_{CCIO} signal.

Figure 5–4 shows the I/O banks of a Stratix® II device. The pins in the I/O banks on the left and right side 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). Pins belonging to the same I/O bank must use the same V_{CCIO} signal.

For more information about the capabilities of each I/O bank, refer to the appropriate device handbook available on the Literature and Technical Documentation page of the Altera website.

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. For more information, 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 groups exist in an I/O bank. The pins in a VREF group share the same V_{CCIO} and V_{REF} voltages.

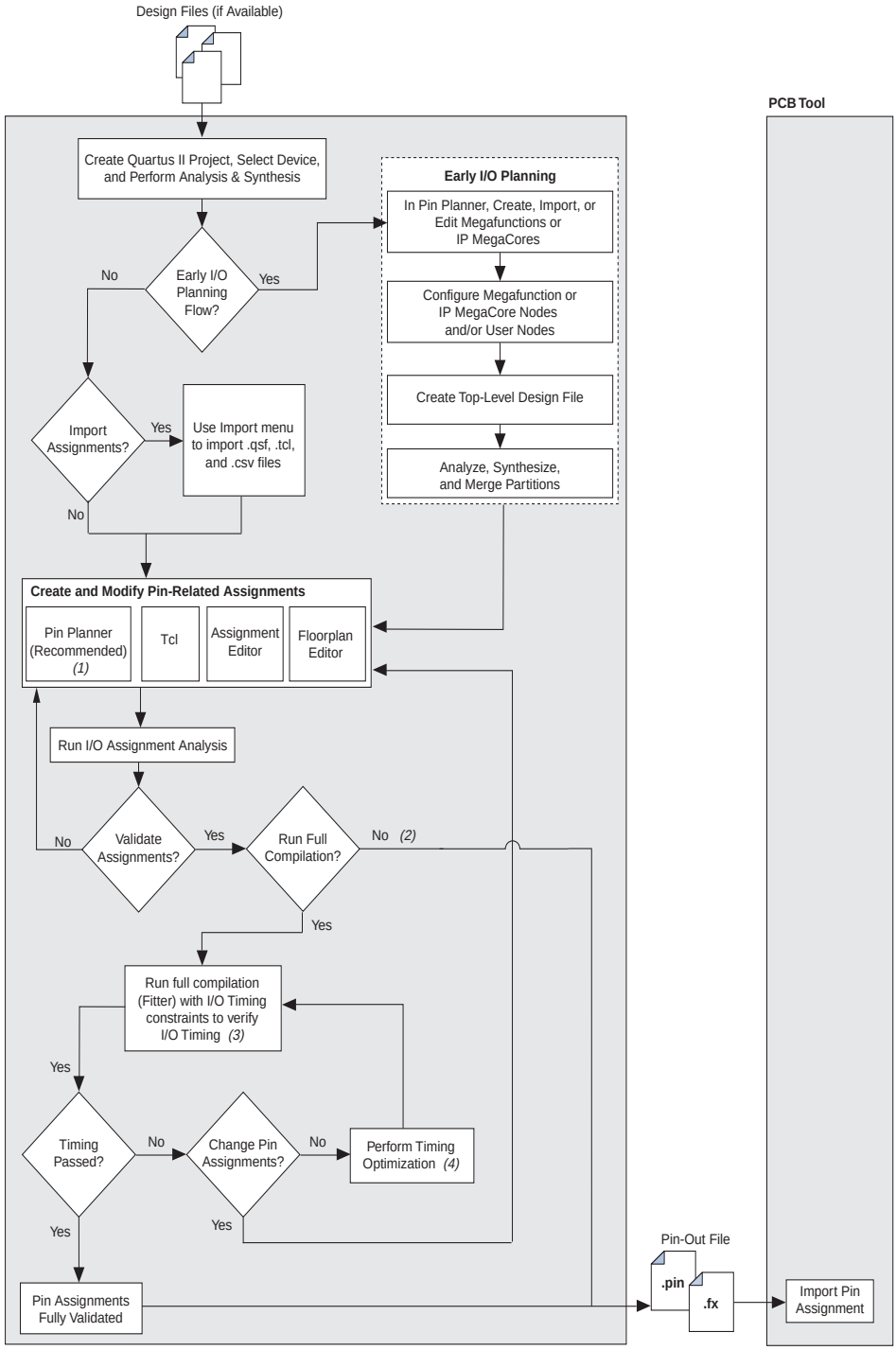
For more information about I/O banks, VREF groups, and supported I/O standards, refer to the appropriate device handbook available on the Literature and Technical Documentation page of the Altera website.

I/O Planning Overview

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 the I/O assignment analysis feature to assist you in I/O planning. The method you use to create I/O assignments depends on your design requirements. Figure 5–5 shows the recommended design flow for creating I/O assignments for a design. I/O planning for your design in the Quartus II software can include the following tasks:

- Selecting a device that meets your logic and I/O requirements, based on the supported I/O standards for the device, I/O bank structure, supply voltage requirements such as V_{REF} and V_{CCIO} requirements in I/O banks, available pins for user I/O, power supply requirements, and other design requirements.
- Preparing your design files. 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 (refer to Figure 5–5) to generate a top-level HDL wrapper file.
- Importing any existing assignments from a Tcl Script File (**.tcl**), Comma-Separated Value File (**.csv**), or Quartus II Settings File (**.qsf**).
- Creating, modifying, and completing all pin-related assignments, including pin location assignments, I/O standard assignments, output loading assignments for output and bidirectional pins, slew rate assignments, current strength assignments.
- Validating your pin-related assignments. You can perform preliminary I/O assignment validation while creating the assignments with the live I/O check feature; you can perform a more a thorough validation with the I/O assignment analysis feature; and finally you can perform complete I/O assignment validation by running the Fitter with timing constraints.
- Generating a validated Pin-Out File (**.pin**) for third-party PCB tools.

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** is created at the *<project_dir>* level. The **.fx** is created at the *<project_dir/board/.../>* level.

(3) You must complete your design files and constraints before you begin full compilation. To learn how to create I/O timing constraints, refer to *The Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

(4) For more information, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Selecting a Device

Before you begin I/O planning or I/O assignment analysis you must first select a device family that has an appropriate I/O structure, supports the correct I/O standards, has enough available pins for user I/O, supports the correct clocking schemes and options, and has an appropriate I/O bank structure for your design. Then, you must choose an appropriate device from a supported device family for your design.

For more information, refer to the various device handbooks available on the Literature and Technical Documentation page of the Altera website.

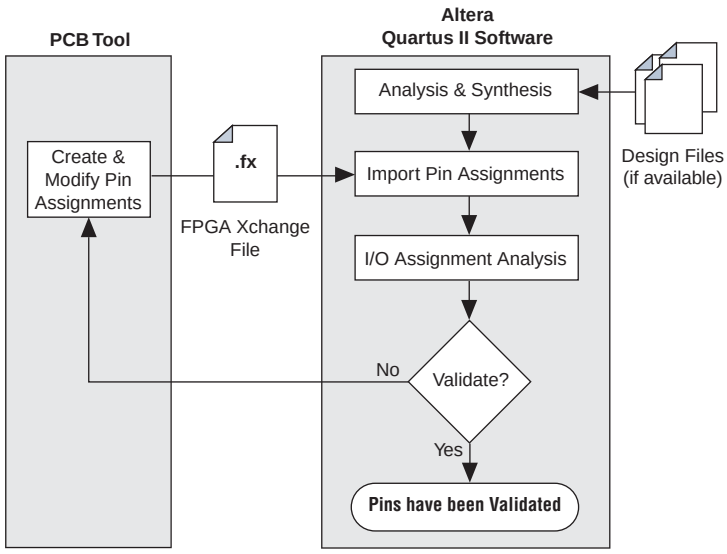
For more information about selecting a device in the Quartus II software, refer to *Setting Up and Running a Compilation* in Quartus II Help.

Working with Third-Party PCB Tools

If you have not designed your PCB, the recommended I/O planning flow (refer to Figure 5–5) indicates that you should first create and validate your I/O assignments in the Quartus II software, and then export them to the PCB tool. If your PCB is partially designed, Figure 5–6 shows the recommend I/O planning flow to ensure that your pin assignments are valid. First, you create your device assignments in your PCB tool and then import them into the Quartus II software for validation.

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



Performing Early I/O Planning with the Pin Planner

In design planning, typically you create design elements in an HDL such as Verilog HDL or VHDL, or in a schematic editor such as the Quartus II Block Editor. Central to the design is a top-level file that instantiates the next level of hierarchy and includes port names and their direction. Example 5–1 shows a top-level file, written in Verilog HDL, that lists the input and output ports in a design.

Example 5–1. Top-Level Design File

```
module top (      clk_in,
                  rst,
                  a,
                  z,
                  b,
                  c_in,
                  d,
                  e);

input clk_in;
input rst_;
input a;
input z;
input [7:0] b;
input [7:0] c_in;
input [7:0] d;
output reg [7:0] e;


/* Instantiations of sub blocks */

endmodule
```

Top-level design files often contain interfaces for memory, high-speed I/O, device configuration, and debugging tools. Listing the ports in an HDL or drawing them in the schematic can be extremely time-consuming. To reduce the time spent in creating top-level design files, you can use the Pin Planner to create a top-level design file if the design files for your entire project are not available or are incomplete. The interfaces between your target device and other devices are determined and documented in design specifications. By adding the interfaces required to connect your target device and other devices in your design directly in the Pin Planner, you can plan your I/O assignments efficiently without design files, and generate a top-level module in Verilog HDL or VHDL. By importing or creating MegaCore® functions or Altera megafunctions in the Pin Planner, as well as creating or adding additional top-level I/O information, the generated top-level design file accurately anticipates the rest of the HDL to come.

The following sections describe the typical steps of the early I/O planning flow:

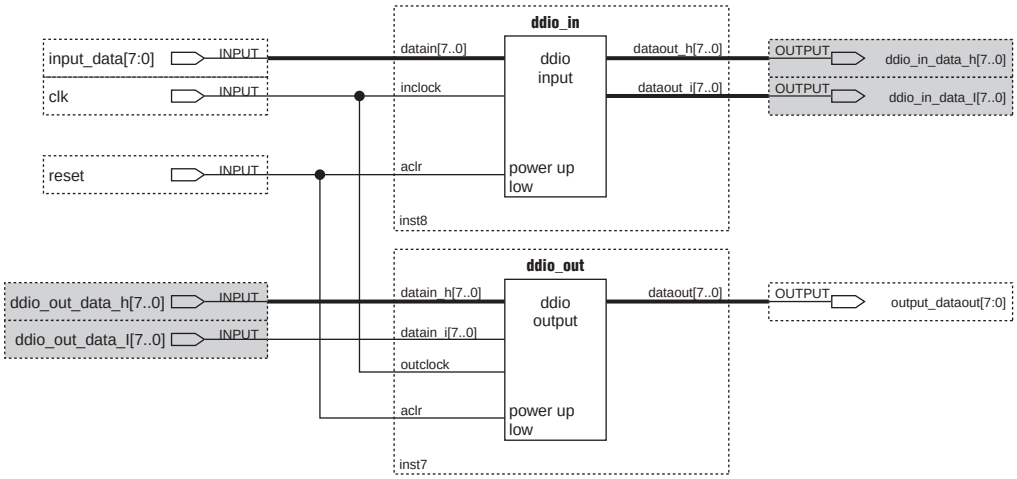
- "Instantiating or Importing IP Cores in the Pin Planner"
- "Adding and Connecting Nodes" on page 5–9
- "Setting Up and Creating the Top-Level Design File" on page 5–10

Instantiating or Importing IP Cores in the Pin Planner

You can use the MegaWizard™ Plug-In Manager from the Pin Planner to instantiate or import custom IP cores to assist you in I/O planning. Adding interface information directly in the Pin Planner allows you to assign required pins without manually creating each pin individually.

You can create complex interfaces from the Pin Planner. Figure 5–7 shows the megafunctions ddio_in and ddio_out in the schematic created by invoking the MegaWizard Plug-In Manager from the Pin Planner. The ports shown in gray are ports specified as internal ports, which are connected later in the design.

Figure 5–7. Connections between Input and Output Megafunctions and User Nodes



For more information about creating or customizing IP cores with the Pin Planner, refer to *Managing Pins and Groups in the Pin Planner* and *Create/Import Megafunction Dialog Box* in Quartus II Help.

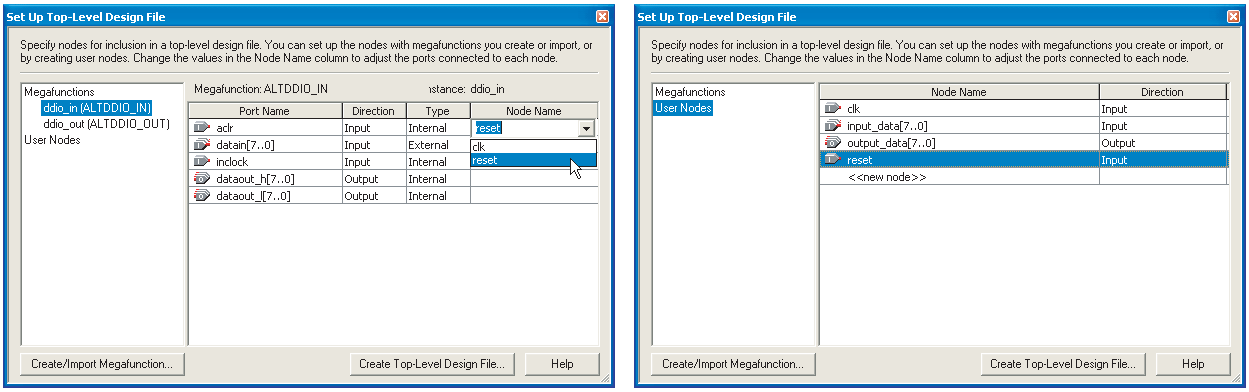
Adding and Connecting Nodes

After you instantiate an IP core in your design, you can set up the user nodes in the design and customize information such as port direction and type in the **Set Up Top-Level File** window. Before you create a top-level design file, you must connect the user nodes and ports to each other and to the rest of the design. You also can add new nodes to the design in the **Set Up Top-Level File** window.

The IP core nodes you specify as internal nodes in the **Set Up Top-Level Design File** dialog box are declared as virtual pins when you generate the top-level design file. The Pin Planner makes virtual pin assignments to internal nodes so that internal nodes are not assigned to device pins during compilation. These virtual pins are not shown in the **All Pins** list or **Groups** list in the Pin Planner because they are not actual external ports of the design. Any new nodes you add to the design in the **Set Up Top-Level Design File** dialog box are declared as ports in the top-level design file and are shown in the **All Pins** list and **Groups** list.

Figure 5–8 shows how to make the user node connection between the **reset** signal and the megafunction `reset` input port shown in Figure 5–7.

Figure 5–8. Connecting a User Node to a Megafunction Port



For more information about connecting nodes from IP cores, refer to *Generating a Top-Level Design File Based on Pin Planner Megafunctions and User Nodes* in Quartus II Help.

Setting Up and Creating the Top-Level Design File

You can create a top-level design file after you create pin connections and add or modify user nodes and IP core nodes with the Pin Planner. If the internal logic is incomplete, generating the top-level design file allows you to validate your I/O assignments and provides a basis on which to build the rest of your design.

You must update the top-level design file whenever you change the top-level ports of the design, including any node changes made in the **Set Up Top-Level Design File** window.

Example 5–2 shows a sample of a top-level HDL wrapper file representing the design in Figure 5–7.

Example 5–2. 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
```

For more information about generating a top-level design file, refer to *Generating a Top-Level Design File Based on Pin Planner Megafunctions and User Nodes* in Quartus II Help.

After you generate the top-level design file and compile the design, use I/O assignment analysis as described in "Validating Pin Assignments with I/O Assignment Analysis" on page 5–26 and continue with your design flow by modifying or creating pin assignments with the Pin Planner.

Importing and Exporting Pin Assignments

If you have existing pin assignments in the file formats **.tcl**, **.csv**, **.qsf**, **.fx**, or **.pin** from a different Quartus II project or from third-party PCB tools, you can transfer these assignments between the Quartus II software and other tools.

Importing and Exporting Assignments with the Quartus II Software

You can import and export pin-related assignments contained in **.tcl**, **.csv**, and **.qsf** files with the Quartus II software. You can import assignments from **.tcl**, **.csv**, and **.qsf** files and then view and change the assignments in the Pin Planner.

When you create pin assignments with the Pin Planner all your pin-related assignments are written to the **.qsf** as Tcl commands. You can import and export **.qsf** files between projects to transfer all assignments, including pin assignments.

For more information, refer to *Importing and Exporting Assignments* in Quartus II Help.

For more information about **.qsf** files, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*.

When you export pin assignments from the Pin Planner as a **.csv**, the row and column headings in the exported file is in the same order and format as the columns displayed 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.

When you export pin assignments as Tcl commands in a **.tcl**, you create a script you can later run to add the assignments as part of a scripted compilation flow.

For more information about importing and exporting pin assignments as **.tcl** and **.csv** files, refer to *Assigning Pins* in Quartus II Help.

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*.

Importing and Exporting Assignments with Third-Party PCB Tools

After creating and validating your pin assignments, you can export device and pin-related information from the Quartus II software to third-party PCB tools for board development. You must generate a **.pin** with the Quartus II software to export the correct pin locations and other important pin information.

The device **.pin** files display the pin name and pin number as well as detailed properties about each pin. Table 5–1 describes the columns in a **.pin**.

Table 5–1. .pin Header Description

Column Name	Description
Pin Name/Usage	The name of the design pin, or whether the pin is GND or V_{CC} pin
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 the pin
I/O Bank	The I/O bank to which the pin belongs
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, refer to the *Pin-Out Files for Altera Devices* page of the Altera website.

To transfer device and pin-related information between the Quartus II software and the Mentor Graphics I/O Designer software you must generate an **.fx**. Importing assignments into the I/O Designer software requires both the **.fx** and a **.pin** generated by the Quartus II software. However, the Quartus II software requires only the **.fx** to import pin assignments back from the I/O Designer software.

For more information, refer to *Importing and Exporting Assignments* in Quartus II Help.

For more information about the I/O Designer software and the DxDesigner interface, refer to the *Mentor Graphics PCB Tools Support* chapter in volume 2 of the *Quartus II Handbook*.

Creating Pin-Related Assignments

A pin-related assignment is any assignment applied to a top-level pin. For example, a pin location assignment 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 drive strength, or assigning a slew rate to a pin.

When making pin assignments, 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 pins you intend to use in the future, 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 I/O assignment analysis.

For more information about reserving pins with the Pin Planner, refer to *Assigning Pins* in Quartus II Help.

Table 5–2 describes the tools and features in the Quartus II software for creating reserved pins and other pin-related assignments. Each tool and feature is described in more detail in the following sections. Altera recommends that you use the Pin Planner to create and edit pin-related assignments; however, depending on your design flow, you may find some of the other tools useful for working with pin-related assignments.

Table 5–2. Overview of Quartus II Tools and Features to Create Pin-Related Assignments

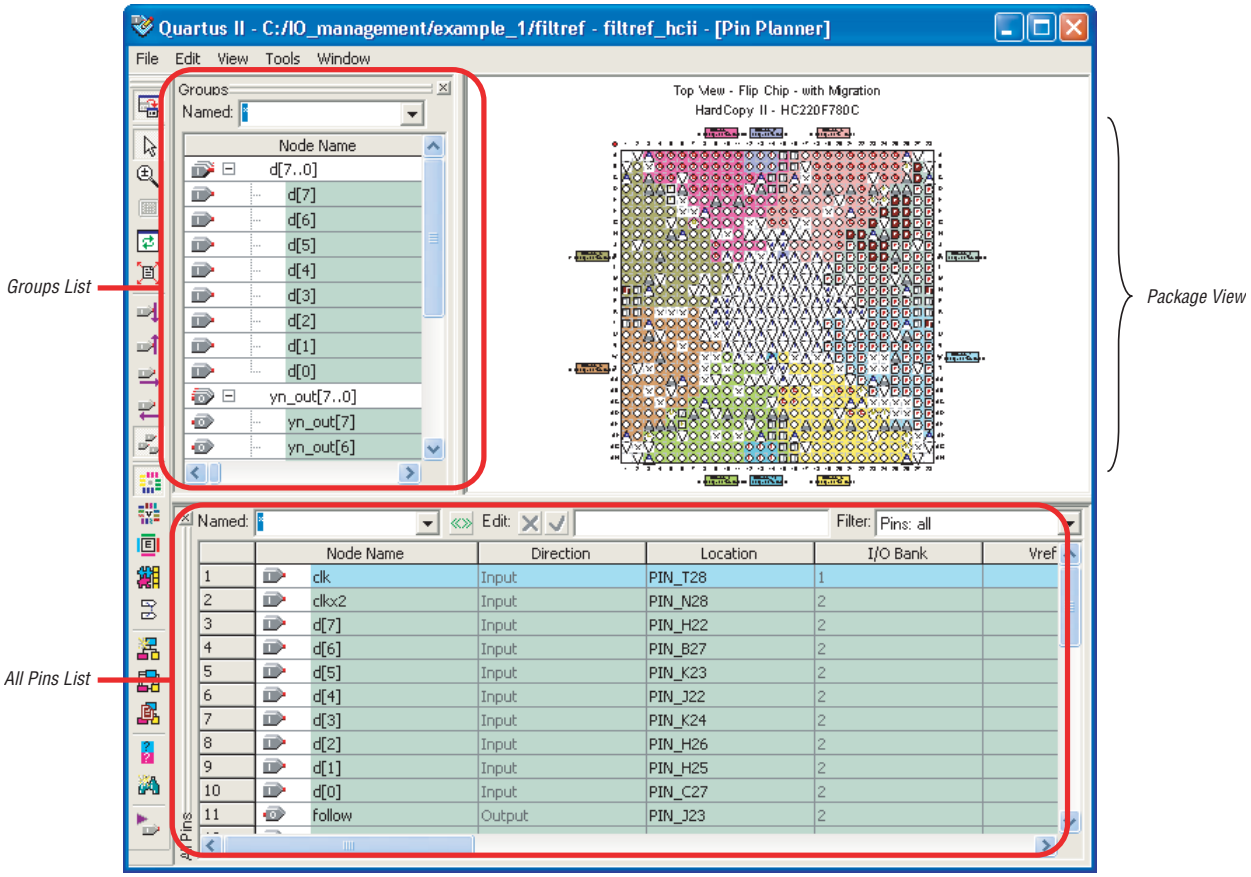
Feature	Overview
Pin Planner	■ Create pin location assignments to one or more node names by dragging and dropping unassigned pins into the package view ■ Edit pin location assignments for one or more node names by dragging and dropping groups of pins in the package view ■ Visually analyze pin resources in the package view ■ Display I/O banks, VREF groups, and differential pin pairs ■ View the function of package pins with the Pin Legend window ■ Make correct pin location decisions by referring to the Pad View window ■ Create, import, and edit IP cores for early I/O planning ■ Generate a top-level wrapper file without design files based on early I/O assignments ■ Configure board trace model assignments, instead of capacitive loading assignments, to generate Advanced I/O Timing results
Tcl Scripts	■ Create any pin-related assignments for multiple pins ■ Store and reapply all pin-related assignments with Tcl scripts ■ Create assignments from the command line
Chip Planner	■ 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 with attributes in the design files to pass assignments to the Quartus II software

Creating Pin Assignments With the Pin Planner

During I/O planning, it can be cumbersome to try to correlate pin numbers with their relative location on the package and their pin properties. The Pin Planner (refer to Figure 5–9) is an interface for creating and editing pin-related assignments. The Pin Planner provides an intuitive graphical representation of the target device to make it easy to plan your I/Os, create reserved pins, and create pin location assignments. 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 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.

For more information about the Pin Planner, refer to *About the Pin Planner* in Quartus II Help. For more information about creating pin assignments with the Pin Planner, refer to *Assigning Pins* in Quartus II Help.

Figure 5–9. Pin Planner



Finding Compatible Pin Locations with the Pin Finder

As device 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. As you edit pin assignments to help your design fit or to achieve timing goals, you can use the tools in the Pin Planner to help you find appropriate pin locations for your design I/O nodes.

For more information, refer to *Managing Pins and Groups in the Pin Planner* in Quartus II Help.

Verifying Pin Migration Compatibility

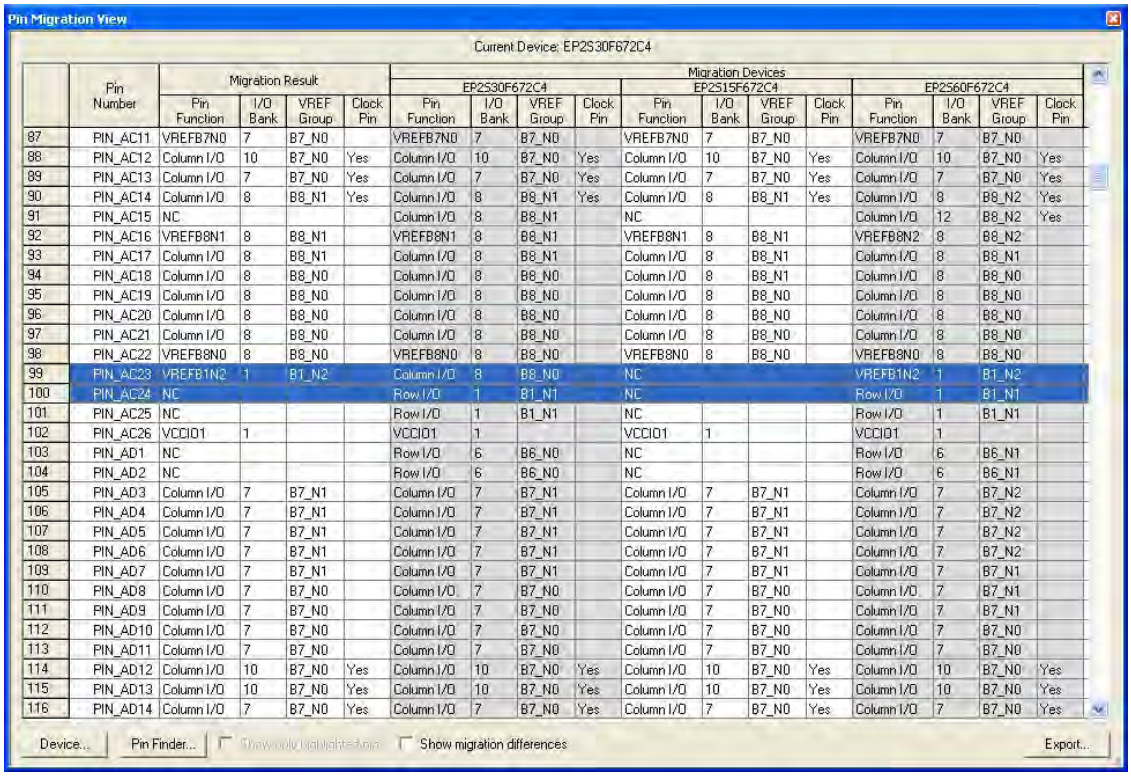
You can use the Pin Migration View window in Pin Planner to assist you in verifying whether your pin assignments migrate to a different device successfully. You can vertically migrate to a device with a different density while using the same device package, or migrate between packages with different densities and ball counts.

For more information, refer to *Viewing Pin Migration Compatibility* in Quartus II Help.

When you select migration devices early in the design process, the Pin Planner displays only the pins that are available in the current device and in all migration devices. If you select migration devices later in your design cycle, there may be assignments for I/O nodes in your original design that do not have corresponding pins in a migration device. If no corresponding pin exists, the Compiler cannot honor the assignment and an error occurs when you try to recompile the design.

The Pin Migration View window helps you identify the difference in pins that can exist between migration devices. For example, Figure 5–10 shows the highlighted pin AC24 existed in the target EP2S30 device, but does not exist in one of the migration devices, resulting in a No Connect (NC).

Figure 5–10. Pin Migration View



The migration result for the pin function of highlighted PIN_AC23 is not an NC but a voltage reference VREFB1N2 even though the pin is an NC in one of the migration devices. VREF standards have a higher priority than an NC, thus the migration result display the voltage reference. Even if you do not use that pin for a port connection in your design, you must use the VREF standard for I/O standards that require it on the actual board for the migration device.

If one of the migration devices has pins intended for connection to V_{CC} or GND and these same pins are I/O pins on a different device in the migration path, the Quartus II software ensures these pins are not used for I/O. Ensure that these pins are connected to the correct PCB plane.

When migrating between two devices in the same package, pins that are not connected to the smaller die may be intended to connect to V_{CC} or GND on the larger die. To facilitate migration, you can connect these pins to V_{CC} or GND in your original design because the pins are not physically connected to the smaller die.

For more information about migration, refer to *AN90: SameFrame Pin-Out Design for FineLine BGA Packages*. For more information about designing for HardCopy series devices, refer to the *Quartus II Support for HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook*.

Viewing Simultaneous Switching Noise (SSN) Results

You can perform and SSN analysis of your design to estimate the voltage noise for each pin in the design. You can view the SSN results in the Pin Planner and adjust your I/O assignments to avoid potential signal integrity issues.

For more information about running the SSN Analyzer and viewing the results in the Pin Planner, refer to *Running the SSN Analyzer* in Quartus II Help.

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 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

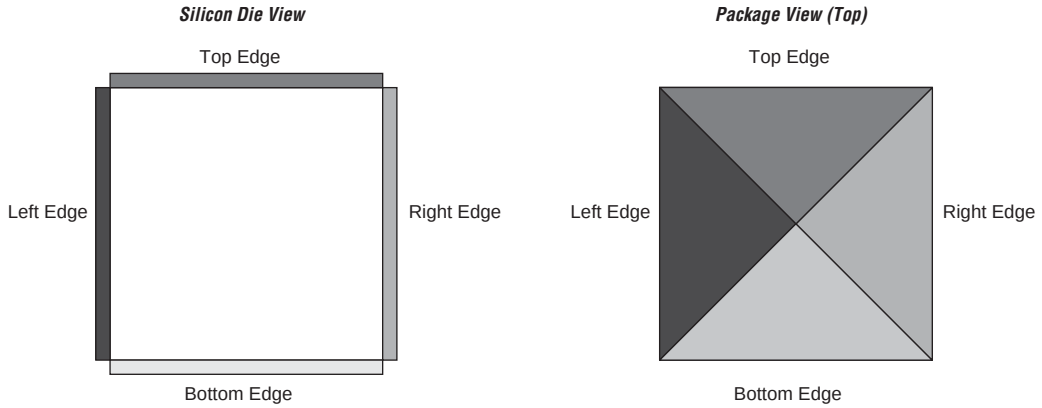
For more information about device support for I/O bank, VREF group, and edge location assignments, refer to *I/O bank*, *VREF group*, and *edge* in Quartus II Help.

You can assign your pins to a location with the Pin Planner. 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 Class I**, can be placed in the same I/O bank.

If your design contains a large bus that exceeds the pins available in a particular I/O bank, you can use edge location assignments to place the bus. Edge location assignments improve the circuit board routing ability of large buses, because they are close together near an edge. Figure 5–11 shows Altera device package edges.

For more information about creating pin assignments with the Pin Planner, refer to *Assigning Pins* in Quartus II Help.

Figure 5–11. Die View and Package View of the Four Edges on an Altera Device



Creating Exclusive I/O Group Assignments

You can create exclusive groups comprised of pins by creating and modifying the **Exclusive I/O Group** logic option with the Pin Planner. When you create exclusive I/O groups in your 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 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.

For more information about creating logic option assignments with the Pin Planner, refer to *Assigning Pins* in Quartus II Help.

Changing the Slew Rate and Drive Strength

As part of I/O planning you can set both the slew rate and drive strength of pins. Both slew rate and drive strength affect the outgoing signal integrity of the pin. Depending on your target device family, you can adjust the slew rate of a pin with either the **Slew Rate** or **Slow Slew Rate** logic option; you can adjust the drive strength of a pin with the **Current Strength** logic option. The settings you create for slew rate and drive strength are honored during live I/O check, I/O assignment analysis, and full compilation.

For more information about slew rate support, refer to the appropriate device handbook available on the Literature and Technical Documentation page of the Altera website.

For more information about creating logic option assignments with the Pin Planner, refer to *Assigning Pins* in Quartus II Help.

If you want to make changes to slew rate or drive strength after you compile your design, you can use the Resource Property Editor to perform engineering change orders (ECOs). ECOs allow you to compile the changes to your design, without changing the synthesis or fitting results.

For more information about making post-compilation changes, refer to *About Post-Compilation Changes* in Quartus II Help.

For more information about ECOs, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*. For more information about the effect of I/O settings on signal integrity on the board, refer to *AN 476: Impact of I/O Settings on Signal Integrity in Stratix III Devices*.

Assigning Locations for Differential Pins

When you use the Pin Planner to assign a differential I/O standard to a single-ended top-level pin in your design, it automatically recognizes the negative pin as part of the differential pin pair assignment and creates the negative pin for you. The Quartus II software writes the location assignment for the negative pin to the **.qsf** file; however, the I/O standard assignment is not added to the **.qsf** file for the negative pin of the differential pair.

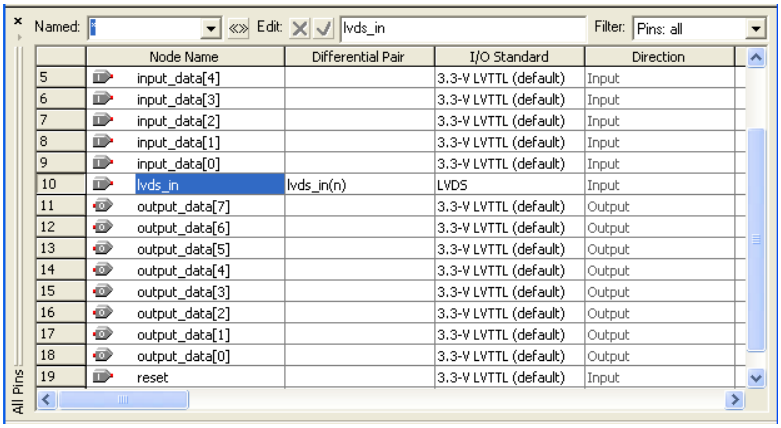
For example, Figure 5–12 shows a design in which you have a top-level pin defined as `lvds_in` to which you assign a differential I/O standard. The Pin Planner automatically creates the differential pin, `lvds_in(n)`, to complete the differential pin pair.

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 target device. Single-ended pins that feed a PLL and are assigned to the negative clock pin device cause the design to not fit.

For more information about identifying and assigning differential pins with the Pin Planner, refer to *Assigning Pins* in Quartus II Help.

For more information about assigning locations for differential pins in HDL code with low-level I/O primitives, refer to"Creating Pin Assignments with Low-Level I/O Primitives" on page 5–22.

Figure 5–12. Creating a Differential Pin Pair in the Pin Planner



Creating Pin Assignments with the Chip Planner

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 and determine the distances between user I/O pads and V_{CC}, GND, and V_{REF} pads to avoid signal integrity issues.

For more information about creating pin location assignments in the Chip Planner, refer to *Working with Assignments in the Chip Planner* in Quartus II Help.

For more information about pin placement guidelines, refer to the appropriate device handbook available on the Literature and Technical Documentation page of the Altera website.

Creating Pin Assignments with Tcl Scripts

You can use Tcl scripts to create pin-related assignments as part of a script-based compilation flow. You can enter individual Tcl commands in the Tcl Console window of the Quartus II software. To run a Tcl script with the Quartus II software, type the following command at a system prompt:

```
quartus_sh -t my_tcl_script.tcl ↵
```

Example 5–3 shows the `set_location_assignment` and `set_instance_assignment` Tcl commands used to create 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]
```

For more information about creating and running Tcl scripts with the Quartus II software, refer to *Creating and Running Tcl Scripts* in Quartus II Help.

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 and *API Functions for Tcl* in Quartus II Help.

Creating Pin Assignments in HDL Code

You can use synthesis attributes or low-level I/O primitives to embed pin-related assignments directly in your HDL code. When you analyze and synthesize your HDL code, the information in the HDL code is converted into the appropriate assignments. There are two ways to specify pin-related assignments with HDL code:

- 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 for setting parameters or attributes

Synthesis Attributes

Synthesis attributes allow you to embed pin-related assignments in your HDL code. During Analysis and Synthesis, the Quartus II software reads these synthesis attributes and translates them into assignments. The assignments are then populated in the Pin Planner. If you modify or delete these pin assignments in the Pin Planner and then recompile your design, any changes made in the Pin Planner take precedence over the assignments you made with synthesis attributes in your HDL code. Quartus II integrated synthesis supports the `chip_pin`, `useioff`, and `altera_attribute` synthesis attributes.

For more information about integrated synthesis, synthesis attributes, and syntax, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For more information about synthesis attributes supported by third-party synthesis tools, contact your vendor.

chip_pin and useioff

Use the `chip_pin` and `useioff` synthesis attributes to create pin location assignments and to create **Fast Input Register**, **Fast Output Register,** and **Fast Output Enable Register** logic option assignments, respectively, in your HDL code. For all other assignments, including pin-related assignments, use the `altera_attribute` synthesis attribute as discussed in the "altera_attribute" section.

For more information, refer to *useioff VHDL Synthesis Attribute*, *useioff Verilog HDL Synthesis Attribute*, *chip_pin VHDL Synthesis Attribute*, and *chip_pin Verilog HDL Synthesis Attribute* in Quartus II Help.

Example 5–4 and Example 5–5 use the `chip_pin` and `useioff` attributes to embed location and **Fast Input Register** logic option assignments in both a Verilog HDL and VHDL design file using the 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

Use the `altera_attribute` synthesis attribute to create other pin-related assignments in your HDL code. The `altera_attribute` attribute is understood only by Quartus II integrated synthesis and supports all types of instance assignments.

For more information, refer to *altera_attribute VHDL Synthesis Attribute*, and *altera_attribute Verilog HDL Synthesis Attribute* in Quartus II Help.

Example 5–6 and Example 5–7 use the `altera_attribute` attribute to embed **Fast Input Register** logic option assignments and I/O standard assignments in 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;
```

Creating Pin Assignments with Low-Level I/O Primitives

You can create pin-related assignments for your top-level nodes using low-level I/O primitives, which allow you to create pin location assignments and set I/O standards, drive strengths, slew rates, and on-chip termination (OCT) value assignments. You can also use low-level differential I/O primitives to define both positive and negative pins of a differential pair in the HDL code for your design.

The pin-related assignments made with primitives do not automatically appear in the Pin Planner. When you use low-level I/O primitives to define various pin-related I/O assignments, the assignments are honored only after you perform a full compilation. After performing a full compilation, you can populate these assignments in the Pin Planner by back-annotating pin assignments.

For more information about back-annotating assignments, refer to *Back-Annotating Assignments for A Project* in Quartus II Help. For more information about differential I/O primitives, refer to *Primitives* in Quartus II Help.

For more information about using low-level I/O primitives in your design, refer to the *Designing with Low-Level Primitives User Guide*.

Validating Pin Assignments

The Quartus II software includes predefined I/O rules to guide you in pin placement and checks your pin-related assignments against these rules during pin planning. You must validate all pin-related assignments in your design. During a full compilation, the Quartus II software does not report illegal pin assignments until the Fitter stage. To preliminarily validate pin-related assignments against the predefined I/O rules, you can use the live I/O check feature or run I/O assignment analysis after performing analysis and synthesis. Typically, the I/O assignment analysis completes quickly. By preliminarily validating your pin-related assignments before fully compiling your design, you can avoid recompiling your design to fix pin-related assignment errors, thus reducing your overall design time. To fully validate pin-related assignments against all I/O timing checks, you must perform a full compilation.

Table 5–3 and Table 5–4 list a subset of the I/O rule checks performed when you run I/O assignment analysis.

For more information about each I/O rule, including which devices support which rules, refer to the device handbooks available on the Literature and Technical Documentation page of the Altera website.

Table 5–3. Examples of I/O Rule Checks (Part 1 of 2)

Rule	Description	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.	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.	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.	No
I/O standard and location conflicts	Checks whether the pin location supports the assigned I/O standard.	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.	No

Table 5–3. Examples of I/O Rule Checks (Part 2 of 2)

Rule	Description	HDL Required?
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.	No
I/O standard and drive strength conflicts	Checks whether the drive strength assignments are within the specifications of the I/O standard.	No
Drive strength and location conflicts	Checks whether the pin location supports the assigned drive strength.	No
BUSHOLD and location conflicts	Checks whether the pin location supports BUSHOLD. For example, dedicated clock pins do not support BUSHOLD.	No
WEAK_PULLUP and location conflicts	Checks whether the pin location supports WEAK_PULLUP (for example, dedicated clock pins do not support WEAK_PULLUP)	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.	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.	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.	Yes
PLL and I/O pin location compatibility check	Checks whether pins connected to a PLL are assigned to the dedicated PLL pin locations.	Yes

Table 5–4. SSN-Related Rules

Rule	Description	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.	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.	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.	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.	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.	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.	No
Too many outputs in a VREFGROUP	Checks whether too many outputs are in a VREFGROUP.	No

Validating Pin Assignments with the Live I/O Check Feature

The live I/O check feature provides live I/O rule checking capability to prevent you from creating pin placements that violate I/O fitting rules. When the live I/O check feature is turned on, 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 so you can immediately correct basic errors 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} and V_{REF} voltage compatibility rules
- Electromigration (current density) rules
- Simultaneous Switching Output (SSO) rules
- I/O property compatibility rules, such as drive strength compatibility, I/O standard compatibility, `PCI_IO` clamp diode compatibility, and I/O direction compatibility

When the live I/O check feature is turned on, the Quartus II software prevents you from assigning pins to unavailable locations. The following are examples of unassignable locations:

- An I/O bank or VREF group with no available pins
- The negative pin of a differential pair if the positive pin of the differential pair is assigned with a node name with a differential I/O standard
- Pin locations that do not support the I/O standard assigned to the selected node name
- For HSTL- and SSTL-type I/O standards, VREF groups of a different V_{REF} voltage than the selected node name

You can turn on or turn off the live I/O check feature at any time. By default, the live I/O check feature is turned off. When the live I/O check feature is turned on, the Quartus II software immediately checks whether your new pin-related assignments pass the basic I/O buffer rules. The Live I/O Check Status window displays the total numbers of errors and warnings while you create and edit pin-related assignments. The Messages window shows detailed messages about any errors or warnings.

Although 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.

For more information about using the live I/O check feature to validate pin assignments, refer to *Assigning Pins* in Quartus II Help.

Validating Pin Assignments with I/O Assignment Analysis

Performing I/O assignment analysis allows you to validate your I/O assignments and surrounding logic for illegal assignments and violations of board layout rules early in the design process. The I/O assignment analysis feature also checks blocks that directly feed or are fed by resources such as a PLLs, LVDS, or gigabit transceiver blocks. You can check the legality of pin assignments before you compile your design, or allow the process to run automatically during compilation. If design files are available, you can perform more thorough legality checks on the I/O pins and surrounding logic in your design. Legality checks include proper VREF pin use, valid pin location assignments, and acceptable mixed I/O standards. Altera recommends that you run I/O assignment analysis each time you add or modify a pin-related assignment.

If you have partial or complete design files, you must perform Analysis and Synthesis to generate a synthesized (mapped) netlist before you can perform I/O assignment analysis.

Performing I/O assignment analysis directs the Fitter to read assignments from your mapped netlist and the **.qsf** to determine the legality of your pin-related assignments. These pin-related assignments include pin settings such as I/O standards, drive strength, and location assignments.

For more information about performing I/O assignment analysis, refer to *Assigning Pins* in Quartus II Help.

Incomplete I/O assignments trigger warnings during I/O assignment analysis. You can view the I/O Assignment Warnings report to find and resolve warnings generated during I/O assignment analysis. For example, you may receive a warning that some of the pins in the design are missing a drive strength or slew rate. Single-ended output and bidirectional pins default to the non-calibrated OCT setting if you do not assign drive strength and slew rate options to the pins, or if other OCT options are assigned to the pins. To resolve this issue, you can either assign drive strength or slew rate settings to the pins with the **Current Strength** or **Slew Rate** or **Slow Slew Rate** logic options, or assign the **Termination** logic option to the pins with a series setting. You cannot use drive strength and slew rate settings when a pin is assigned an OCT setting.

For more information about creating logic option assignments with the Pin Planner, refer to *Assigning Pins* in Quartus II Help.

During I/O assignment analysis the Fitter automatically assigns suggested pin locations to unassigned pins in your design, based on your design constraints, so it can perform pin legality checks. For example, if you assign an edge location to a group of LVDS pins, the Fitter assigns pin locations for each LVDS pin in the specified edge location and then performs legality checks. To display the Fitter-placed pins use the Show Fitter Placements feature in the Pin Planner. To accept these suggested pin locations, you must back-annotate your pin assignments.

For more information about the Show Fitter Placements feature, refer to the *Show Commands* in Quartus II Help. For more information about back-annotating assignments, refer to *Back-Annotating Assignments for A Project* in Quartus II Help.

The following design flows show two different circumstances in which you can use I/O assignment analysis:

- Use the flow shown in Figure 5–13 if the you must complete board layout before starting the design. This flow does not require design files and checks the legality of your pin assignments.
- Use the flow shown in Figure 5–14 if your design is complete. 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.

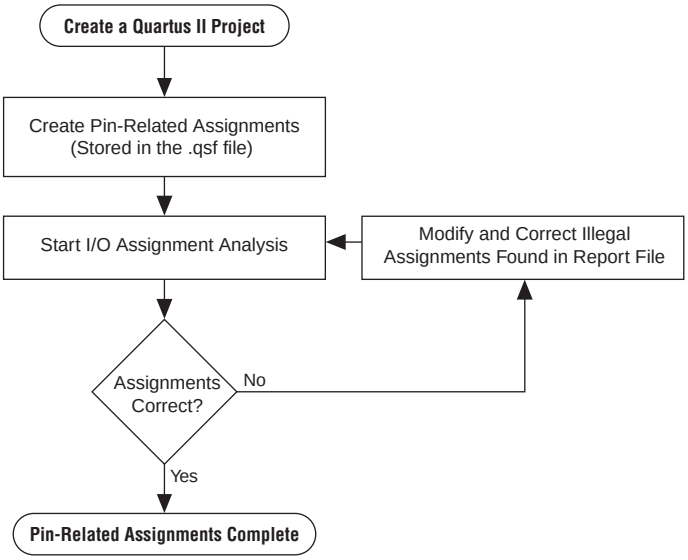
Running I/O Assignment Analysis without Design Files

During the early stages of developing a device, board layout engineers may request preliminary or final pin-outs. It is time consuming to manually check whether the pin-outs violate any design rules. To quickly perform basic checks on the legality of your pin assignments, you can use the Quartus II software to perform I/O assignment analysis.

If you create pin-related assignments in Mentor Graphics I/O Designer software, you can import an **.fx** into the Quartus II software.

Running I/O assignment analysis performs limited checks on pin assignments made in a design in which you specified a device, but does 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 create pin-related assignments for each pin with the Pin Planner. After you assign an I/O standard to each reserved pin, run I/O assignment analysis to ensure that there are no I/O standard conflicts in each I/O bank. Figure 5–13 shows the work flow for assigning and analyzing pin-outs without design files.

Figure 5–13. Assigning and Analyzing Pin-Outs without Design Files



When you make and analyze pin-related assignments without design files, make sure you reserve the pins you intend to use as I/O pins, so the Fitter can determine each pin type. After performing I/O assignment analysis, correct any errors reported by the Fitter and rerun I/O assignment analysis until all errors are corrected.

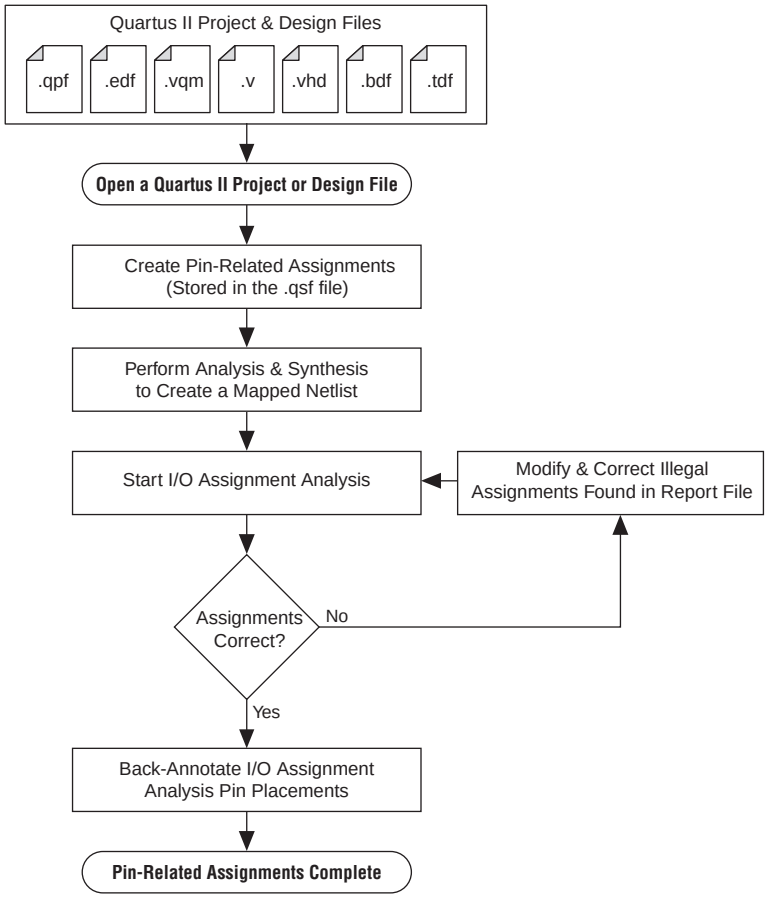
For more information about reserving pins with the Pin Planner, refer to *Assigning Pins* in Quartus II Help.

Without a complete design, running I/O assignment analysis performs limited checks and cannot guarantee that your assignments do not violate design rules.

Running I/O Assignment Analysis with Design Files

If you have preliminary or complete design files, you can run I/O assignment analysis to help you determine if your design will fit. The rules checked during I/O assignment analysis depend on the completeness of the design. If you have a complete design, the legality of all pin-related assignments are thoroughly checked during I/O assignment analysis. If you have a partial design, which can be just the top-level wrapper file, the legality of those pin-related assignments for which there is enough information are checked during I/O assignment analysis. Figure 5–14 shows the work flow for assigning and analyzing pin-outs without design files.

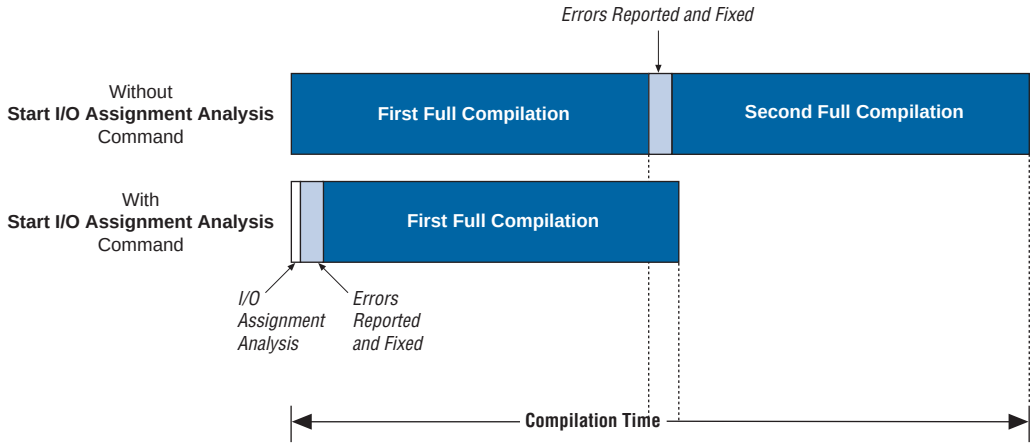
Figure 5–14. Assigning and Analyzing Pin-Outs with Design Files



If you run I/O assignment analysis on incomplete design files, you may still encounter errors during full compilation. For example, you may 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 you have not yet instantiated in the design. The checks run during I/O assignment analysis do not account for the logic that the pin drives, and do not 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, include these MegaWizard Plug-In Manager-generated files. To assign and analyze pin-related assignments successfully, after performing I/O assignment analysis, correct any errors reported by the Fitter and rerun I/O assignment analysis until all errors are corrected.

Figure 5–15 shows the compilation time benefit of performing I/O assignment analysis before running a full compilation.

Figure 5–15. Saving Compilation Time with the I/O Assignment Analysis



Optimizing I/O Assignment Analysis with Output Enable Group Logic Option Assignments

Each device has a certain number of VREF pins, and each VREF pin supports a certain number of I/O pins. A VREF pin and its supported I/O pins are called a VREF bank. The VREF pins are used only for inputs with VREF I/O standards, such as HSTL- and SSTL-type I/O standards. 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 example, for devices in 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 more information about device VREF pins and their associated I/O pins, refer to the *Pin-Out Files for Altera Devices* page of the Altera website.

For interfaces that use bidirectional VREF I/O pins, your design must meet the output restriction for each I/O bank when the pins are driving in either direction. If a set of bidirectional signals are controlled by different output enables, they are treated as independent output enables during I/O assignment analysis, thus creating a situation in which the Fitter may determine that your design violates VREF restrictions. To treat the set of bidirectional signals as a single output enable group so that the Fitter does not determine that the design violates the requirements for the maximum number of pins driving out of a VREF group, assign the **Output Enable Group** logic option assignment to the bidirectional signals. Assign an integer value for the **Output Enable Group** logic option assignment and assign the same integer value to all sets of signals that are driving in the same direction. Assigning this logic option to groups of signals is important in the case of external memory interfaces.

For example, in the case of a DDR2 interface in a Stratix II device, the device can have 30 pins in a VREF group. Each byte lane for a ×8 DDR2 interface has one DQS pin and eight DQ pins, for a total of nine pins per byte lane. The DDR2 interface uses **SSTL-18 Class I** 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 four byte lanes. Using 30 I/O pins in a VREF group, there are three byte lanes and an extra byte

lane that supports the three remaining pins. If you do not use the **Output Enable Group** logic option assignment, the Fitter analyzes each byte lane as an independent group driven by a unique output enable during I/O assignment analysis. With this arrangement, the worst-case scenario is when the three 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 Fitter reports an error that is not applicable to your design. Assigning the **Output Enable Group** logic option assignment to the DQS and DQ pins forces the Fitter to check these pins as a group driven by a common output enable during I/O assignment analysis. When you use the **Output Enable Group** logic option assignment, the DQS and DQ pins are checked as all input pins or all output pins and are not in violation of the I/O rules.

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). To avoid errors during I/O assignment analysis, use the **Output Enable Group** logic option assignment to assign the data mask to the same value as the DQ and DQS signals.

You can also assign the **Output Enable Group** logic option to 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, thus removing the VREF input pins from the VREF analysis. For example, the QVLD signal for an RLDRAM II interface is active only during a read. During a write, the QVLD pin is not active and does not count as an active VREF input pin in the VREF group. You can place the QVLD pins in the same output enable group as the RLDRAM II data pins.

Understanding the I/O Assignment Analysis Report

When I/O assignment analysis is complete, you can view detailed analysis reports and a **.pin**. 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.

The Fitter section of the Compilation report contains information generated during I/O assignment analysis, including the following reports:

- I/O Assignment Warnings
- Resource Section
- I/O Rules Section

The I/O Assignment Warnings report provides a list of pins and the Fitter warnings generated for the pins during I/O assignment analysis

The Resource Section contains reports that categorize the pins as input pins, output pins, and bidirectional pins. You can view the utilization of each I/O bank in your device in the I/O Bank Usage report.

The I/O Rules Section includes detailed information about the I/O rules tested during I/O assignment analysis and contains the following reports:

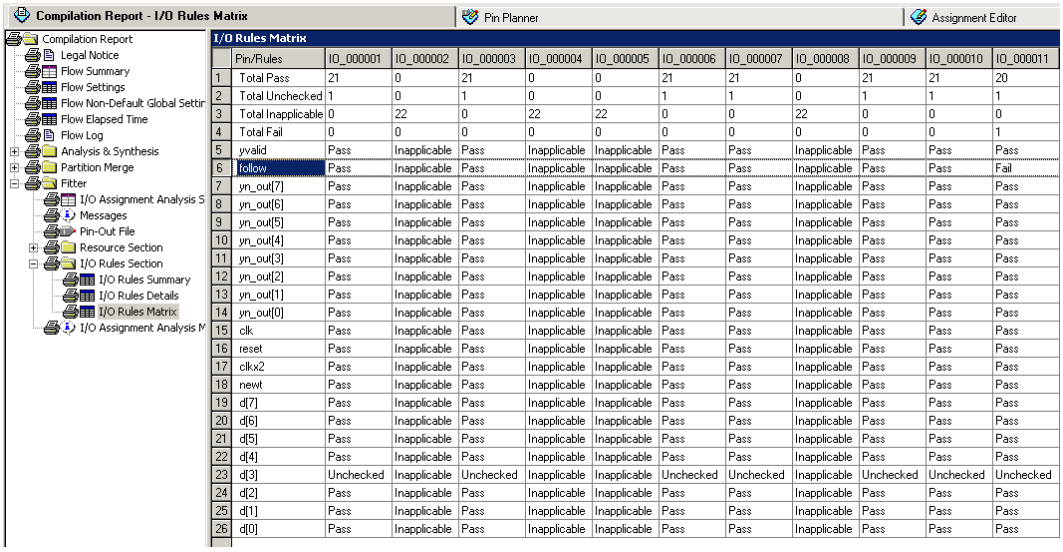
- I/O Rules Summary report
- I/O Rules Details report
- I/O Rules Matrix report

The I/O Rules Summary report provides a quick summary of the number of I/O rules tested and how many applicable rules passed, failed, or were not checked due to other failing rules.

The I/O Rules Details report provides detailed information on all I/O rules. The **Status** column indicates whether applicable rules passed, failed, or could not be checked. All rules are given a level of severity rating to indicate their level of importance for an effective analysis.

The I/O Rules Matrix report (refer to Figure 5–16) provides a list of the I/O rules checked by the Fitter for each pin in the design. Rules that apply to the target device family either pass or fail for each pin. Rules marked **Inapplicable** are rules that do not apply to the target device family. You can ignore any rule marked **Inapplicable**.

Figure 5–16. I/O Rules Matrix



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

Validating Pin Assignments with Full Compilation

After performing preliminary pin assignment validation with the live I/O check feature and running I/O assignment analysis, you must perform a final I/O timing check of you design by performing a full compilation. To avoid costly board respins, you must include complete design files and constraints. With timing information, the Fitter makes intelligent placement and routing to achieve optimal timing performance in your design. Use the TimeQuest Timing Analyzer to create timing constraints for input, output, and bidirectional pins.

For more information about the TimeQuest analyzer, refer to the *The Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Performing I/O Timing Analysis

Timing analysis is usually run during a full compilation of your design or when you perform an early timing estimate. You can also run timing analysis independently after you fully compile the design. For example, if you change the slew rates or drive strengths of some I/O pins with 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, especially with high-speed designs, take board-level signal integrity and timing into account. When adding a 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, is vital for proper system operation

As part of I/O planning, you must understand the I/O timing results reported by the Quartus II software after performing timing analysis on your design. If all your design files are complete and you have fully compiled your design, all the timing checks related to I/O timing are performed during timing analysis. 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 in I/O timing results is how accurately you specify the output loads of the output and bidirectional pins in your 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:

- Advanced I/O timing using a user-defined board trace model to produce enhanced timing reports from accurate, "board-aware", simulation models

  Advanced I/O timing allows you to configure a complete board trace model for each I/O standard or pin used in your design. With advanced I/O timing, the TimeQuest 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.

  For more information about advanced I/O timing, including device support, refer to *About Advanced I/O Timing* in Quartus II Help.

- I/O timing using a default or user-specified capacitive load without signal integrity analysis

  The TimeQuest analyzer creates timing reports that measure t_{CO} to an I/O pin using a default or user-specified value for a capacitive load.

- Full board routing simulation in third-party tools using Altera-provided or Quartus II software-generated IBIS or HSPICE I/O models

  The IBIS and HSPICE Writers the simulation model files for use by third-party board simulation tools. 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.

For devices that support advanced I/O timing, it is the default method of I/O timing analysis. For all other devices, you must use a default or user-specified capacitive load assignment to determine t_{CO} and power measurements.

For more information about advanced I/O timing support, refer to the appropriate device handbook available on the Literature and Technical Documentation page of the Altera website. 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 page of the Altera website.

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*.

Enabling and Configuring Advanced I/O Timing

With the advanced I/O timing feature, you can expand upon the basic timing and power measurements made with the capacitive loading settings. The advanced I/O timing feature 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.

When you use the advanced I/O timing feature, the board trace model replaces any capacitive load setting you made 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, including the **Near capacitance**, **Far capacitance**, and **Transmission line distributed capacitance** elements of the model.

Defining Overall Board Trace Models

You can define an overall board trace model for each I/O standard in your design that is the default model for all pins that use a particular I/O standard. After configuring the overall board trace model, you can customize the model for specific pins using the Board Trace Model window in the Pin Planner.

Custom component value changes you make to selected pins in the Pin Planner take priority and are not affected by subsequent changes to a specific components for the entire design. Similarly, any changes you make to specific pins do not affect the component settings for the entire design.

When you define an overall board trace model you can specify the board trace, termination, and capacitive load parameters for each I/O standard. 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 without advanced I/O timing. For differential I/O standards, the component values you set are used for both the positive and negative signals of a differential pin pair.

All the assignments for board trace models you specify are saved to the **.qsf**. You can also use Tcl commands to create board trace model assignments. Example 5–8 shows Tcl commands for specifying board trace model assignments.

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 se1_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 se1_p
```

For more information about defining a board trace model for your entire design, refer to *Using Advanced I/O Timing* in Quartus II Help. For more information about configuring specific pins in your board trace model, refer to *Creating Board Trace Model Assignments in the Pin Planner* in Quartus II Help. For more information about configuring component values for a board trace model, including a complete list of the supported unit prefixes and setting the values with Tcl scripts, refer to *Board Trace Model* Quartus II Help.

For more information about the default models used for measuring I/O delay, refer to the appropriate device handbook available on the Literature and Technical Documentation page of the Altera website.

Customizing the Board Trace Model in the Pin Planner

In the Pin Planner, you can view a graphical representation of the board trace model you configured with the Board Trace Model window. Initially, the settings you create for the overall board trace model match the settings in the Pin Planner. For differential signals, the Board Trace Model window displays the routing and components for both the positive and negative signals of the differential pair. Any changes you make 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.

When editing board trace model assignments, 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, select **short** or **open**, respectively, for the component value.

Configuring Board Trace Models

The Quartus II software provides board trace model templates for various I/O standards in which you can fill in various parameters. Figure 5–17 shows the template for a **2.5-V** I/O standard. This model consists of near-end and far-end board component parameters.

Modeling of the near-end of the board trace includes the elements which are close to the device and modeling of the far-end includes the elements which are at the receiver end of the link, closer to the receiving device. The topology represented in the Quartus II board trace model is conceptual and does not necessarily match the board trace component for component. For example near-end model parameters can represent device-end discrete termination and breakout traces. Far-end modeling can represent the bulk of the board trace to discrete external memory components, and the far end termination network. The same circuit can be analyzed with near-end modeling of the entire board, including memory component termination, and far-end modeling of the actual memory component.

Figure 5–17. Board Trace Model

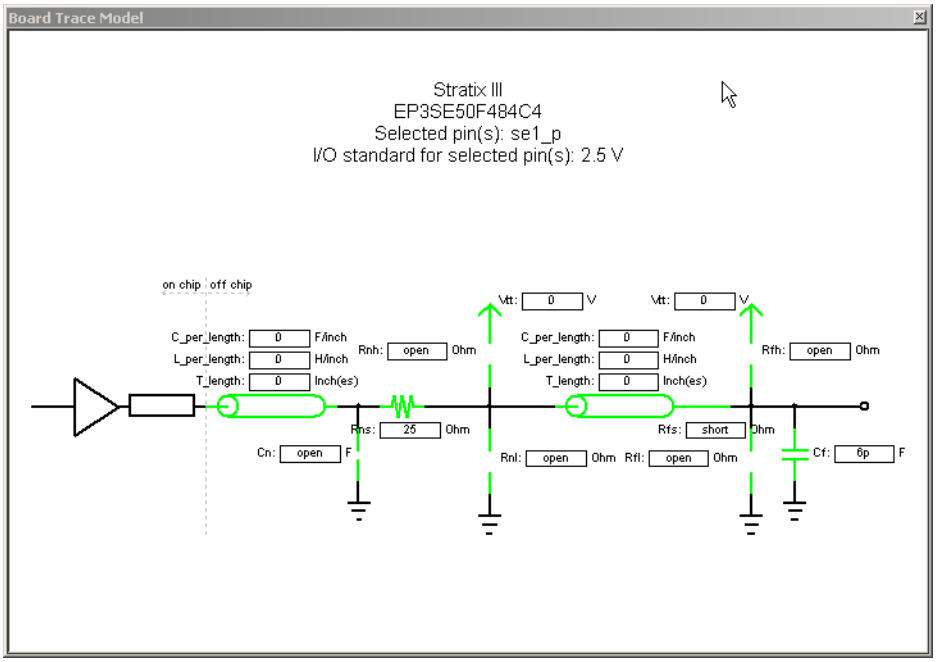
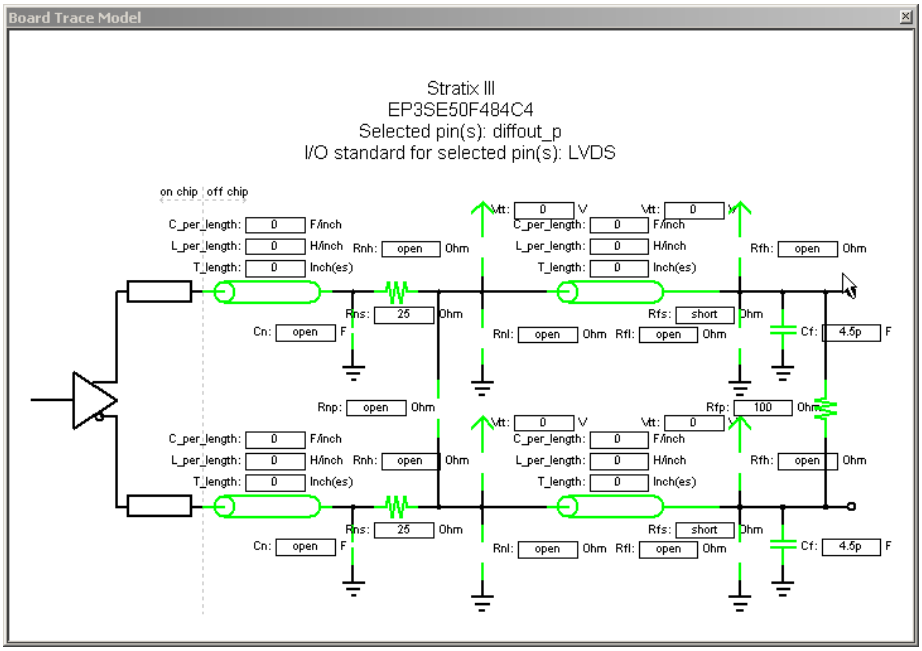


Figure 5–18 shows the template for the **LVDS** I/O standard. The far-end capacitance (Cf) represents the external-device or multiple-device capacitive load. If you have multiple devices on the far-end, you must find the equivalent capacitance at the far-end, taking into account all receiver capacitances. The far-end capacitance can be the sum of all the receiver capacitances.

For more information about the specifications for external device capacitance values, refer to the appropriate device handbook available on the Literature and Technical Documentation page of the Altera website.

The Quartus II software models lossless transmission lines, and does not require a transmission-line resistance value. Only distributed inductance (L) and capacitance (C) values are needed. The distributed L and C values of transmission lines must be entered on a per-inch basis, and can be obtained from the PCB vendor or manufacturer, the CAD Design tool, or a signal integrity tool such as the Mentor Graphics Hyperlynx software.

Figure 5–18. Differential Board Trace Model



Specifying Near-End vs Far-End Timing Analysis

With advanced I/O timing analysis, you have the option of selecting a near-end or far-end point for your I/O timing. With near-end timing, the timing is analyzed to the device pin. Figure 5–17 shows near-end timing ending at the vertical dashed line separating the device I/O pin and off-chip components.

By default, advanced I/O timing analysis analyzes output I/O timing to the device pin. When you choose a near-end endpoint, you can use the `set_output_delay` SDC timing constraint to account for the delay across the board. However, when you choose a far-end I/O timing endpoint, then advanced I/O timing analysis analyzes timing to the external device input, at the far end of the board trace. Whether you choose a near-end or far-end timing endpoint, the board trace models are taken into account during timing analysis.

For more information about calculating I/O timing to the near-end or far-end of the board trace, refer to *Using Advanced I/O Timing* in Quartus II Help.

Understanding Advanced I/O Timing Analysis Reports

When you perform advanced I/O analysis, the TimeQuest analyzer creates reports that signal integrity reports that provide board delay estimates and signal integrity data.

The TimeQuest analyzer section of the Compilation report contains information generated during advanced I/O timing analysis, including the following reports:

- Board Trace Model Assignments
- Signal Integrity Metrics

The Board Trace Model Assignments report summarizes the board trace model component settings for each output and bidirectional signal.

The Signal Integrity Metrics report contains all the signal integrity metrics calculated during advanced I/O timing analysis based on the board trace model settings for each output or bidirectional pin. The reports contain many metrics, including measurements at both the FPGA pin and at the far-end load of board delay, steady state voltages, and rise and fall times.

By default, the TimeQuest analyzer generates the Slow-Corner Signal Integrity Metrics report. To generate a Fast-Corner Signal Integrity Metrics report you must change the delay model.

For more information about the reports generated during advanced I/O timing analysis, refer to *Advanced I/O Timing Reports* in Quartus II Help. For more information about the metrics calculated during advanced I/O timing analysis, including diagrams illustrating the metrics on output waveforms, refer to *Signal Integrity Metrics* in Quartus II Help. For more information about changing the delay model, refer to the *Create Timing Netlist Dialog Box* in Quartus II Help.

For information about the configuration and use of the TimeQuest analyzer, refer to *The Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Adjusting I/O Timing and Power with Capacitive Loading

When calculating t_{CO} and power for output and bidirectional pins, the TimeQuest analyzer and the PowerPlay Power Analyzer use a bulk capacitive load. You can adjust the value of the capacitive load per I/O standard to obtain t_{CO} and power measurements that more accurately reflect the behavior of the output or bidirectional net on your PCB. Input pins ignore any capacitive load settings. You can adjust the capacitive load settings per I/O standard, in picofarads (pF), for your entire design. During compilation, the Compiler measures power and t_{CO} measurements based on your settings. You can also adjust the capacitive load on an individual pin with the **Output Pin Load** logic option.

For more information about adjusting the value of the capacitive load for your entire design, refer to *Setting Up and Running a Compilation* in Quartus II Help. For more information about adjusting the value of the capacitive load on an individual pin, refer to *Assigning Pins* in Quartus II Help.

Incorporating PCB Design Tools

Signal and pin assignments are initially made by the chip designer and it is up to the board designer to correctly transfer these assignments to the symbols in their system circuit schematics and board layout. As the board design progresses, pin reassignments may be requested or required to optimize the layout. These reassignments must in turn be communicated to the chip designer, so that the new assignments can be validated during I/O assignment analysis and processed through an updated placement-and-routing of the device.

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*.

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 I/O Assignment Analysis

You can run I/O assignment analysis with a Tcl command or with a command-line command.

Enter the following in the Tcl console or a Tcl script:

```
execute_flow -check_ios ↵
```

Type the following at a system command prompt:

```
quartus_fit <project name> --check_ios ↵
```

For more information about running I/O assignment analysis, refer to "Understanding the I/O Assignment Analysis Report" on page 5–31.

Generating a Mapped Netlist

You can generate a mapped netlist with a Tcl command or with a command-line command.

Enter the following in the Tcl console or in a Tcl 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

You can reserve pins with a Tcl command.

Use the following Tcl command to reserve a pin:

```
set_instance_assignment -name RESERVE_PIN <value> -to <signal name>
```

Use one of the following valid reserved pin values:

- "AS BIDIRECTIONAL"
- "AS INPUT TRI-STATED"
- "AS OUTPUT DRIVING AN UNSPECIFIED SIGNAL"
- "AS OUTPUT DRIVING GROUND"
- "AS SIGNALPROBE OUTPUT"

Ensure you include the quotation marks when specifying the reserved pin value.

Creating Location Assignments

You can create location assignments with a Tcl command.

Use the following Tcl command to assign a signal to a pin or device location:

```
set_location_assignment <location> -to <signal name> ↵
```

Valid locations are pin locations, I/O bank locations, or edge locations. Pin locations include pin names, such as PIN_A3. I/O bank locations include IOBANK_1 up to IOBANK_*n*, in which *n* is the number of I/O banks in the device.

Use one of the following valid edge location values:

- EDGE_BOTTOM
- EDGE_LEFT
- EDGE_TOP
- EDGE_RIGHT

For more information about location assignments, refer to "Creating Location Assignments" on page 5–17.

For more information about I/O banks in your device, refer to the appropriate device handbook available on the Literature and Technical Documentation page of the Altera website.

Creating Exclusive I/O Group Assignments

You can create exclusive I/O group assignments with a Tcl command.

Use the following Tcl command to create an exclusive I/O group assignments:

```
set_instance_assignment -name "EXCLUSIVE_IO_GROUP" -to pin
```

For more information about exclusive I/O group assignments, refer to "Creating Exclusive I/O Group Assignments" on page 5–18.

Changing the Slew Rate and Drive Strength

You can create slew rate and drive strength assignments with a Tcl command.

Use the following Tcl commands to create an slew rate and drive strength assignments:

```
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 slew rate and drive strength assignments, refer to "Creating Pin Assignments with the Chip Planner" on page 5–20.

Conclusion

The Quartus II software provides many tools and features to help you with I/O planning, including the ability to validate pin assignments in all design stages, even before the development of your design. The ability to import and export assignments between the Quartus II software and other PCB tools allows 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.

Document Revision History

Table 5–5 shows the revision history for this chapter.

Table 5–5. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Reorganized and edited the chapter ■ Added links to Quartus II Help for procedural information previously included in the chapter ■ Added information on rules marked Inapplicable in the I/O Rules Matrix Report ■ Added information on assigning slew rate and drive strength settings to pins to fix I/O assignment warnings
November 2009	9.1.0	■ Reorganized entire chapter to include links to Quartus II help for procedural information previously included in the chapter ■ Added documentation on near-end and far-end advanced I/O timing
March 2009	9.0.0	■ Updated "Pad View Window" on page 5–20 ■ Added new figures: →Figure 5–15 →Figure 5–16 ■ Added new section "Viewing Simultaneous Switching Noise (SSN) Results" on page 5–17 ■ Added new section "Creating Exclusive I/O Group Assignments" on page 5–18

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

Take an online survey to provide feeback about this handbook chapter.

6. Simultaneous Switching Noise (SSN) Analysis and Optimizations

QII52018-10.0.0

FPGA design has evolved from small programmable circuits to designs that compete with multimillion-gate ASICs. At the same time, the I/O counts on FPGAs and logic density requirements of designs have increased exponentially. The higher-speed interfaces in FPGAs, including high-speed serial interfaces and memory interfaces, require careful interface design on the PCB. Designers must address the timing and signal integrity requirements of these interfaces early in the design cycle. Simultaneous switching noise (SSN) 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 to and out of the FPGA. Altera recommends that you perform SSN analysis early in your FPGA design and prior to the layout of your PCB with complete SSN analysis of your FPGA in the Quartus® II software. This chapter describes the Quartus II SSN Analyzer tool and covers the following topics:

- "Definitions"
- "Understanding SSN" on page 6–2
- "SSN Estimation Tools" on page 6–4
- "SSN Analysis Overview" on page 6–5
- "Optimizing Your Design for SSN Analysis" on page 6–8
- "Performing SSN Analysis and Viewing Results" on page 6–15
- "Decreasing Processing Time for SSN Analysis" on page 6–17

Definitions

The terminology used in this chapter includes the following terms:

Aggressor: An output or bidirectional signal that contributes to the noise for a victim I/O pin

PDN: Power distribution network

QH: Quiet high signal level on a pin

QHN: Quiet high noise on a pin, measured in volts

QL: Quiet low signal level on a pin

QLN: Quiet low noise on a pin, measured in volts

SI: Signal integrity (a superset of SSN, covering all noise sources)

SSN: Simultaneous switching noise

SSO: Simultaneous switching output (which are either the output or bidirectional pins)

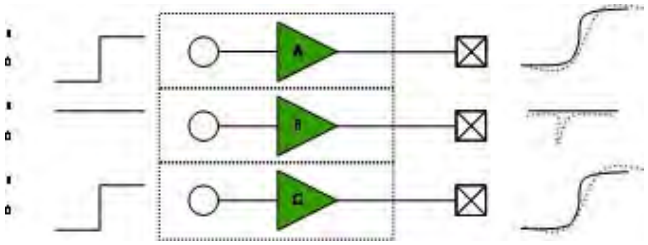
Victim: An input, output, or bidirectional pin that is analyzed during SSN analysis. During SSN analysis, each pin is analyzed as a victim. If a pin is an output or bidirectional pin, the same pin acts as an aggressor signal for other pins.

Understanding SSN

SSN is defined as a noise voltage induced onto a single victim I/O pin on a device due to the switching behavior of other aggressor I/O pins on 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 without noise interference, as shown by 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 timing noise on pins A and C, as shown by the dotted curves in the figure.

Figure 6–1. System with Three Pins



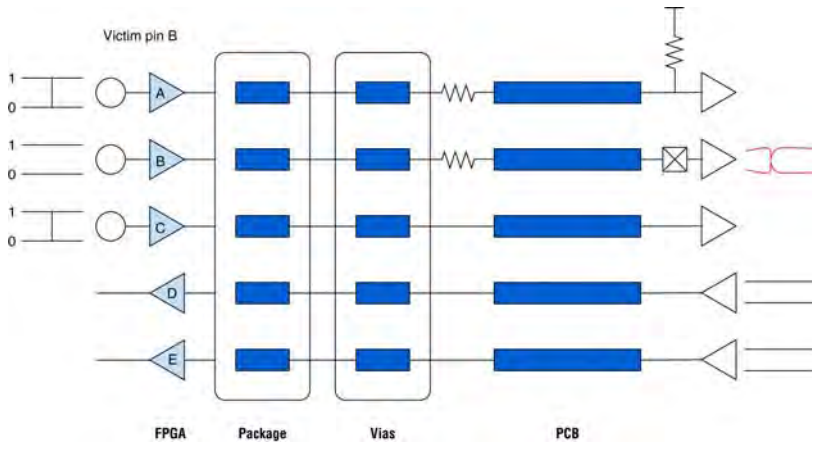
Voltage noise is measured as the worst-case change in voltage of a signal due to SSN. When a signal is QH, it is measured as the change in voltage toward 0 V. When a signal is QL, it is measured as the change in voltage toward V_{CC}.

In the Quartus II software, only voltage noise is analyzed. Voltage noise can be caused by SSOs under two worst-case conditions:

- The victim pin is high and the aggressor pins (SSOs) are switching from low to high
- The victim pin is low and the aggressor pins (SSOs) are switching from high to low

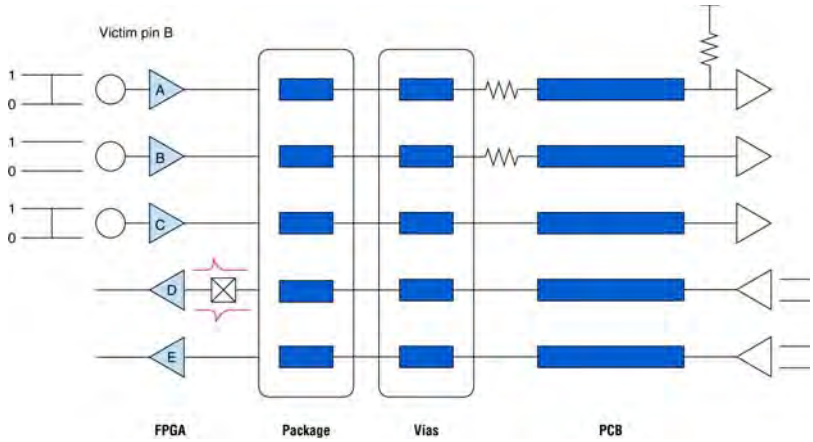
For outputs, the noise is computed at the far-end receiver for pin B (refer to Figure 6–2).

Figure 6–2. Quiet High Output Noise Estimation



For inputs, the noise is computed at the FPGA bumps as shown in for pin D (refer to Figure 6–3).

Figure 6–3. Quiet Low Input Noise Estimation



SSN can occur in any system, but the induced noise does not always result in failures. Voltage functional errors are caused by SSN on quiet victim pins only when the voltage values on the quiet pins change by a large enough voltage that the logic listening to that signal reads a change in the logic value. For QH signals, a voltage functional error occurs when noise events cause the voltage to fall below V_{IH}. Similarly, for QL signals, a voltage functional error occurs when noise events cause the voltage to rise above V_{IL} (refer to Figure 6–4). Because V_{IH} and V_{IL} are different for different I/O standards, and because signals have different quiet voltage values, the absolute amount of SSN, measured 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 (refer to Figure 6–4).

Figure 6–4. Reporting Noise Margins

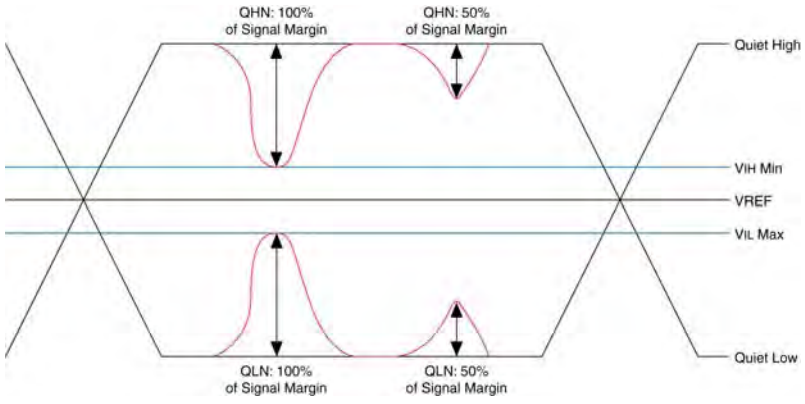
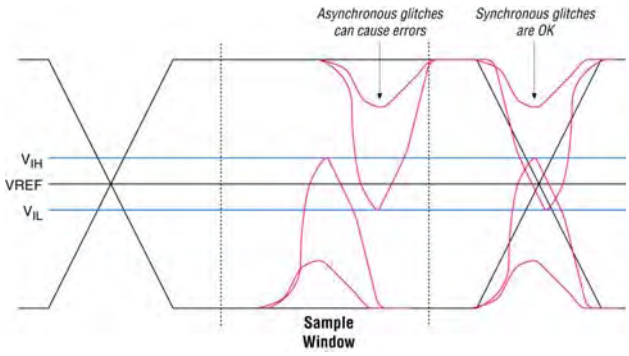


Figure 6–4 shows four noise events, two on QH signals and two on QL signals. The two noise events on the right-side of the figure consume 50 percent of the signal margin and do not cause voltage functional errors. However, the two noise events on the left side of the figure consume 100 percent of the signal margin and can cause a voltage functional error.

Figure 6–5 illustrates a synchronous voltage noise event that does not result in a voltage functional error. Noise or glitches caused by aggressor signals are synchronously related to the victim pin outside of the sampling window of a receiver. The noise or glitches affect the switching time of a victim pin, but are not considered an input threshold violation failure.

Figure 6–5. Synchronous Voltage Noise



For more information about the design factors that affect the noise margins during SSN analysis in the Quartus II software, refer to "SSN Analysis Overview".

SSN Estimation Tools

Addressing SSN early in your FPGA design and PCB layout can help you avoid costly board respins and lost time, both of which can impact your time-to-market. Altera provides many tools for SSN analysis and estimation, including the following tools:

- SSN characterization reports
- An early SSN estimation (ESE) tool
- The SSN Analyzer in the Quartus II software

For more information about the SSN characterization reports and the ESE tool, including device support information, refer to the Signal Integrity Center page of the Altera website.

For more information about the devices for which you can run the SSN Analyzer, refer to *About the SSN Analyzer* in Quartus II Help.

The ESE tool is useful for preliminary SSN analysis of your FPGA design; for more accurate results, however, you must use the SSN Analyzer in the Quartus II software. Table 6–1 compares some of the differences between the ESE tool and the SSN Analyzer.

Table 6–1. Comparison of ESE Tool and SSN Analyzer Tool

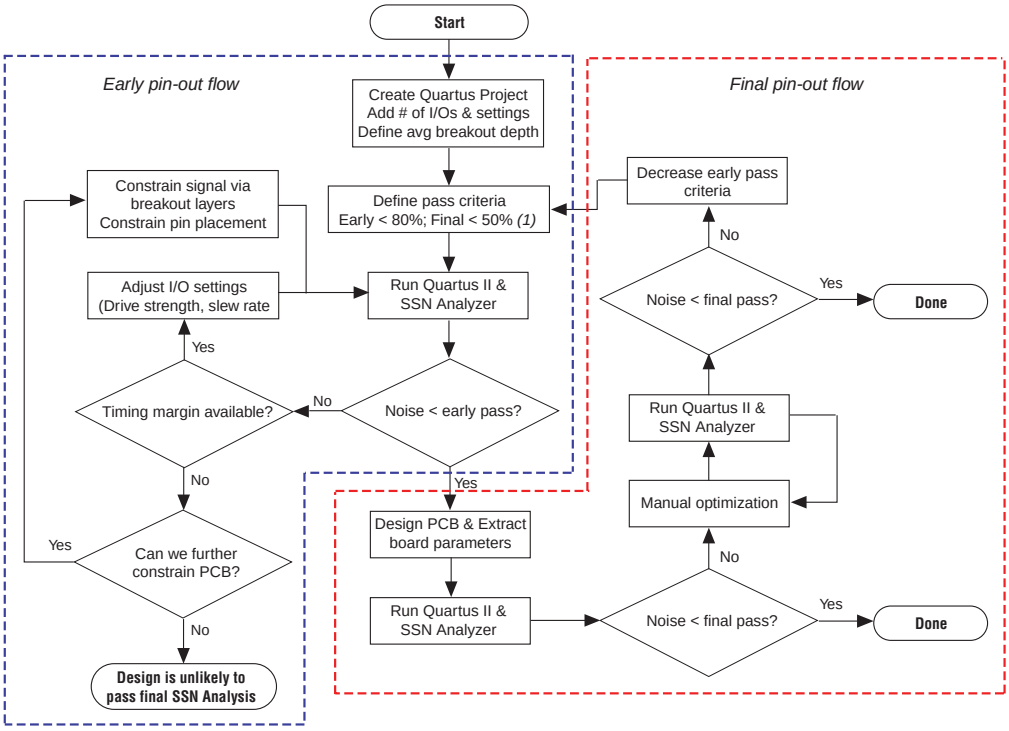
ESE Tool	SSN Analyzer
Is not integrated with the Quartus II software.	Integrated with the Quartus II software, allowing you to perform preliminary SSN analysis while making I/O assignment changes in the Quartus II software.
QL and QH levels are computed assuming a worst-case pattern of I/O placements.	QL and QH levels are computed based on the I/O placements in your design.
No support for entering board information.	Supports board trace models and board layer information, resulting in a more accurate SSN analysis.
No graphical representation.	Integrated with the Quartus II 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.	Requires you to create a Quartus II software project and provide the top-level port information.

SSN Analysis Overview

You can run the SSN Analyzer at different stages in your design cycle to obtain SSN results. The accuracy of the results depends on the completeness of your design information. Altera recommends that you start SSN analysis early in the design cycle to obtain preliminary results and make adjustments to your I/O assignments, and iterate through the design cycle to finally perform a fully constrained SSN analysis with complete information about your board.

Figure 6–6 shows the flows for both early pin-out and final pin-out SSN analysis. The early pin-out flow assumes conservative design rules initially, and then lets you analyze the design and iteratively apply tighter design rules until SSN analysis indicates your design meets SSN constraints. You must define pass criteria for SSN analysis as a percentage of signal margin in both the early pin-out flow and the final pin-out flow. The pass criteria you define is specific to your design requirements. For example, a pass criterion you might define is a condition that verifies you have sufficient SSN margins in your design. You may require that the acceptable voltage noise on a pin must be below 70% of the voltage level for that pin. The pass criteria for the early-pin out flow may be higher than the final pin-out flow criteria, 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 6–6. Pin-Out Analysis *(Note 1)*



Note to Figure 6–6:

(1) Pass criteria determined by customer requirements.

Performing Early Pin-Out SSN Analysis

In the early stages of your design cycle, before you create pin location for your design, use the early pin-out flow (refer to Figure 6–6) to obtain preliminary SSN analysis results. In order to obtain useful SSN results, you must define the top-level ports of your design, but your design files do not have to be complete.

Performing Early Pin-Out SSN Analysis with the ESE Tool

If you know the I/O standards and signaling standards for your design, you can use the ESE tool to perform an initial SSN evaluation.

For more information about the ESE tool, refer to the Signal Integrity Center page of the Altera website.

Performing Early Pin-Out SSN Analysis with the SSN Analyzer

If you have complete information for the top-level ports of your design, you can use the SSN Analyzer to perform an initial SSN evaluation. Use the following steps to perform early pin-out SSN analysis:

1. Create a project in the Quartus II software.
2. Specify your top-level design information either in schematic form or in HDL code.
3. Perform Analysis and Synthesis.

4. Create I/O assignments, such as I/O standard assignments, for the top-level ports in your design.

   ☞ Do not create pin location assignments. The Fitter automatically creates optimized pin location assignments.

5. If you do not have completed design files and timing constraints, run I/O assignment analysis.

   ☞ During I/O assignment analysis, the Fitter places all the unplaced pins on the device, and checks all the I/O placement rules.

6. Run the SSN Analyzer.

For more information about creating and managing projects, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*. For more about generating a top-level design file in the Quartus II software and I/O assignment analysis, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

In the early stages of your design cycle, you may not have complete board information, such as board trace parameters, layer information, and the signal breakout layers. If you run the SSN Analyzer without this specific information, it uses default board trace models and board layer information for SSN analysis, and as a result the SSN Analyzer confidence level is low. If the noise amounts are larger than the pass criteria for early pin-out SSN analysis, verify whether the SSN noise violations are true failures or false failures. For example, sometimes the SSN Analyzer can determine whether pins are switching synchronously and use that information to filter false positives; however, it may not be able to determine all the synchronous groups. You can improve the SSN analysis results by adjusting your I/O assignments and other design settings. After you optimize your design such that it meets the pass criteria for the early pin-out flow, you can then begin to design your PCB.

For more information, refer to "Optimizing Your Design for SSN Analysis".

Performing Final Pin-Out SSN Analysis

You perform final pin-out SSN analysis after you place all the pins in your design, or the Fitter places them for you, and you have complete information about the board trace models and PCB layers. Even if your design achieves sufficient SSN results during early pin-out SSN analysis, you should run SSN analysis with the complete PCB information to ensure that SSN does not cause failures in your final design.You must specify the board parameters in the Quartus II software, including the PCB layer thicknesses, the signal breakout layers, and the board trace models, before you can run SSN analysis on your final assignments.

For more information, refer to "Optimizing Your Design for SSN Analysis".

If the SSN analysis results meet the pass criteria for final pin-out SSN analysis, SSN analysis is complete. If the SSN analysis results do not meet the pass criteria, you must further optimize your design by changing the board and design parameters and then rerun the SSN Analyzer. If the design still does not meet the pass criteria, reduce the pass criteria for early pin-out SSN analysis, and restart the process. By reducing the pass criteria for early pin-out SSN analysis, you place a greater emphasis on reducing

SSN through I/O settings and I/O placement. Changing the drive strength and slew rate of output and bidirectional pins, as well as adjusting the placement of different SSOs, can affect SSN results. Adjusting I/O settings and placement allows the design to meet the pass criteria for final pin-out SSN analysis after you specify the actual PCB board parameters.

Design Factors Affecting SSN Results

There are many factors that affect the SSN levels in your design. The two main factors are the drive strength and slew rate settings of the output and bidirectional pins in your design.

For more information about the factors that contribute to SSN voltage noise in your FPGA design and 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 the Signal Integrity Center page of the Altera website.

Optimizing Your Design for SSN Analysis

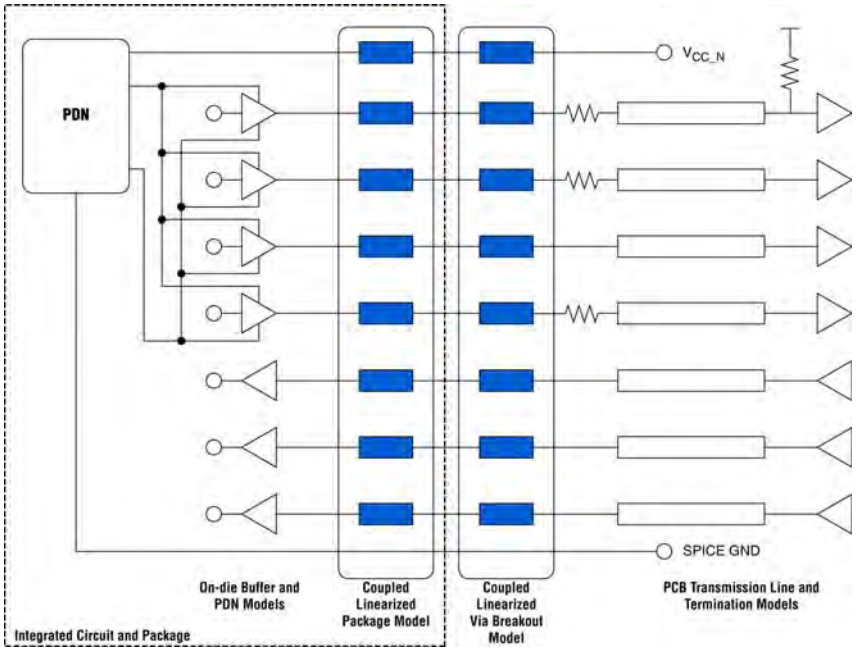
The SSN Analyzer gives you flexibility to precisely define your system to obtain accurate SSN results. The SSN Analyzer produces a voltage noise estimate for each input, output, and bidirectional pin in the design. It allows you to estimate the SSN levels, comprised of QLN and QHN levels, for your FPGA pins. Performing SSN analysis helps you optimize your design for SSN during compilation.

Because the SSN Analyzer is integrated into the Quartus II software, it can automatically set up a system topology that matches your design. The SSN Analyzer accounts for different I/O standards and slew rate settings for each buffer in the design and models different board traces for each signal. Also, it correctly models the state of the unused pins in the design. The SSN Analyzer leverages any custom board trace assignments you set up for use by the advanced I/O timing feature.

The SSN Analyzer also models the package and vias in the design. Models for the different packages that Altera devices support are integrated into the Quartus II software. In the Quartus II software, you can specify different layers on which signals break out, each with its own thickness, and then specify which signal breaks out on which layer.

Figure 6–7 shows the circuit topology the SSN Analyzer automatically constructs. After constructing the circuit topology, the SSN Analyzer uses a simulation-based methodology to determine the SSN for each victim pin in the design.

Figure 6–7. Circuit Topology for SSN Analysis



Optimizing Pin Placements for Signal Integrity

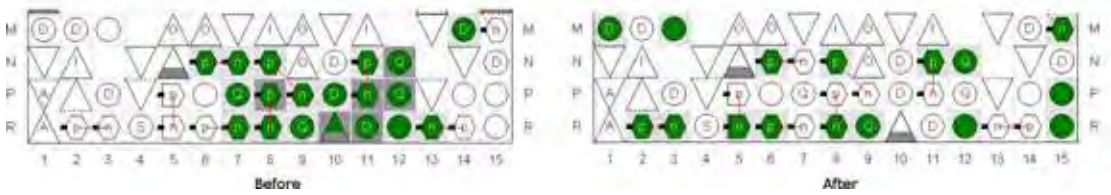
You can take advantage of a built-in SSN optimization feature in the Quartus II software with the **SSN Optimization** logic option.

The I/O placements in your design may be affected when you use this option. Setting this option to **Normal compilation** does not affect the f_{MAX} of your design during compilation, however setting this option to **Extra effort** level may impact your design f_{MAX}.

☞ In order to use the **SSN Optimization** logic option, Altera recommends that you do not create location assignments for your pins; instead, let the Fitter place the pins during compilation so that it places the pins to meet the timing performance of your design. To display the Fitter-placed pins use the Show Fitter Placements feature in the Pin Planner. To accept these suggested pin locations, you must back-annotate your pin assignments.

Figure 6–8 shows the results of turning on the **SSN Optimization** logic option for a design. The image on the left shows the placement of the pins without the **SSN Optimization** logic option, and the image on the right shows the adjustments the Fitter made to pin placements to reduce the amount of SSN in the design when the **SSN Optimization** logic option is turned on.

Figure 6–8. SSN Analysis Results Before and After Using the SSN Optimization Logic Option



For more information about creating project-wide logic option assignments, refer to *Setting Up and Running the Fitter* in Quartus II Help. For more information about the Show Fitter Placements feature, refer to *Show Commands* in Quartus II Help. For more information about back-annotating assignments, refer to *Back-Annotating Assignments for A Project* in Quartus II Help.

For more information about design optimization features, refer to the *Area, Timing, and Power Optimization* section in volume 2 of the *Quartus II Handbook*.

Specifying Board Trace Model Settings

The SSN Analyzer uses circuit models to determine voltage noise during SSN analysis. The circuit topology (refer to Figure 6–7) is incomplete without board trace information and PCB layer information. You must describe the board trace and PCB layer parameters in your design to accurately compute the SSN in your FPGA device. However, if you do not specify some or all of the board trace parameters and PCB layer information, the SSN Analyzer uses default parameters during SSN analysis. When you use the default parameters, the SSN confidence level is low.

For more information about the default parameters used by the SSN Analyzer and SSN confidence levels, refer to "Confidence Metric Details Report" on page 6–16.

The board trace models required for the SSN Analyzer include the board trace termination resistors, pin loads (capacitance), and transmission line parameters. You can define 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 advanced I/O timing.

For more information about defining board trace models and advanced I/O timing, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

You can define an overall board trace model for each I/O standard in your design; this overall board trace model is the default model for all pins that use a particular I/O standard. After configuring the overall board trace model, you can customize the model for specific pins. The parameters you specify for the board trace model are also used in during advanced I/O timing analysis with the TimeQuest Timing Analyzer. If you already specified the board trace models as part of your advanced I/O timing assignments, the same parameters are used during SSN analysis.

For more information about defining a board trace model for your entire design, refer to *Using Advanced I/O Timing* in Quartus II Help. For more information about configuring specific pins in your board trace model, refer to *Creating Board Trace Model Assignments in the Pin Planner* in Quartus II Help. For more information about configuring component values for a board trace model, including a complete list of the supported unit prefixes and setting the values with Tcl scripts, refer to *Board Trace Model* in Quartus II Help.

All the assignments for board trace models you specify are saved to the **.qsf**. You can also use Tcl commands to create board trace model assignments. Example 6–1 shows Tcl commands for specifying transmission line parameters.

Example 6–1. Specifying Board Trace Models

```
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]
```

The best way to calculate transmission line parameters is to use a two-dimensional solver to estimate the inductance per inch and capacitance per inch for the transmission line. 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, you can obtain the far-end loading information in the data sheet for your target device.

For more information, refer to the *Device Family Data Sheet* in the appropriate device handbook available on the Literature and Technical Documentation page of the Altera website.

Defining PCB Layers and PCB Layer Thickness

Every PCB is fabricated using a number of layers. To remove some of the pessimism from your SSN results, Altera recommends that you create assignments describing your PCB layers in the Quartus II software. You can specify the number of layers on you PCB, and their thickness. The PCB layer information is used only during SSN analysis and is not used in other processes run by the Quartus II software. If a custom PCB breakout region is not described you can select the default thickness, which directs the SSN Analyzer to use a single-layer PCB breakout region during SSN analysis.

For more information about specifying PCB layer information, refer to *Running the SSN Analyzer* in Quartus II Help.

All the assignments you create for the PCB layers are saved to the **.qsf**. You can also use Tcl commands to create PCB layer assignments. You can create any number of PCB layers, however, the layers must be consecutive. Example 6–2 shows Tcl commands for specifying PCB layer assignments.

Example 6–2. Specifying PCB Layer 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
```

Figure 6–9 shows the layout cross-section of a PCB in the Cadence Allegro PCB tool. The cross-section shows the stackup information of a PCB, which tells you the number of layers used in your PCB. The PCB shown in this example consists of various signal and circuit layers on which FPGA pins are routed, as well as the power and ground layers.

Figure 6–9. Snapshot of Stackup of a PCB Shown in the Allegro Board Design Environment

Layout Cross Section

Cross Section

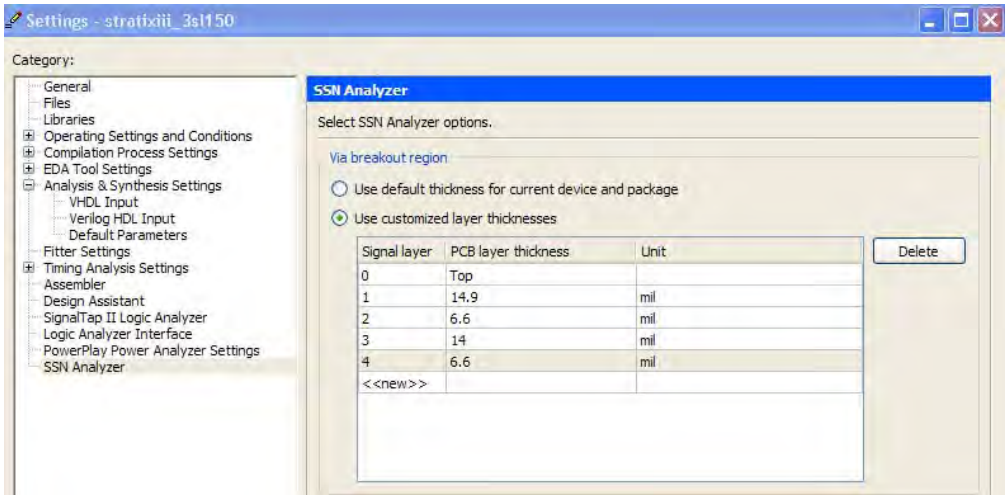
	Subclass Name	Type	Material	Thickness (MIL)
1		SURFACE	AIR	
2	TOP	CONDUCTOR	PLATED_COPPER_FOIL	1.9
3		DIELECTRIC	FR-4	3.6
4	L2-PWR	PLANE	COPPER	1.2
5		DIELECTRIC	FR-4	3
6	L3-GND	PLANE	COPPER	1.2
7		DIELECTRIC	FR-4	4
8	L4-SIGNAL	CONDUCTOR	COPPER	0.6
9		DIELECTRIC	FR-4	6
10	L5-SIGNAL	CONDUCTOR	COPPER	0.6
11		DIELECTRIC	FR-4	4
12	L6 GND	PLANE	COPPER	1.2
13		DIELECTRIC	FR-4	3
14	L7-PWR	PLANE	COPPER	1.2
15		DIELECTRIC	FR-4	4
16	L8-SIGNAL	CONDUCTOR	COPPER	0.6
17		DIELECTRIC	FR-4	6
18	L9-SIGNAL	CONDUCTOR	COPPER	0.6
19		DIELECTRIC	FR-4	4
20	L10-GND	PLANE	COPPER	1.2
21		DIELECTRIC	FR-4	3

In this example, each of the four signal layers are a different thickness, with the depths shown in the **Thickness (MIL)** column. The layer thickness for each signal layer is computed as follows:

- Signal Layer 1 is the L4-SIGNAL, at thickness (1.9+3.6+1.2+3+1.2+4=) 14.9 mils
- Signal Layer 2 is the L5-SIGNAL, at thickness (0.6+6=) 6.6 mils
- Signal Layer 3 is the L8-SIGNAL, at thickness (0.6+4+1.2+3+1.2+4=) 14 mils
- Signal Layer 4 is the L9-SIGNAL, at thickness (0.6+6=) 6.6 mils

Figure 6–10 shows the results in the Quartus II software after you enter these PCB signal layers and thickness assignments.

Figure 6–10. PCB Layers Specified in the Quartus II Software



Specifying Signal Breakout Layers

Each user I/O pin in your FPGA device can break out at different layers on your PCB. In the Pin Planner, you can specify on which layers the I/O pins in your design break out. The breakout layer information is used only during SSN analysis and is not used in other processes run by the Quartus II software. To assign a pin to PCB layer, follow these steps:

1. On the Assignments menu, click **Pin Planner**.
2. If necessary, perform Analysis & Elaboration, Analysis & Synthesis, or fully compile the design to populate the Pin Planner with the node names in the design.
3. Right-click anywhere in the **All Pins** or **Groups** list, and then click **Customize Columns**.
4. Select the **PCB layer** column and move it from the **Available columns** list to the **Show these columns in this order** list.
5. Click **OK**.
6. In the **PCB layer** column, specify the PCB layer to which you want to connect the signal.
7. On the File menu, click **Save Project** to save the changes.

When you create PCB breakout layer assignments in the Pin Planner, you can assign the pin to any layer, even if you did not yet define the PCB layer.

Creating I/O Assignments

I/O assignments are required in FPGA design and are also used during SSN analysis to estimate voltage noise. 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, drive strength, and slew rate. The SSN Analyzer supports most I/O standards in a device family, such as the **LVTTL** and **LVCMOS** I/O standards.

The SSN Analyzer does not support differential I/O standards, such as the **LVDS** I/O standard and its variations, because differential I/O standards contribute a small amount of SSN.

For more information about supported I/O standards, refer to the appropriate device handbook available on the Literature and Technical Documentation page of the Altera website.

For more information about creating and managing I/O assignments, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Decreasing Pessimism in SSN Analysis

In the absence of specific timing information, the SSN Analyzer analyzes your design under worst-case conditions. Worst-case conditions include all pins acting as aggressor signals on all possible victim pins and all aggressor pins switching with the worst possible timing relationship. The results of SSN analysis under worst-case conditions are very pessimistic. You can improve the results of SSN Analysis by creating group assignments for specific types of pins. Use the following group assignments to decrease the pessimism in SSN analysis results:

- Assign pins to an output enable group—All pins in an output enable group must be either all input pins or all output pins. If all the pins in a group are always either all inputs or all outputs, it is impossible for an output pin in the group to cause SSN noise on an input pin in the group. You can assign pins to an output enable group with the **Output Enable Group** logic option.
- Assign pins to a synchronous group—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. You can assign pins to an output enable group with the **Synchronous Group** logic option. For example, in your design you have a bus with 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 groupmates does not matter, because it does not fall in the sampling window. If you do not assign the bus to a synchronous group, the other 31 pins can act as aggressors for the first pin in that group, leading to higher QL and QH noise levels during SSN analysis.

In some cases, the SSN Analyzer can detect the grouping for bidirectional pins by looking at the output enable signal of the bidirectional pins. However, Altera recommends that you explicitly specify the bidirectional groups and output groups in your design.

For more information about creating logic option assignments, refer to *Assigning Pins* in Quartus II Help.

Excluding Pins as Aggressor Signals

The SSN Analyzer uses the following conditions to exclude pins as aggressor signals for a specific victim pin:

- A pin that is a complement of the victim pin. For example, any pin that is assigned a differential I/O standard cannot be an aggressor pin.
- A programming pin or JTAG pin because these pins are not active in user mode.
- Pins that have the same output enable signal as a bidirectional victim pin that the SSN Analyzer analyzes as an input pin. Pins with the same output enable signal also act as input pins and therefore cannot be aggressor pins at the same time. For information about grouping bidirectional pins, refer to "Performing SSN Analysis and Viewing Results".
- Pins in the same synchronous group as a victim output pin. For information about grouping output pins, refer to "Performing SSN Analysis and Viewing Results".
- A pin assigned the **I/O Maximum Toggle Rate** logic option with a frequency setting of zero. The SSN Analyzer does not consider pins with this setting as aggressor pins.

For more information about creating pin assignments with the Pin Planner, refer to *Assigning Pins* in Quartus II Help.

Performing SSN Analysis and Viewing Results

You can perform SSN analysis either on your entire design, or you can limit the analysis to specific I/O banks.

If you know the problem area for SSN is within one I/O bank and you are changing pin assignments only in that bank, you can run SSN analysis for just that one I/O bank to reduce analysis time.

For more information, refer to *Running the SSN Analyzer* in Quartus II Help.

For more information about I/O bank numbering, refer to the appropriate device handbook available on the Literature and Technical Documentation page of the Altera website.

Understanding the SSN Reports

When SSN analysis is complete, you can view detailed analysis reports. The detailed messages in the reports help you understand and resolve SSN problems.

The SSN Analyzer section of the Compilation report contains information generated during SSN analysis, including the following reports:

- Summary
- Output Pins
- Input Pins
- Unanalyzed Pins
- Confidence Metric Details

Summary Report

The Summary report summarizes the SSN Analyzer status and rates the SSN Analyzer confidence level as low, medium, or high. The confidence level depends on the completeness of your board trace model assignments. 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 noise levels on a victim pin. The accuracy of QH and QL noise levels depends the accuracy of your board trace model assignments.

Output Pins and Input Pins Reports

The Output Pins report lists all of the output pins and bidirectional pins that are treated as output pins during SSN analysis. The Input Pins report lists all of the input pins and bidirectional pins that are treated as inputs during SSN analysis. Both reports list the location assignments for the pins treated as SSN outputs or inputs during SSN analysis, 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.

Unanalyzed Pins Report

Not all pins are analyzed for SSN analysis. The following pins are not analyzed and are reported in the Unanalyzed Pins report:

- Pins assigned the **LVDS** I/O standard or any LVDS variations, such as the **mini-LVDS** I/O standard
- Pins created in the migration flow that cover power and supply pins in other packages
- The negative terminals of pseudo-differential I/O standards; the noise on differential standards is reported as the differential noise and is reported on the positive terminal

Confidence Metric Details Report

The Confidence Metric Details Report lists the values used during SSN Analysis for unspecified I/O, board, and PCB assignments.

Viewing SSN Analysis Results in the Pin Planner

After SSN analysis completes, you can analyze the results in the Pin Planner. In the Pin Planner you can identify the SSN hotspots in your device, as well as the QL and QH noise levels. The QL and QH results for each pin are displayed with a different color that represents whether the pin is below the warning threshold, below the critical threshold, or above the critical threshold. This color representation is also referred to as the SSN map of your FPGA device.

When you view the SSN map, you can customize which details to display, including input pins, output pins, QH signals, QL signals, and noise levels. You can also adjust the threshold levels for QH and QL noise voltages. Adjusting the threshold levels in the Pin Planner does not change the threshold levels reported during SSN analysis and does not change the data in any of the SSN reports.

You can also you change I/O assignments and board trace information and rerun the SSN Analyzer to view the SSN analysis results based on those modified settings.

For more information, refer to *Show SSN Analyzer Results* and *Running the SSN Analyzer* in Quartus II Help.

Decreasing Processing Time for SSN Analysis

FPGA designs are getting larger in density, logic, and I/O count. The time it takes to complete SSN analysis and other Quartus II software processes affects your development time. Faster processing times can reduce your design cycle time. Use the following guidelines to reduce processing time:

- Direct the Quartus II software to use more than one processor for parallel executables, including the SSN Analyzer
- Perform SSN analysis after I/O assignment analysis if your design files and constraints are complete, and you are interested in generating the SSN results early in the design process and want to adjust I/O placements to see if you can obtain better results
- Perform SSN analysis after fitting if you want to view preliminary SSN results that do not take into account complete I/O assignment and I/O timing results
- Perform engineering change orders (ECOs) on your design, rather than recompiling the entire design, if you want to rerun SSN analysis after changing I/O assignments

For more information about using parallel processors, refer to *Setting Up and Running Analysis and Synthesis* and *Compilation Process Settings Page* in Quartus II Help. For more information about performing I/O assignment analysis, refer to *Assigning Pins* in Quartus II Help. For more information about running the Fitter, refer to *Setting Up and Running the Fitter* in Quartus II Help.

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*.

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. The Quartus II software provides several packages to compile your design and create 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 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* and *API Functions for Tcl* in Quartus II Help.

Optimizing Pin Placements for Signal Integrity

You can create an assignment that directs the Fitter to optimize pin placements for signal integrity with a Tcl command.

The following Tcl command directs the Fitter to optimize pin placement for signal integrity without affecting design f_{MAX}:

```
set_global_assignment -name OPTIMIZE_SIGNAL_INTEGRITY "Normal
Compilation"
```

For more information, refer to "Optimizing Pin Placements for Signal Integrity" on page 6–9.

Defining PCB Layers and PCB Layer Thickness

You can create PCB layer and thickness assignments with a Tcl command. shows Tcl commands for specifying PCB layer assignments.

Example 6–3. Specifying PCB Layer 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 Tcl commands specify that there are seven PCB layers in the design, each with a different thickness. In each assignment, the letter `M` indicates the unit of measurement is millimeters. When you specify PCB layer assignments with Tcl commands, you must list the layers in consecutive order. For example, you would receive an error during SSN Analysis if your Tcl commands created 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
```

To create assignments with the unit of measurement in mils, refer to the syntax in the following Tcl commands. These Tcl commands specify the same settings as shown in Figure 6–10 on page 6–13.

```
set_global_assignment -name PCB_LAYER_THICKNESS 14.9MIL -section_id 1
set_global_assignment -name PCB_LAYER_THICKNESS 6.6MIL -section_id 2
set_global_assignment -name PCB_LAYER_THICKNESS 14MIL -section_id 3
set_global_assignment -name PCB_LAYER_THICKNESS 6.6MIL -section_id 4
```

For more information, refer to "Defining PCB Layers and PCB Layer Thickness" on page 6–11.

Specifying Signal Breakout Layers

You can create signal breakout layer assignments with a Tcl command. Example 6–4 shows Tcl commands for specifying signal breakout layer assignments:

Example 6–4. Specifying Signal Breakout Layer Assignments

```
set_instance_assignment -name PCB_LAYER 10 -to e[2]
set_instance_assignment -name PCB_LAYER 3 -to e[3]
```

When you create PCB breakout layer assignments with Tcl commands, if you do not specify a PCB layer, or if you specify a PCB layer that does not exist, the SSN Analyzer breaks out the signal at the bottommost PCB layer.

If you create a PCB layer breakout assignment to a layer that does not exist, the SSN Analyzer will generate a warning message.

For more information, refer to "Specifying Signal Breakout Layers" on page 6–13.

Decreasing Pessimism in SSN Analysis

You can create output enable group and synchronous group assignments to help decrease pessimism during SSN Analysis with a Tcl command.

The following Tcl command assigns the bidirectional bus `DATAINOUT` to an output enable group:

```
set_instance_assignment -name OUTPUT_ENABLE_GROUP 1 -to DATAINOUT
```

The following Tcl command assigns the bus `PCI_ADD_io` to a synchronous group:

```
set_instance_assignment -name SYNCHRONOUS_GROUP 1 -to PCI_AD_io
```

For more information, refer to "Decreasing Pessimism in SSN Analysis" on page 6–14.

Performing SSN Analysis

You can perform SSN analysis with a command-line command. Use the `quartus_si` package that is provided with the Quartus II software.

Type the following command at a system command prompt to start the SSN Analyzer:

```
quartus_si <project name> ↵
```

To analyze just one I/O bank, type the following command at a system command prompt:

```
quartus_si <project revision> <--bank = bank id> ↵
```

For example, to run analyze the I/O bank 2A type the following command:

```
quartus_si counter --bank=2A ↵
```

For more information, refer to "Performing SSN Analysis and Viewing Results" on page 6–15.

For more information about the `quartus_si` package, type `quartus_si -h` at a system command prompt.

Conclusion

To assist you with SSN Analysis, you can use the fast and accurate SSN Analyzer to help you 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 the tools you need to ensure your FPGA design meets your SSN requirements.

Document Revision History

Table 6–2 shows the revision history for this chapter.

Table 6–2. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Reorganized and edited the chapter ■ Added links to Quartus II Help for procedural information previously included in the chapter
November 2009	9.1.0	■ Added "Figure 6–9 shows the layout cross-section of a PCB in the Cadence Allegro PCB tool. The cross-section shows the stackup information of a PCB, which tells you the number of layers used in your PCB. The PCB shown in this example consists of various signal and circuit layers on which FPGA pins are routed, as well as the power and ground layers." on page 6–12 ■ Updated for the Quartus II software 9.1 release
March 2009	9.0.0	Initial release

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

Take an online survey to provide feedback about this handbook chapter.

7. Signal Integrity Analysis with Third-Party Tools

QII53020-10.0.0

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 7–16

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

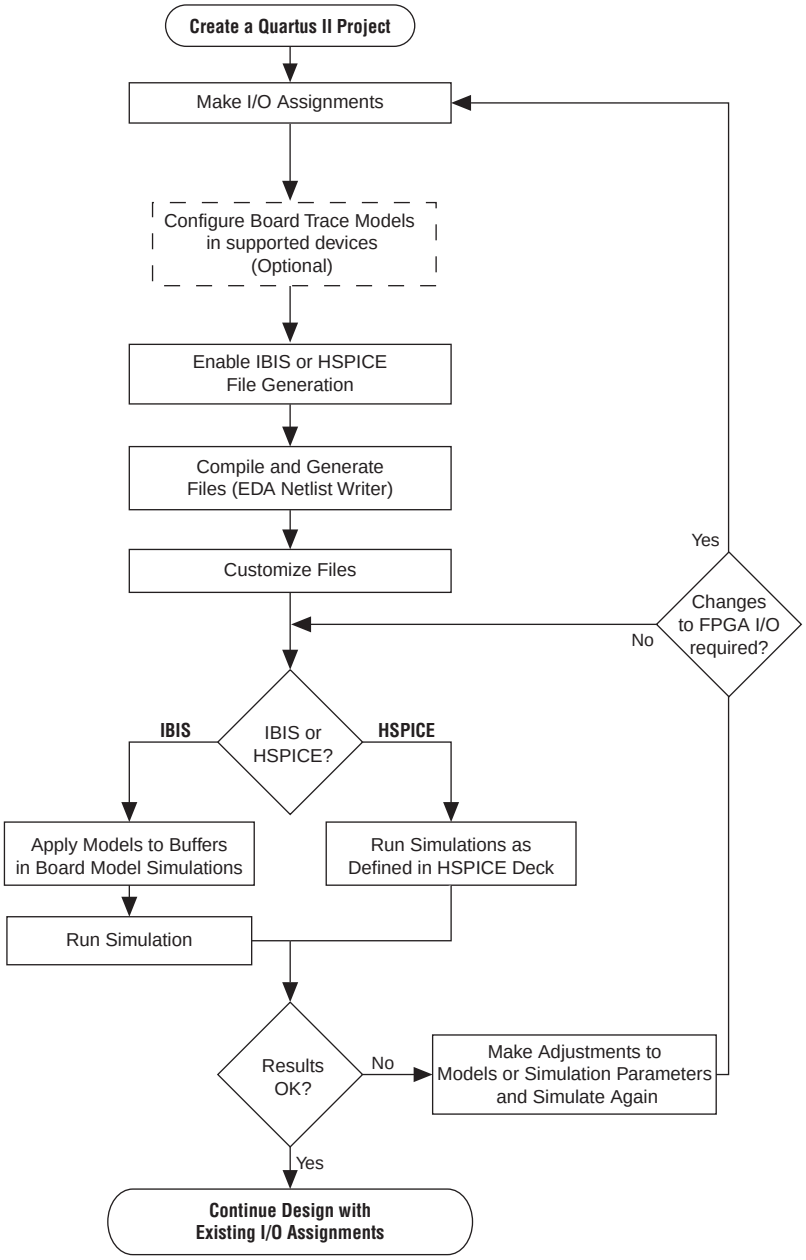
Board signal integrity analysis can take place at any point in the FPGA design process and is often performed before and after board layout. If it is performed early in the process as part of a pre-PCB layout analysis, the models used for simulations can be more generic and can be changed as much as 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.

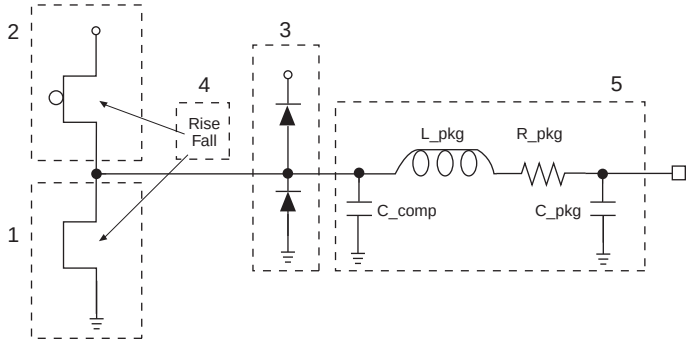
Simulation with IBIS Models

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.

For step-by-step instructions on how to generate IBIS models with the Quartus II software, refer to *Generating IBIS Output Files with the Quartus II Software* in Quartus II Help.

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

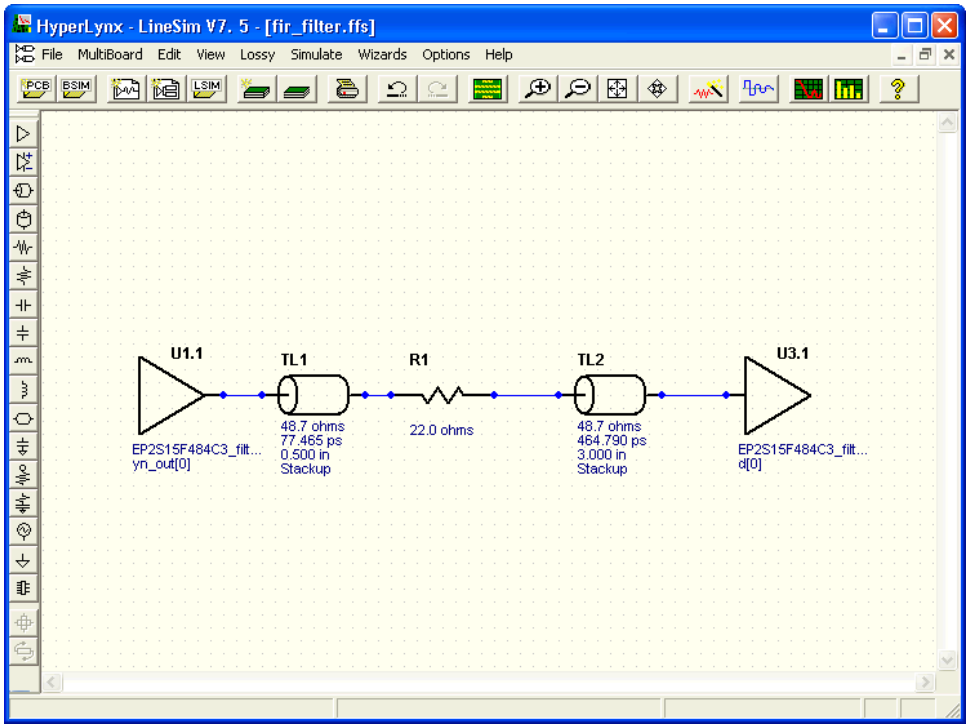
You must integrate IBIS models downloaded from the Altera website (www.altera.com) or created with the Quartus II IBIS Writer into board design simulations to accurately model timing and signal integrity. The HyperLynx software from Mentor Graphics is one of the most popular tools for design simulation. 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 7–3. 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 7–3. 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.

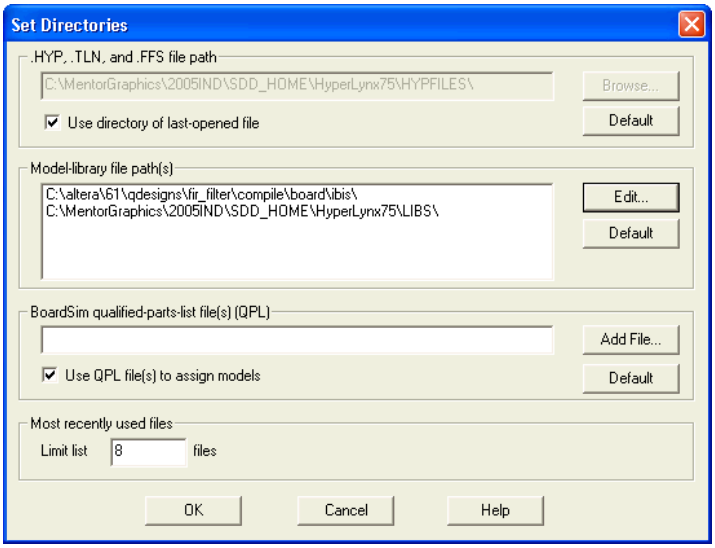
Configuring LineSim to Use Altera IBIS Models

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:

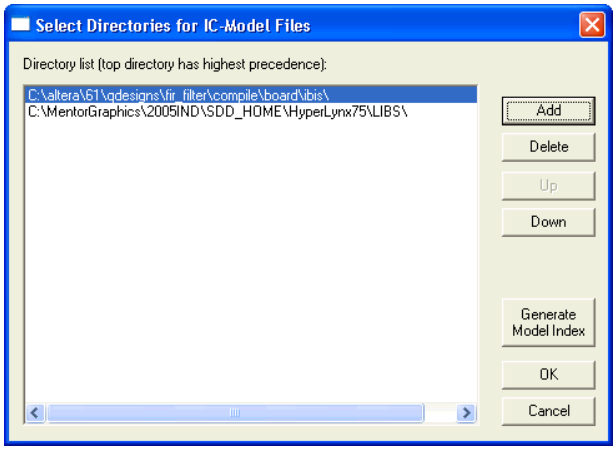
1. From the Options menu, click **Directories**. The **Set Directories** dialog box appears (Figure 7–4). The **Model-library file path(s)** list displays the order in which LineSim searches file directories for model files.

Figure 7–4. 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 7–5).

Figure 7–5. LineSim Select Directories Dialog Box



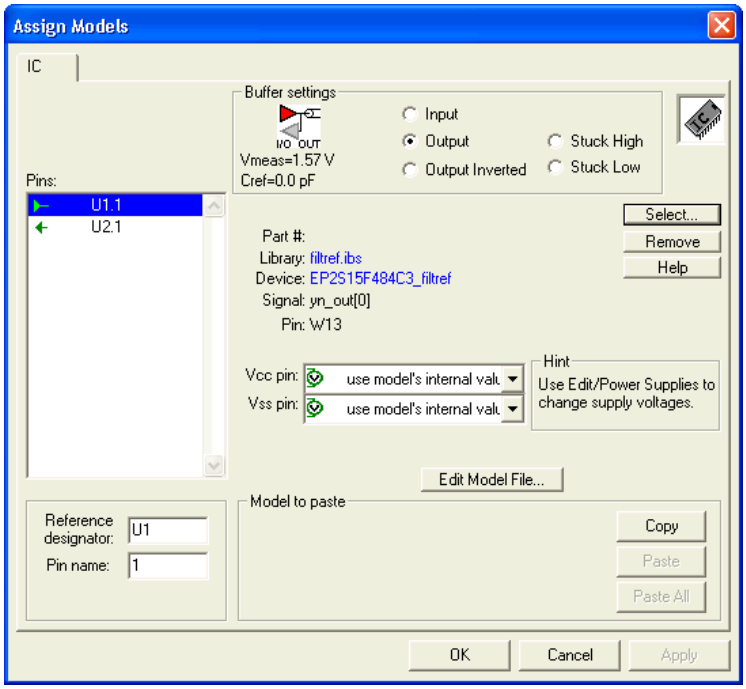
3. Click **Add**
4. Browse to the default IBIS model location, <*project directory*>**/board/ibis**. Click **OK**.
5. Click **Up** to move the IBIS model directory to the top of the list. Click **Generate Model Index** to update LineSim's model database with the models found in the added directory.
6. 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:

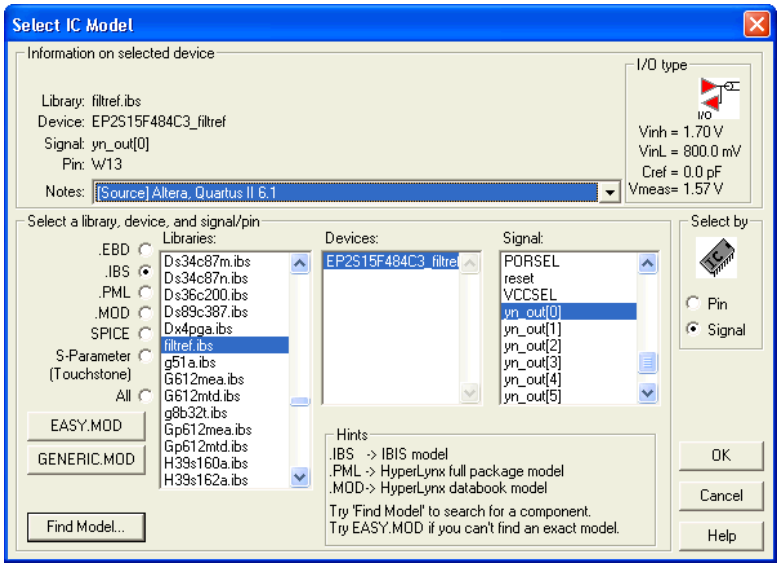
1. Double-click a buffer symbol in your schematic to open the **Assign Models** dialog box (Figure 7–6). You can also click **Assign Models** from the buffer symbol's right-click menu.

Figure 7–6. 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 7–7).

Figure 7–7. LineSim Select IC Model Dialog Box



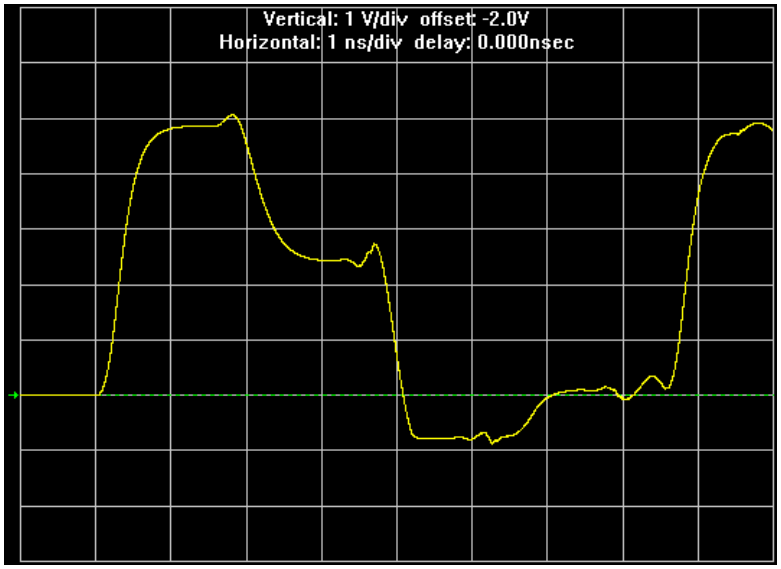
4. To filter the list of available libraries to display only IBIS models, select **.IBS**. Scroll through the **Libraries** list, and click the name of the library for your design. By default, this is <*project name*>**.ibs**.

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

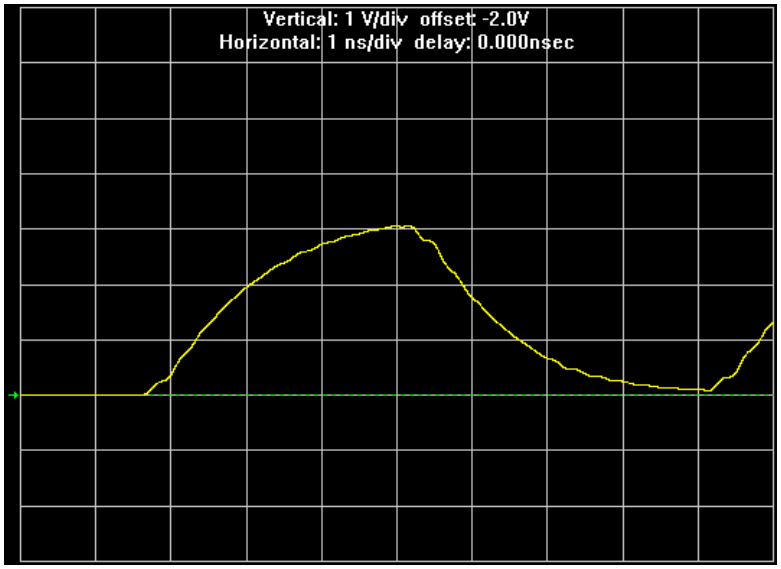
You can now run any desired simulations and make adjustments to the I/O assignments or simulation parameters as required. For example, if you see too much overshoot in the simulated signal at the destination buffer after running a simulation (as shown in Figure 7–8), you could adjust the drive strength 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 7–8. 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 7–9), 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 7–9. 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 7–17. 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.

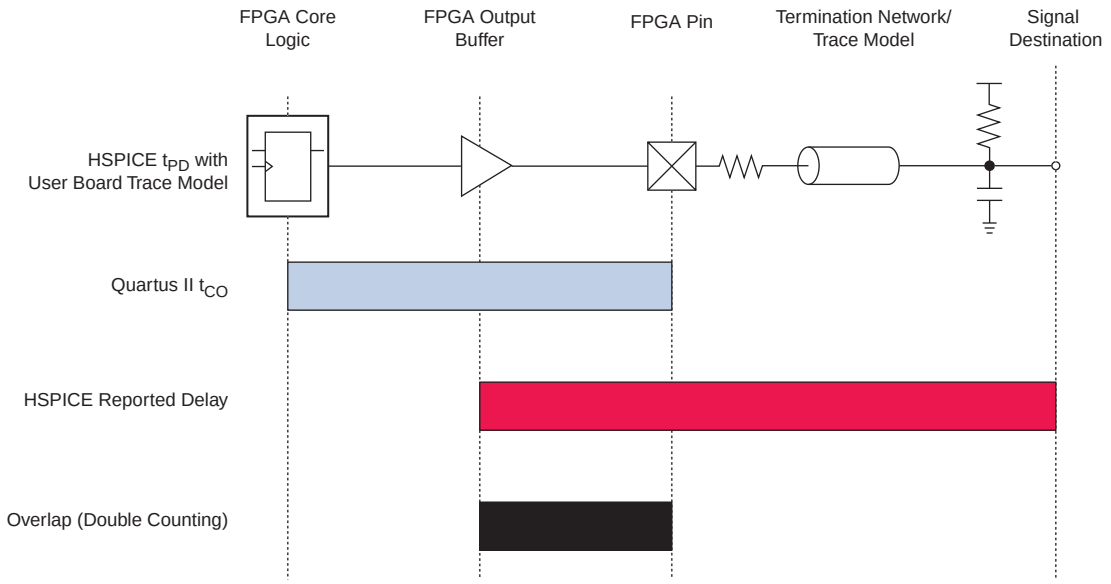
The Double Counting Problem in HSPICE Simulations

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 7–10.

Figure 7–10. 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.

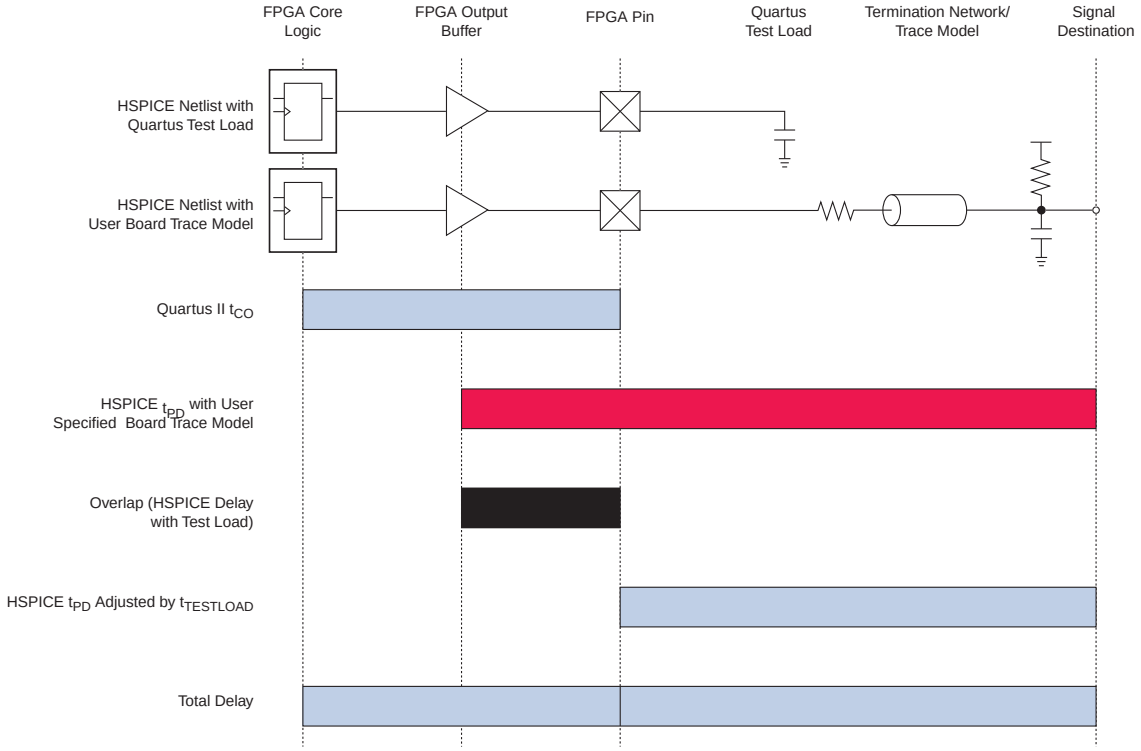
It is apparent immediately that if these two delays were simply added together, the delay between the output buffer and the device pin would be counted twice in the calculation. A model or simulation that does not account for this double count would create overly pessimistic simulation results, because 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 7–11.

Figure 7–11. Common Test Loads Used for Output Timing



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.

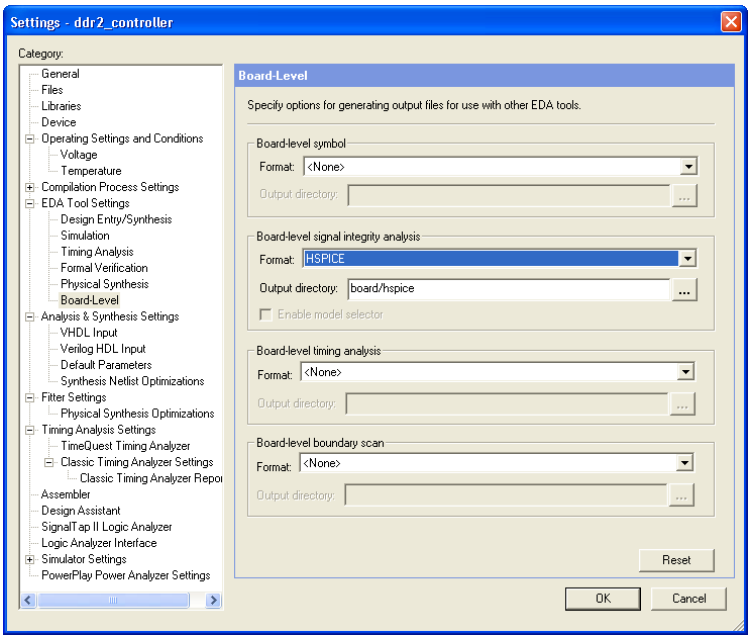
The Quartus II software provides multiple methods for creating these assignments:

- 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 7–12) to generate the HSPICE decks from the Quartus II software.

Figure 7–12. 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

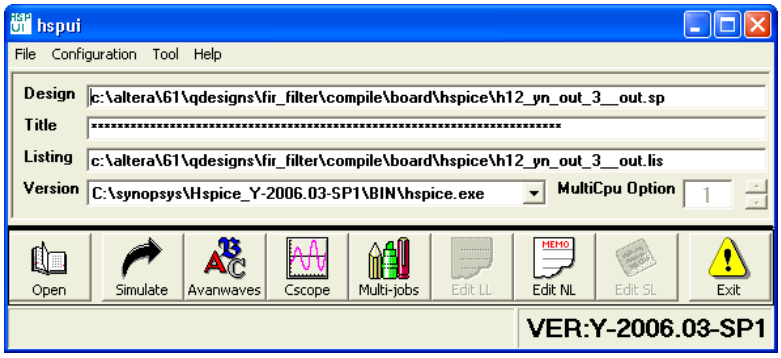
```
* Sample source stimulus placeholder
*  - Replace this with your I/O driver model
```

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 7–13.

Figure 7–13. 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

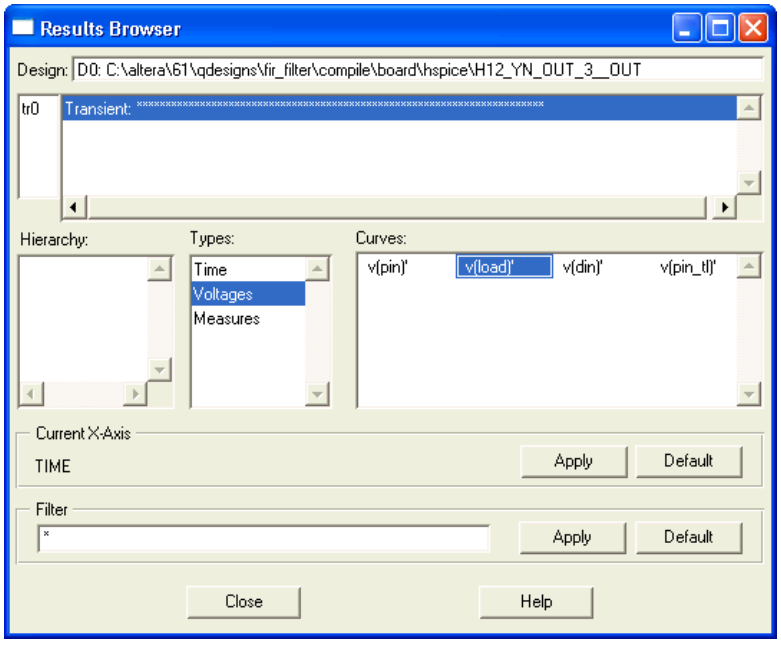
The **.lis** file stores the collected simulation data in tabular form. The default simulation configured by the HSPICE Writer produces delay measurements for rising and falling transitions on both input and output simulations. These measurements are found in the **.lis** file and named `tpd_rise` and `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 7–14.

Figure 7–14. HSPICE AvanWaves Results Browser



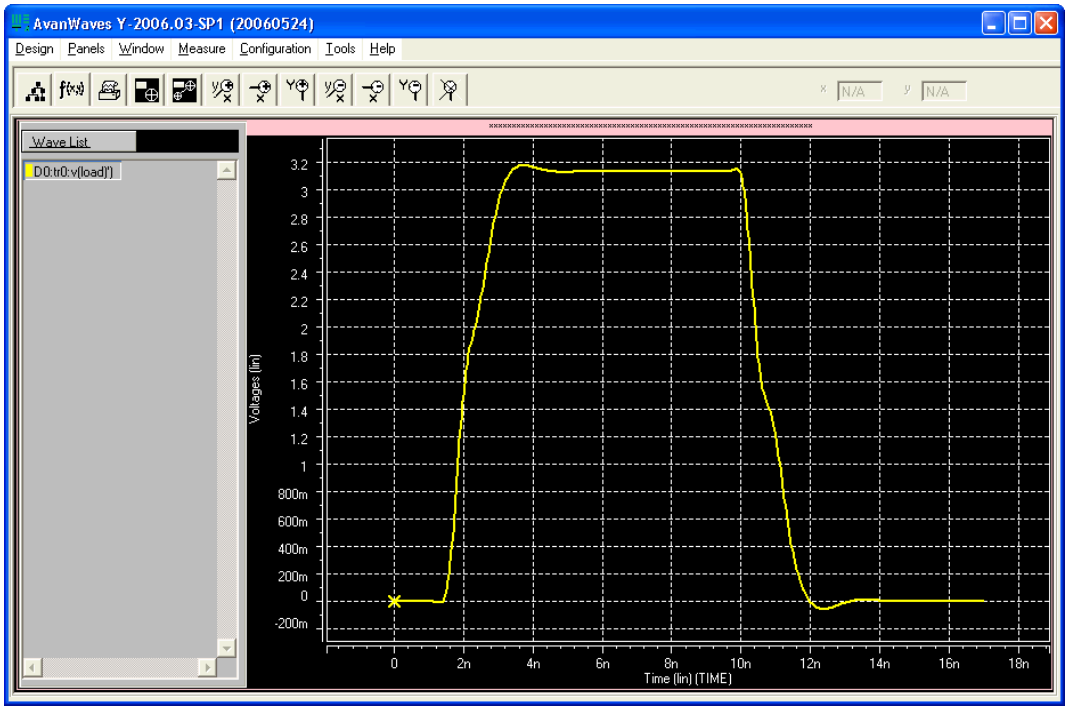
The **Results Browser** lets you select which waveform to view quickly in the main viewing window. If multiple simulations are run on the same signal, the list at the top of the **Results Browser** displays the results of each simulation. Click the simulation description to select which simulation to 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 7–15).

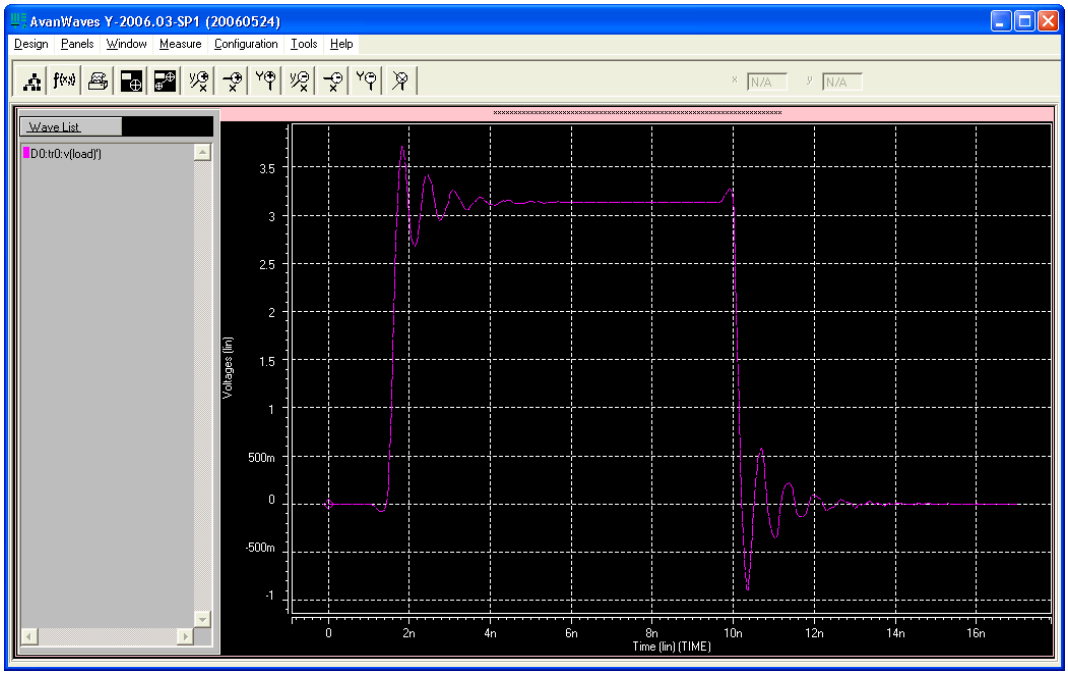
Figure 7–15. AvanWaves Waveform Viewer



Making Design Adjustments Based on HSPICE Simulations

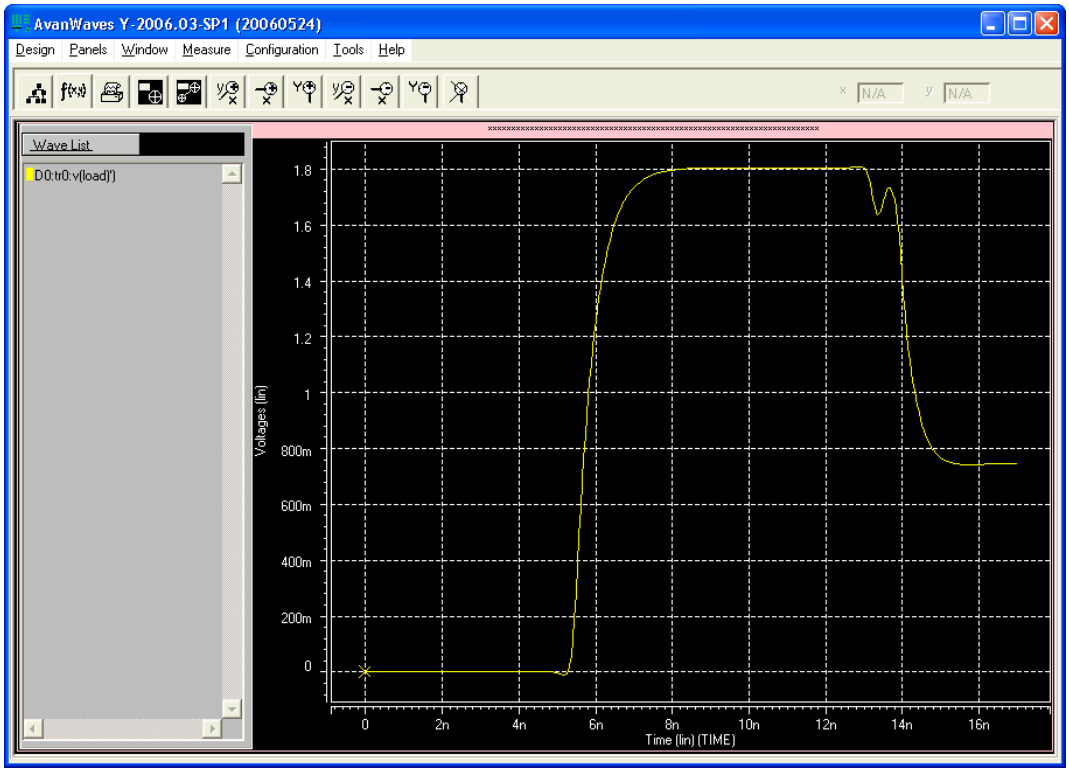
Based on the results of your simulations, you can make adjustments to the I/O assignments or simulation parameters if required. For example, after you run a simulation and see overshoot or ringing in the simulated signal at the destination buffer as shown in the example in Figure 7–16, you can adjust 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.

Figure 7–16. 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 7–17, 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 7–17. 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.

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 7–17.

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 7–17.

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 7–17 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.

Document Revision History

Table 7–3 shows the revision history for this chapter.

Table 7–3. Document Revision History

Date and Document Version	Version	Changes Made
July 2010	10.0.0	■ Updated device support.
November 2009	9.1.0	■ No change to content.
March 2009	9.0.0	■ Was volume 3, chapter 12 in the 8.1.0 release. ■ No change to content.
November 2008	8.1.0	■ Changed to 8-1/2 x 11 page size. ■ Added information for Stratix III devices. ■ Input signals for Cyclone III devices are supported.
May 2008	8.0.0	■ 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.

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

Take an online survey to provide feedback about this handbook chapter.

8. Mentor Graphics PCB Design Tools Support

QII52015-10.0.0

This chapter discusses how the Quartus® II software interacts with the Mentor Graphics I/O Designer software and the DxDesigner software to provide a complete FPGA-to-board design workflow.

With today's large, high-pin-count and high-speed FPGA devices, good and correct PCB design practices are essential to ensure correct system operation. The PCB design takes place concurrently with the design and programming of the FPGA. The FPGA or ASIC designer initially creates signal and pin assignments, and the board designer must correctly transfer these assignments to the symbols in their system circuit schematics and board layout. As the board design progresses, Altera recommends reassigning pins to optimize the PCB layout. Ensure that you inform the FPGA designer of the pin reassignments so that the new assignments are included in an updated placement and routing of the design.

The Mentor Graphics® I/O Designer software allows you to take advantage of the full FPGA symbol design, creation, editing, and back-annotation flow supported by the Mentor Graphics tools.

This chapter covers the following topics:

- Performing 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 for board design and layout engineers who want to start the FPGA board integration while the FPGA is still in the design phase. Alternatively, the board designer can plan the FPGA pin-out and routing requirements in the Mentor Graphics tools and pass the information back to the Quartus II software for placement and routing. Part librarians can also benefit from this chapter by learning how to use output from the Quartus II software to create new library parts and symbols.

The procedures in this chapter require the following software:

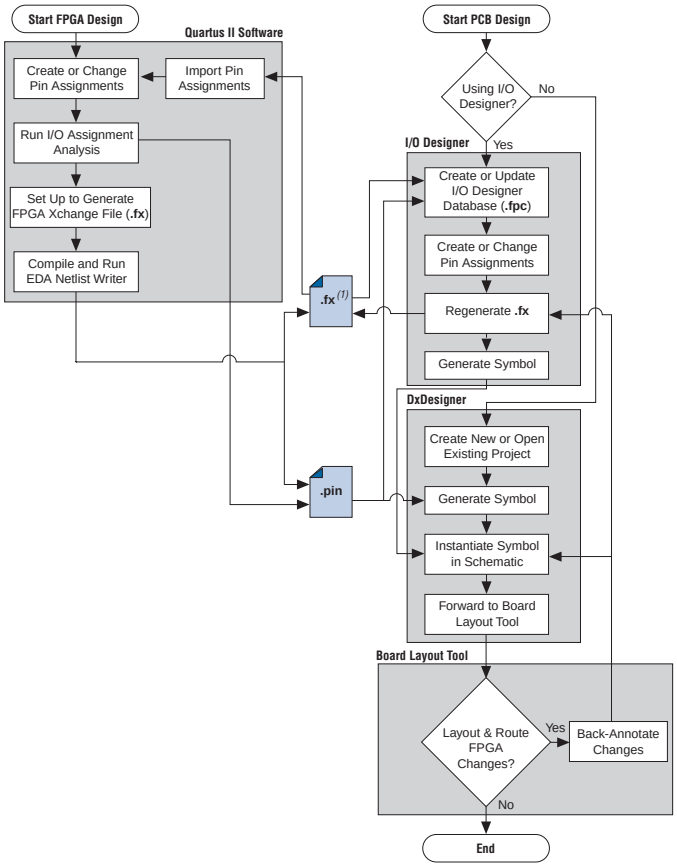
- The Quartus II software version 5.1 or later
- DxDesigner software version 2004 or later
- Mentor Graphics I/O Designer software (optional)

To obtain and license the Mentor Graphics tools and for product information, support, and training, refer to the Mentor Graphics website (www.mentor.com).

FPGA-to-PCB Design Flow

You can 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 **.fx** in the output directory you specify in the **Board-Level** page of the **Settings** dialog box. However, the Quartus II software and the I/O Designer software can import pin assignments from an **.fx** located in any directory. Altera recommends working with a backup **.fx** to prevent overwriting existing assignments or importing invalid assignments.

To perform the design flow shown in Figure 8–1, follow these steps:

1. In the Quartus II software, set up the board-level assignment settings to generate an **.fx** for symbol generation.
2. Compile your design to generate the **.fx** and Pin-Out File (**.pin**). You can locate the generated **.fx** and **.pin** files in the Quartus II project directory.

3. Create a board design with the DxDesigner software and the I/O Designer software by performing the following steps:
   a. Create a new I/O Designer database based on the **.fx** and the **.pin** files.
   b. In the I/O Designer software, make adjustments to signal and pin assignments.
   c. Regenerate the **.fx** in the I/O Designer software to export the I/O Designer software changes to the Quartus II software.
   d. Generate a single or fractured symbol for use in the DxDesigner software.
   e. Add the symbol to the **sym** directory of a DxDesigner project, or specify a new DxDesigner project with the new symbol.
   f. Instantiate the symbol in your DxDesigner schematic and export the design to the board layout tool.
   g. 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.
4. Create a board design with the DxDesigner software without the I/O Designer software by performing the following steps:
   a. Create a new DxBoardLink symbol with the **Symbol** wizard and reference the **.pin** from the Quartus II software in an existing DxDesigner project.
   b. Instantiate the symbol in your DxDesigner schematic and export the design to a board layout tool.

You can update these symbols with design changes with or without the I/O Designer software. If you use the Mentor Graphics I/O Designer software and you change symbols with the DxDesigner software, you must reimport the symbols into I/O Designer to avoid overwriting your symbol changes.

Performing Simultaneous Switching Noise (SSN) Analysis of Your FPGA

With the Quartus II software, you can extract pin assignment data and perform SSN analysis of your design for designs targeting the Stratix III device family. You can perform SSN analysis early in the board layout stage as part of your overall pin planning process; however, you do not have to perform SSN analysis to generate pin assignment data from the Quartus II software. You can use the SSN Analyzer tool in the Quartus II software to optimize the pin assignments for better SSN performance.

For more information about the SSN Analyzer, refer to the *Simultaneous Switching Noise (SSN) Analysis and Optimizations* chapter in volume 2 of the *Quartus II Handbook* and *About the SSN Analyzer* in Quartus II Help.

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 **.pin** and **.fx** files (refer to Figure 8–2). The **.pin** is an output file generated by the Quartus II Fitter that contains pin assignment information. You can use the Quartus II Pin Planner to set and change the assignments contained in the **.pin** and then transfer the assignments to the Mentor Graphics tools. You cannot, however, import pin assignment changes from the Mentor Graphics tools into the Quartus II software with the **.pin**.

The **.pin** lists all used and unused pins on your selected Altera device. It also provides the following basic information fields for each assigned pin on the device:

- Pin signal name and usage
- Pin number
- Signal direction
- I/O standard
- Voltage
- I/O bank
- User or Fitter-assigned

The **.fx** is 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. The **.fx** generated by the Quartus II software lists only assigned pins and 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

The **.fx** generated by the I/O Designer software lists all pins, including unused pins. In addition to the advanced information fields listed above, the **.fx** generated by the Mentor Graphics I/O Designer software also includes the following information fields:

- Device pin name
- Pin set
- Pin set position
- Pin set group
- Super pin set group
- Super pin set position

For more information about **.fx** files and the information fields added by the Mentor Graphics software, refer to *FPGA Xchange-Format File (.fx) Definition* in Quartus II Help and Mentor Graphics website (www.mentor.com) respectively.

The I/O Designer software can also read from or update a Quartus II Settings File (**.qsf**). The design flow uses the **.qsf** in a similar manner to the **.fx**, but does not transfer pin swap group information between the I/O Designer software and the Quartus II software.

Because the **.qsf** contains additional information about your project that the Mentor Graphics I/O Designer software does not use, Altera recommends using the **.fx** instead of the **.qsf**.

For more information about the **.qsf**, refer to *Quartus II Settings File (.qsf) Definition* in Quartus II Help.

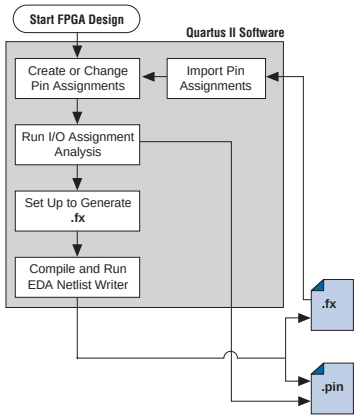
Generating a .pin File

The Quartus II software automatically generates the **.pin** after compiling your FPGA design or during I/O assignment analysis.

To start I/O assignment analysis, on the Processing menu, point to **Start** and then click **Start I/O Assignment Analysis**. The Quartus II Fitter generates the **.pin** and places the file in your Quartus II design directory with the name <*project name*>**.pin**. The Quartus II software cannot import assignments from an existing **.pin**.

Figure 8–2 shows how to generate **.pin** and **.fx** files.

Figure 8–2. Generating .pin and .fx Files *(Note 1)*



Note to Figure 8–2:

(1) For more information about the full design flow, which includes the I/O Designer software, the DxDesigner software, and the board layout tool flowchart details, refer to Figure 8–1.

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*.

Generating an .fx File

You can generate an **.fx** in the Quartus II software for symbol generation in the Mentor Graphics I/O Designer software.

For more information about generating an **.fx**, refer to *Generating FPGA Xchange-Format Files for Use with Other EDA Tools* in Quartus II Help.

Creating a Backup .qsf

To create a backup **.qsf** of your current pin assignments, follow these 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**.

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*.

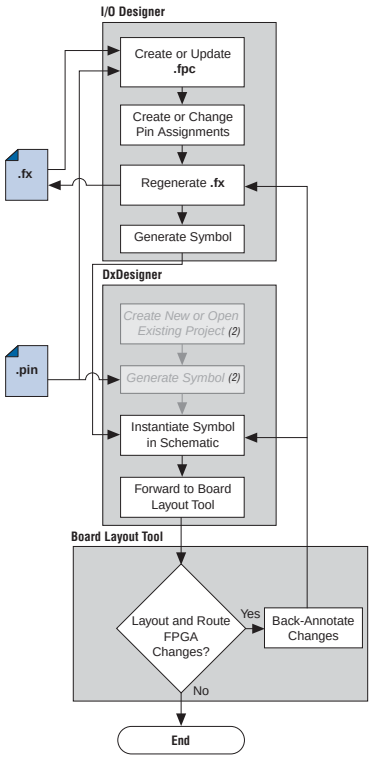
FPGA-to-Board Integration with the I/O Designer Software

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 with 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 an **.fx**, 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) For more information about the full design flow including the Quartus II software flowchart details, refer to Figure 8–1 on page 8–2.

(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 (www.mentor.com).

I/O Designer Database Wizard

An **.fpc** file stores all I/O Designer project information. You can create a new database incorporating information for the **.fx** and **.pin** files generated by the Quartus II software 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 containing only a selection of the target device. This action 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 placement and routing.

You can create an I/O Designer database with only a **.pin** or an **.fx**. However, if you are only using a **.pin**, you cannot import any I/O assignment changes made in the I/O Designer software back into the Quartus II software without first generating an **.fx**. If an **.fx** creates the I/O Designer database, the database may not contain all the available I/O assignment information. The **.fx** generated by the Quartus II software only lists pins with assigned signals. Because the **.pin** lists all device pins—whether signals are assigned to them or not—its use, along with the **.fx**, produces the most complete set of information for creating the I/O Designer database.

If you skip a step in the following process, you can complete the skipped step later. To return to a skipped step, on the Properties menu, click **File**. To create a new I/O Designer database using the **Database** wizard, follow these steps:

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 does not appear, you can access the wizard through the menu. To access the wizard, on the File menu, click **Database Wizard**.

2. Click **Next**. The **Define HDL source file** page appears.

   If no HDL files are available, or if the **.fx** contains your signal and pin assignments, you can skip Step 3 and proceed to Step 4.

   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.

3. If you have created a Verilog HDL or VHDL file in your Quartus II software design, you can add a top-level Verilog HDL or VHDL file in the I/O Designer software. Adding a file allows you to create functional blocks or get signal names from your design. You must create all physical pin assignments in I/O Designer if you are not using an **.fx** or a **.pin**. Click **Next**. The **Database Name** page appears.
4. In the **Database Name** page, type your database file name. Click **Next**. The Database Location window appears.
5. Add a path to the new or an existing database in the **Location** field, or browse to a database location. Click **Next**. The **FPGA flow** page appears.
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 appears.

   The Quartus II software version selections in the Tool/Library menu may not reflect the version of the Quartus II software currently installed in your system even if you are using the latest version of the I/O Designer software. The I/O Designer software uses the version number selection in this window to identify available or obsolete devices in that particular version of the Quartus II software. If you are unsure of the version to select, use the latest 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 (www.mentor.com).

9. In the **FPGAX file name** field, type or browse to the backup copy of the **.fx** generated by the Quartus II software.
10. In the **Pin report file name** field, type or browse to the **.pin** generated by the Quartus II software. Click **Next**.

    You can also select a **.qsf** for update. The I/O Designer software can update the pin assignment information in the **.qsf** without affecting any other information in the file.

    You can select a **.pin** without selecting an **.fx** for import. The I/O Designer software does not generate a **.pin**. To transfer assignment information to the Quartus II software, select an additional file and file type. Altera recommends selecting an **.fx** in addition to a **.pin** for transferring all the assignment information in the **.fx** and **.pin** files.

    In some versions of the I/O Designer software, the standard file picker may incorrectly look for a **.pin** instead of an **.fx**. In this case, select **All Files (*.*)** from the **Save as type** list and select the file from the list.

11. The **Synthesis** page appears. 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*.

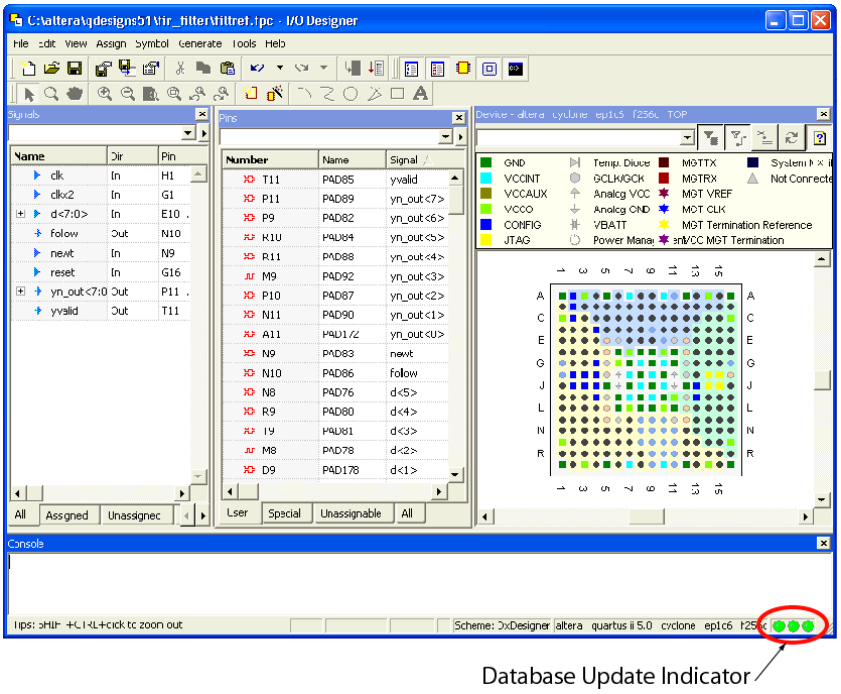
12. On the **PCB Flow** page, you can select an existing schematic project or create a new project as a symbol information destination.
    - To select an existing project, select **Choose existing project** and click **Browse** after the **Project Path** field. The **Select project** dialog box appears. Select the project.
    - To create a new project, in the **Select project** dialog box, select **Create new empty project**. Type the project file name in the **Name** field and browse to the location where you want to save the file. Click **OK**.

If you have not specified a design tool to which you can send symbol information in the I/O Designer software, click **Advanced** in the **PCB Flow** page and select your design tool. If you select the DxDesigner software, you have the option to specify a Hierarchical Occurrence Attributes (**.oat**) file to import into the I/O Designer software. Click **Next** and then click **Finish** to create the database.

In I/O Designer version 2005 or later, the **Update Wizard** dialog box (refer to Figure 8–7 on page 8–13) appears if you are creating the database with the **Database** wizard. Use the **Update Wizard** dialog box to confirm creation of the I/O Designer database using the selected **.fx** and **.pin** 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 8–4).

Figure 8–4. Mentor Graphics I/O Designer Main Window

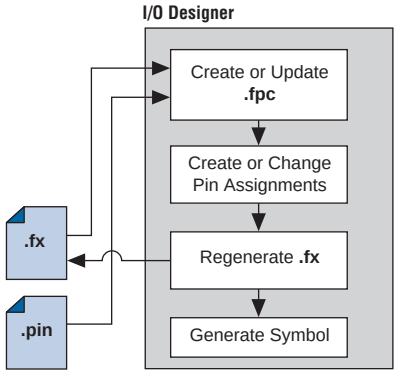


For more information about using the I/O Designer software and the DxDesigner software, refer to the Mentor Graphics website (www.mentor.com) or refer to the I/O Designer software or the DxDesigner Help.

Updating Pin Assignments from the Quartus II Software

As the design process continues, the FPGA designer must make changes to the logic design in the Quartus II software that places signals on different pins after recompiling the design, or manually with 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 connect to the correct pins on the FPGA. Updating the **.fx** and the **.pin** files in the Quartus II software facilitates this flow (Figure 8–5).

Figure 8–5. Updating the I/O Designer Pin Assignments in the Design Flow *(Note 1)*



Note to Figure 8–5:

(1) For more information about the full design flow, which includes the Quartus II software, the DxDesigner software, and the board layout tool flowchart details, refer to Figure 8–1 on page 8–2.

To update the **.fx** in your selected output directory and the **.pin** in your project directory after making changes to the design, perform one of the following tasks:

- compile, *or*
- start EDA Netlist Writer.

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, perform one of the following tasks:

- on the Processing menu, click **Start Compilation**, *or*
- on the Processing menu, click **Start I/O Assignment Analysis**.

For more information about setting up the **.fx** and running the I/O Assignment Analyzer, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

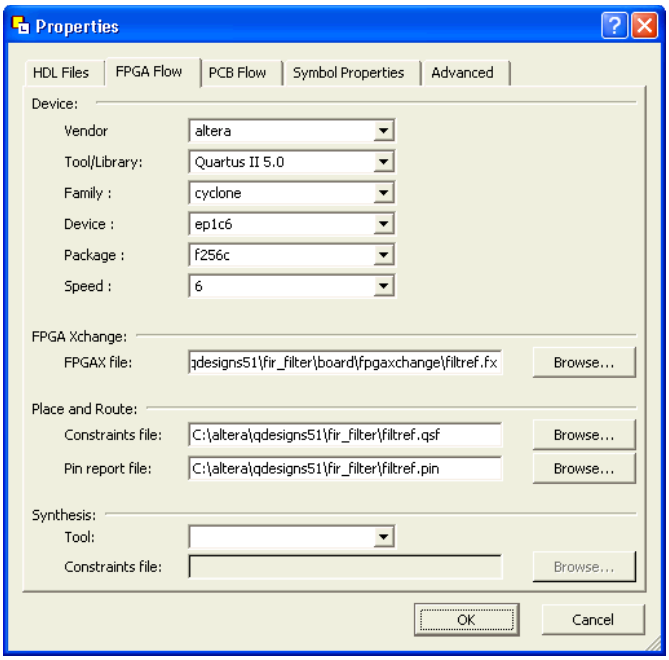
CAUTION

If your I/O Designer database points to the **.fx** 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 **.fx** 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 you update the **.fx** or the **.pin** in the Quartus II software, the I/O Designer database imports the changes. You must set up the locations for the files in the I/O Designer software.

1. To set up the file locations, on the File menu, click **Properties**. The project **Properties** dialog box appears (Figure 8–6).

Figure 8–6. Project Properties Dialog Box

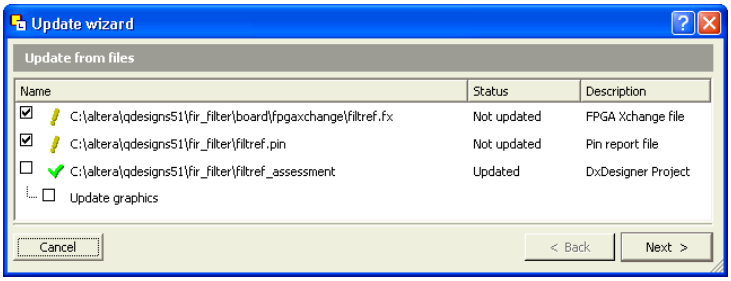


2. Under **FPGA Xchange**, click **Browse** to select the **.fx** file name and location.
3. To specify a Pin report file, under **Place and Route**, click **Browse** to select the **.pin** file name and location.

After you have set up these file locations, the I/O Designer software monitors these files for changes. If the **.fx** or **.pin** changes during the design flow, three indicators flash red in the lower right corner of the I/O Designer GUI (refer to Figure 8–4 on page 8–10). You can continue working or click on the indicators to open the I/O Designer **Update Wizard** dialog box. If you have made changes to your design in the Quartus II software that result in an updated **.fx** or **.pin** and the update indicators do not flash or you have previously canceled an indicated update, manually open the **Update Wizard** dialog box. To open the **Update Wizard** dialog box, on the File menu, click **Update**.

The I/O Designer **Update Wizard** dialog box lists the updated files associated with the database (Figure 8–7).

Figure 8–7. Update Wizard Dialog Box



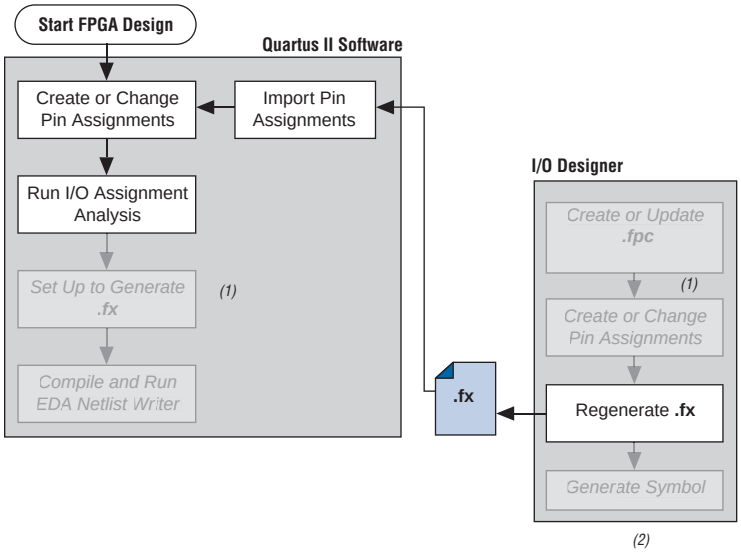
The paths to the updated files have yellow exclamation points and the **Status** column shows **Not updated**, indicating that the database has not yet been updated with the newer information contained in the files. A checkmark to the left of any updated file indicates that the file updates the database. 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 the **.fx** and the **.pin** files simultaneously. Turning on the **.fx** and the **.pin** files for update causes the **Update Wizard** dialog box 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 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 pin-out. The I/O Designer software accommodates this reverse flow as shown in Figure 8–8.

Figure 8–8. Updating the Quartus II Pin Assignments in the Reverse Design Flow



Notes to Figure 8–8:

(1) These are software-specific steps in the design flow and are not necessary for the reverse flow steps of the design.

(2) For more information about the full design flow, which includes the complete I/O Designer software, the DxDesigner software, and the board layout tool flowchart details, refer to Figure 8–1 on page 8–2.

You can make pin assignment changes directly in the I/O Designer software, or the software can automatically update 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 **.fx** 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**.

CAUTION

If your I/O Designer database points to the **.fx** generated by the Quartus II software instead of a backup copy, updating the file from the I/O Designer software overwrites any changes 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, create 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 8–11.

You must import the updated **.fx** into the Quartus II software. To import the file, follow these 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 **.fx** and click **Open**.
5. Click **OK**.

Protecting Assignments in the Quartus II Software

To protect assignments in the Quartus II software, follow these 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 **.fx**. This action automatically creates a backup copy of the Quartus II constraints file that contains all 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 must understand how the PCB works, and to generate a netlist for a board layout tool for board design and routing. The I/O Designer software allows you 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 various functional blocks, allowing multiple part fractures on the same schematic page or across multiple pages. In the DxDesigner software, these part fractures join together with the use of the `HETERO` attribute.

The I/O Designer software can generate symbols for use in various 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 with the **.fx**. This section discusses symbol creation specifically for the DxDesigner software.

You can create schematic symbols with the I/O Designer software in the following ways:

- Manually
- Using the I/O Designer **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 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, start 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 both. Signals imported into the database, usually from Verilog HDL or VHDL files, are the basis of a functional symbol. 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 more information about manually creating, importing, 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.

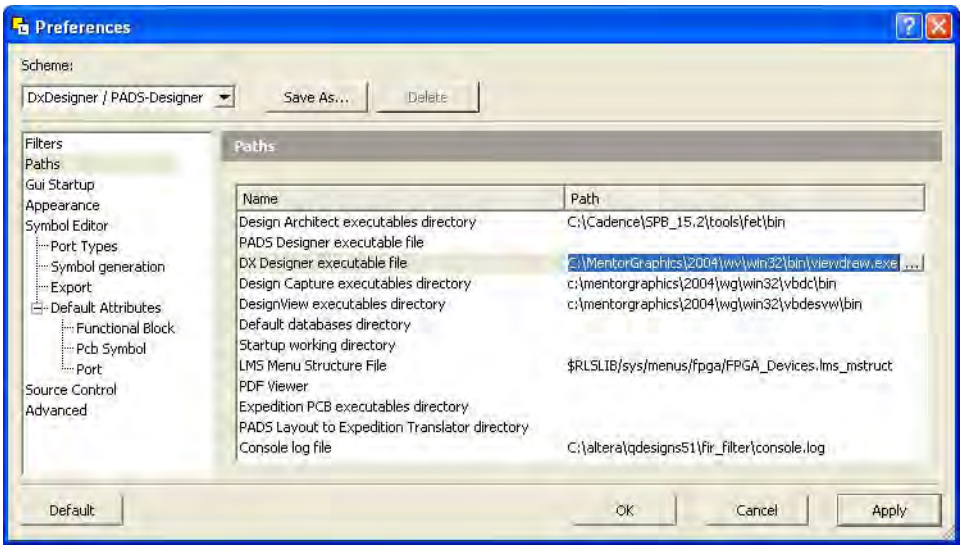
Setting Up the I/O Designer Software to Work with the DxDesigner Software

To verify if you are set up to export symbols to a DxDesigner project, 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, follow these steps:

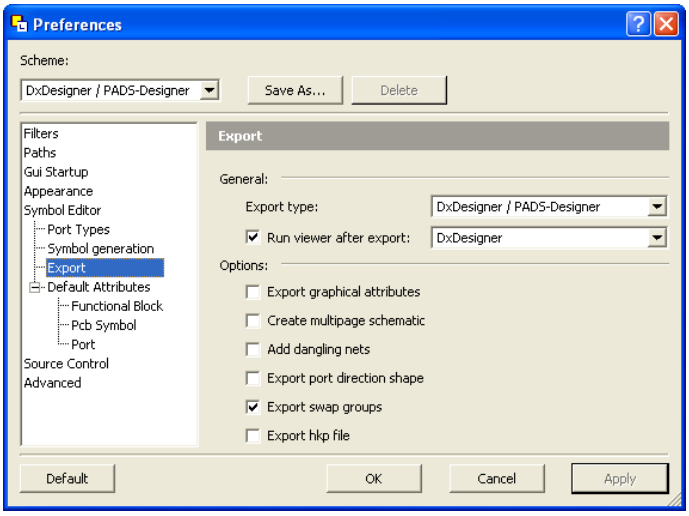
1. Start the I/O Designer software.
2. On the Tools menu, click **Preferences**. The **Preferences** dialog box appears.
3. Click **Paths**, double-click on the **DxDesigner executable file** path field, and click **Browse** to select the location of the DxDesigner application (Figure 8–9).
4. Click **Apply**.

Figure 8–9. Path Preferences Dialog Box



5. Click **Symbol Editor** and click **Export**. In the Export type menu, under **General**, select **DxDesigner/PADS-Designer** (Figure 8–10).
6. Click **Apply** and click **OK**.

Figure 8–10. Symbol Editor Export Preferences



7. On the File menu, click **Properties**. The **Properties** dialog box appears.
8. Click the **PCB Flow** tab and click **Path to a DxDesigner project directory**.
9. Click **OK**.

If you do not have a new DxDesigner project in the **Database** wizard and a DxDesigner project, you must create a new database with the DxDesigner software, and point the I/O Designer software to this new project.

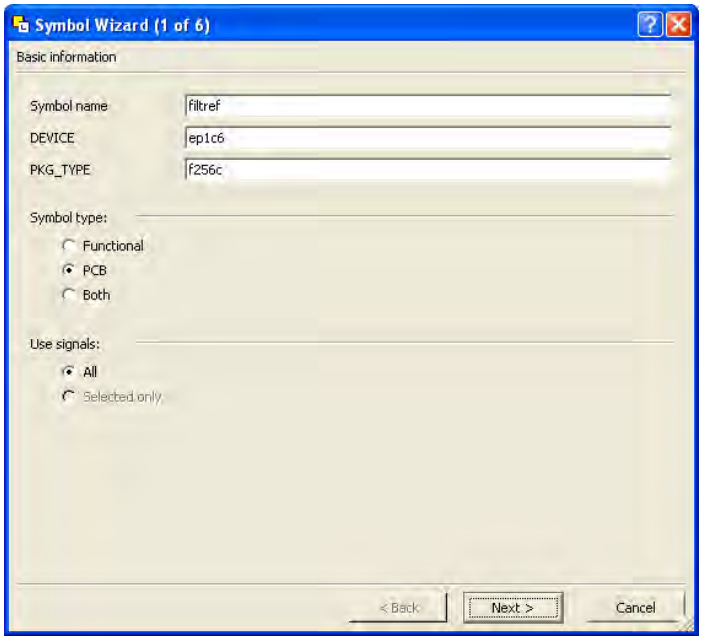
For more information about creating and working with DxDesigner projects, refer to the DxDesigner Help.

Creating Symbols with the Symbol Wizard

You can create, fracture, and edit FPGA symbols based on Altera devices with the I/O Designer **Symbol** wizard. To create a symbol based on a selected Altera FPGA device, follow these steps:

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 appears (Figure 8–11).

Figure 8–11. Symbol Wizard



3. On page 1 of the **Symbol Wizard** page, in the **Symbol name** field, type the symbol name. The **DEVICE** and **PKG_TYPE** fields are automatically populated with the device and package information. Under **Symbol type**, click **PCB**. Under **Use signals**, click **All**.
4. Click **Next**. The **Symbol Wizard (2 of 6)** page appears.

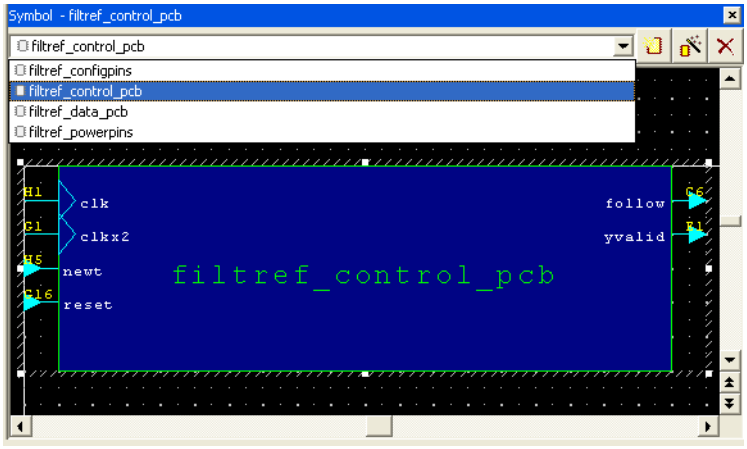
   If the **DEVICE** and **PKG_TYPE** fields are blank or incorrect, cancel the **Symbol** wizard and select the correct device information. On the File menu, click **Properties**. In the Properties window, click the **FPGA Flow** tab and enter the correct device information.

5. On page 2 of the **Symbol Wizard** page, 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** to preserve your current fractures. Select other options on this page as appropriate for your symbol.
6. Click **Next**. The **Symbol Wizard (3 of 6)** page appears.
7. Additional fracturing options are available on page 3 of the **Symbol Wizard** page. After selecting the necessary options, click **Next**. The **Symbol Wizard (4 of 6)** page appears.
8. On page 4 of the **Symbol Wizard** page, select the options for the appearance of the symbols. Select the necessary options and click **Next**. The **Symbol Wizard (5 of 6)** page appears.
9. On page 5 of the **Symbol Wizard** page, define what information you want to label for the entire symbol and for individual pins. Select the necessary options and click **Next**. The **Symbol Wizard (6 of 6)** page appears.

10. On the final page of the **Symbol Wizard** page, add additional signals and pins that have not been placed in the symbol. Click **Finish** when you complete your selections.

You can view your symbol and any fractures you created with the Symbol Editor (Figure 8–12). You can edit parts of the symbol, delete fractures, or rerun the **Symbol** wizard.

Figure 8–12. 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 in the I/O Designer software, with an updated **.fx** from the Quartus II software, or from a back-annotated change in your board layout tool.

Exporting 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**. The **/sym directory** in your DxDesigner project saves each symbol in the database as a separate file. 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 translate into Tcl commands run by the tool. You can view the generated Tcl commands and run scripts, or type individual commands in the I/O Designer Console window.

This scripting support section includes commands that perform some of the operations described in this chapter.

If you want to change the **.fx** 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>
```

You can type the following command to update the I/O Designer database with assignment updates made in the Quartus II software after specifying the **.fx**:

```
update_from_fpga_xchange_file
```

You can type the following command to update the **.fx** with changes made to the assignments in the I/O Designer software for transfer back into the Quartus II software:

```
generate_fpga_xchange_file
```

You can type the following command if you want to import assignment data from a **.pin** created by the Quartus II software:

```
set_pin_report_file -quartus_pin <file name>
```

You can run the I/O Designer **Symbol** wizard with the following command:

```
symbolwizard
```

You can set the DxDesigner project directory path where symbols are saved with the following command:

```
set_dx_designer_project -path <path>
```

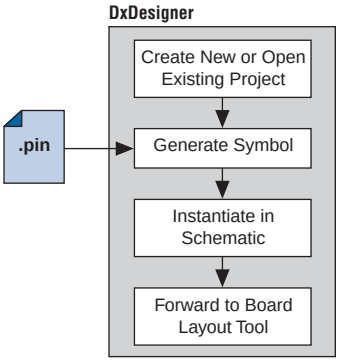
For more information about Tcl scripting and Tcl scripting with the Quartus II software, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*. For more information about the Tcl scripting capabilities of the I/O Designer software as well as a list of available commands, refer 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 the PCB design types. 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** generated by the Quartus II software.

You can only make signal and pin assignment changes in the Quartus II software and these changes reflect as 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 8–13 shows the design flow without the I/O Designer software.

Figure 8–13. Design Flow Without the I/O Designer Software *(Note 1)*



Note to Figure 8–13:

(1) For more information about the full design flow, which includes the Quartus II software, the I/O Designer software, and the board layout tool flowchart details, refer to Figure 8–1 on page 8–2.

For more information about the DxDesigner software, including usage, support, training, and product updates, refer to the Mentor Graphics website (www.mentor.com), or choose Schematic Design Help Topics in the DxDesigner Help.

DxDesigner Project Settings

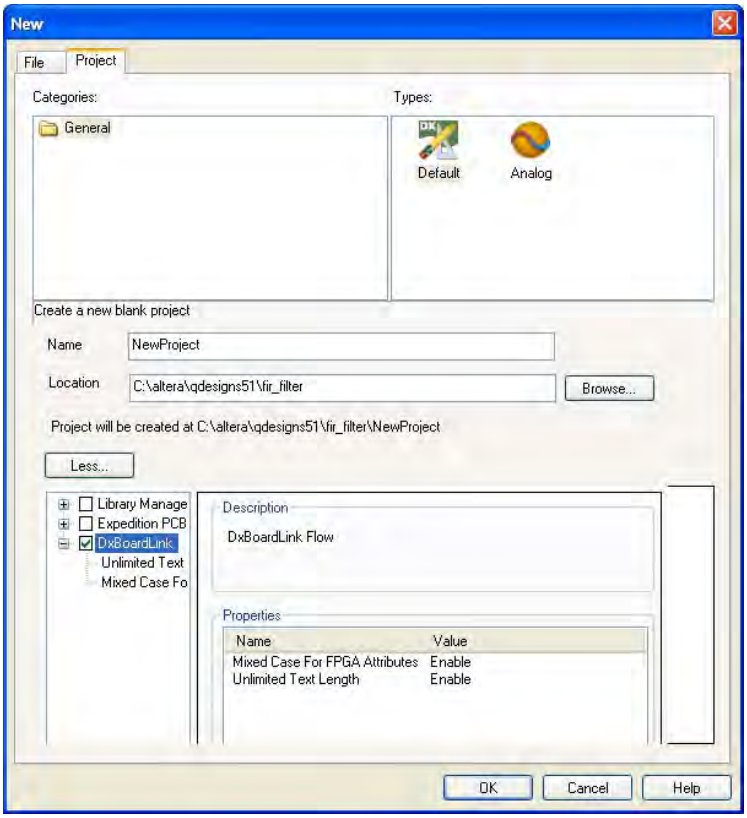
New projects in the DxDesigner software are set up to create FPGA symbols by default. However, if you are using the I/O Designer software with the DxDesigner software, you must enable the DxBoardLink Flow options for complete support and compatibility with the I/O Designer software.

You can enable the DxBoardLink flow design configuration during or after creating a new DxDesigner project.

To enable the DxBoardLink flow design configuration when creating a new DxDesigner project, follow these steps:

1. Start the DxDesigner software.
2. On the File menu, click **New** and click the **Project** tab. The **New** dialog box appears (Figure 8–14).

Figure 8–14. New Project Dialog Box



3. Click **More**. Turn on **DxBoardLink** (Figure 8–14).

☞ To enable the DxBoardLink Flow design configuration in an existing project, click **Design Configurations** in the Design Configuration toolbar and turn on **DxBoardLink** (Figure 8–15).

Figure 8–15. DxBoardLink Design Configuration

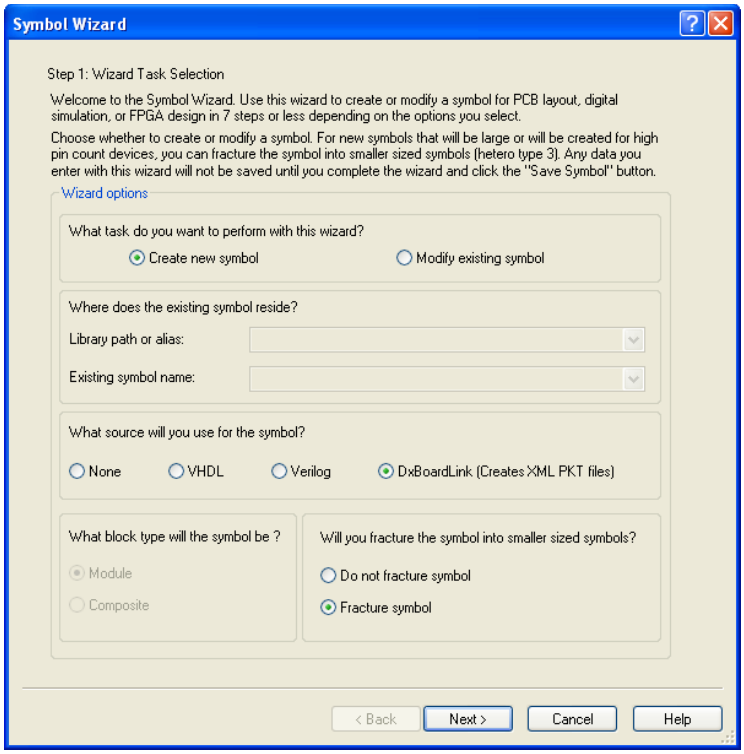
DxDesigner Symbol Wizard

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 with the DxDesigner **Symbol** wizard. To start the **Symbol** wizard, follow these steps:

1. Start the DxDesigner software.
2. Click **Symbol Wizard** in the toolbar, or on the File menu, click **New**. The **New** window appears. Click the **File** tab and create a new file of type **Symbol Wizard**.
3. Type the new symbol name in the name field and click **OK**. The **Symbol Wizard** page appears (Figure 8–16).

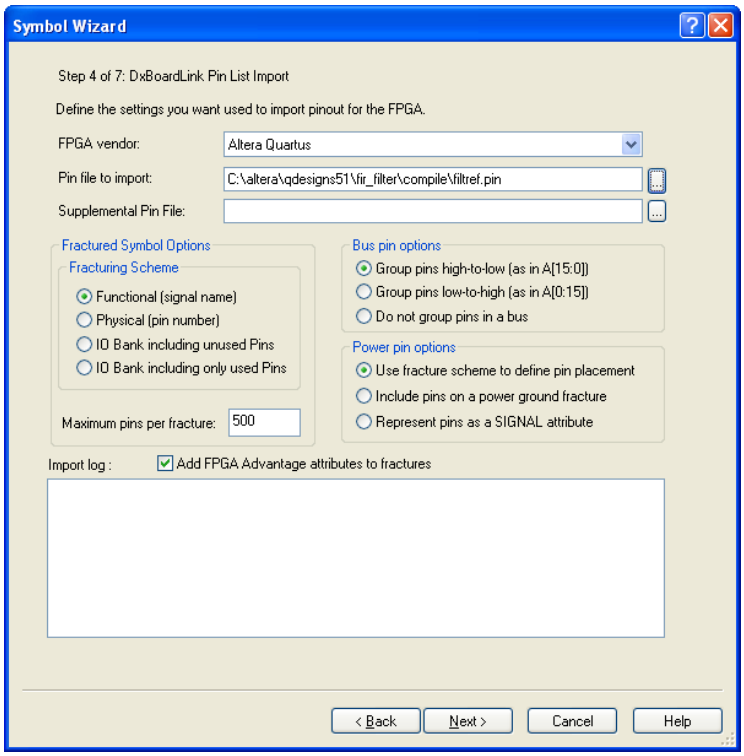
Figure 8–16. Wizard Task Selection



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. Choose whether or not to fracture the symbol. After making your selections, click **Next**. The **New Symbol and Library Name** page appears.
5. On the **New Symbol and Library Name** page, type a name for the symbol, an overall part name for all 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 appears.

6. On the **Symbol Parameters** page, specify the appearance of the generated symbol and how it matches up with the grid you have set in your DxDesigner project schematic. After making your selections, click **Next**. The **DxBoardLink Pin List Import** page appears (Figure 8–17).

Figure 8–17. 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** from your Quartus II design project directory. You can also select choices from the Fracturing Scheme, Bus pin, and Power pin options. After making your selections, click **Next**. The **Symbol Attributes** page appears.
8. On the **Symbol Attributes** page, select to create or modify symbol attributes for use in the DxDesigner software. After making your selections, click **Next**. The **Pin Settings** page appears.
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 making your selections, click **Save Symbol**.

After creating the symbol, you can examine and place any fracture of the symbol in your schematic. You can locate separate files of all the fractures you created 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 manually edit the symbols or with the **Symbol** wizard.

Symbols created in the DxDesigner software can be edited and updated with newer versions of the **.pin** generated by the Quartus II software. However, you cannot fracture a symbol again because symbol fracturing is permanent. To create new fractures for your design, create a new symbol in the **Symbol** wizard, and perform the steps in "DxDesigner Symbol Wizard" on page 8–23.

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 using the Quartus II software with the Mentor Graphics toolset assists the FPGA designer and the board designer in preventing errors and focusing their attention on the design.

Document Revision History

Table 8–1 shows the revision history for this chapter.

Table 8–1. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Removed Reference Document section. ■ General style editing. ■ Added a link to Help in "Performing Simultaneous Switching Noise (SSN) Analysis of Your FPGA" on page 8–3. ■ Removed Figure 8–4 on page 8–9 and Figure 8–5 on page 8–11. ■ Updated "Generating an .fx File" on page 8–6.
November 2009	9.1.0	■ Added minor information about simultaneous switching noise (SSN) analysis on "Performing Simultaneous Switching Noise (SSN) Analysis of Your FPGA" on page 8–3. ■ General style editing.
March 2009	9.0.0	■ 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.
November 2008	8.1.0	Changed to 8½" × 11" page size. No change to content.
May 2008	8.0.0	Updated references.

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

Take an online survey to provide feedback about this handbook chapter.

9. Cadence PCB Design Tools Support

QII52014-10.0.0

This chapter addresses how the Quartus® II software interacts with the Cadence Allegro Design Entry HDL software and the Cadence Allegro Design Entry CIS (Component Information System) software (also known as OrCAD Capture CIS) to provide a complete FPGA-to-board integration design workflow.

With today's large, high-pin-count and high-speed FPGA devices, good PCB design practices are important to ensure the correct operation of your system. The PCB design takes place concurrently with the design and programming of the FPGA. An FPGA or ASIC designer initially creates the signal and pin assignments and the board designer must transfer these assignments to the symbols used in their system circuit schematics and board layout correctly. As the board design progresses, you must perform pin reassignments to optimize the layout. You must communicate pin reassignments to the FPGA designer to ensure the new assignments are processed through the FPGA with updated placement and routing.

This chapter is intended for board design and layout engineers who want to begin the FPGA board integration process while the FPGA is still in the design phase. Part librarians can also benefit from this chapter by learning the method to use output from the Quartus II software to create new library parts and symbols.

This chapter discusses 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 making signal and pin assignment changes 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 making signal and pin assignment changes in the Quartus II software.
- Using Altera-provided device libraries in the Cadence Allegro Design Entry CIS software.

The procedures in this chapter require the following software:

- The Quartus II software version 5.1 or later
- The Cadence Allegro Design Entry HDL software or the Cadence Allegro Design Entry CIS software version 15.2 or later
- The OrCAD Capture software with the optional CIS option version 10.3 or later (optional)

These programs are very similar because the Cadence Allegro Design Entry CIS software is based on the OrCAD Capture software. This chapter refers to the Cadence Allegro Design Entry CIS software; however, any procedural information can also apply to the OrCAD Capture software unless otherwise noted.

For more information about obtaining and licensing the Cadence tools and for product information, support, and training, refer to the Cadence website (www.cadence.com). For more information about the OrCAD Capture software and the CIS option, refer to the Cadence website (www.cadence.com). For more information about Cadence and OrCAD support and training, refer to the EMA Design Automation website (www.ema-eda.com).

Product Comparison

The Cadence and OrCAD design tools are different in their function and location of product information. Table 9–1 lists the Cadence and OrCAD products described in this chapter and provides information about changes, product information, and support.

Table 9–1. Cadence and OrCAD Product Comparison

Description	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, the Cadence Allegro Design Entry CIS software 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 provides support and training for all OrCAD products.
Vendor Design Flow	Cadence Allegro 600 series, formerly known as the Expert Series, for high-end, high-speed design.	Cadence Allegro 200 series, formerly known as the 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

You can create a design flow integrating an Altera FPGA design from the Quartus II software through a circuit schematic in the Cadence Allegro Design Entry HDL software or the Cadence Allegro Design Entry CIS software. Figure 9–1 shows the design flow with the Cadence Allegro Design Entry HDL software. Figure 9–2 shows the design flow with the Cadence Allegro Design Entry CIS software.

Figure 9–1. Design Flow with the Cadence Allegro Design Entry HDL Software

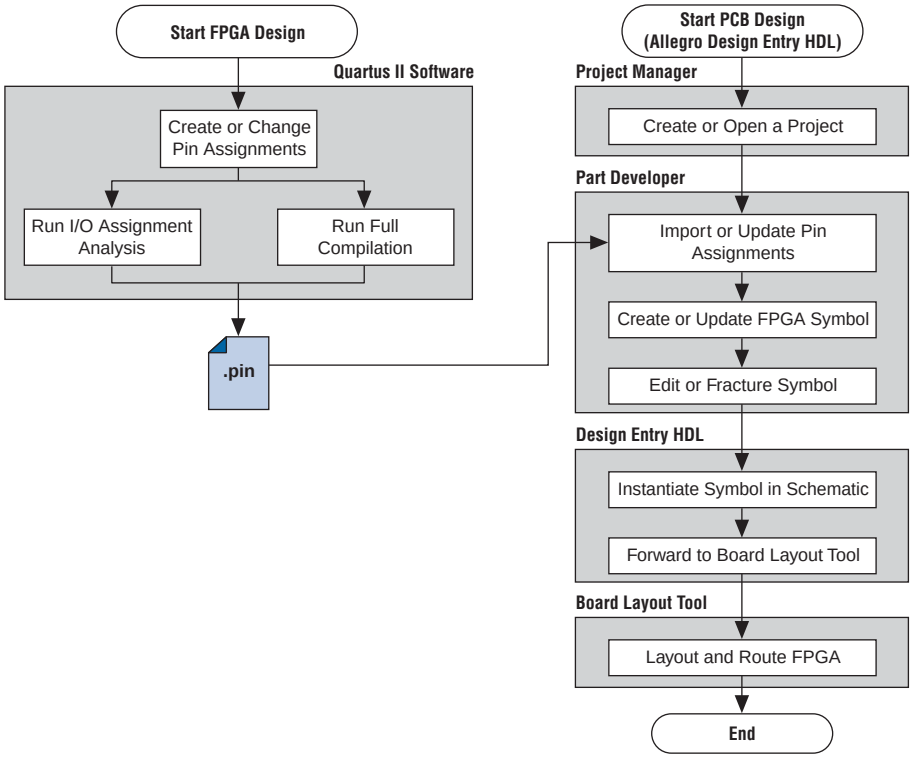


Figure 9–2. Design Flow with the Cadence Allegro Design Entry CIS Software

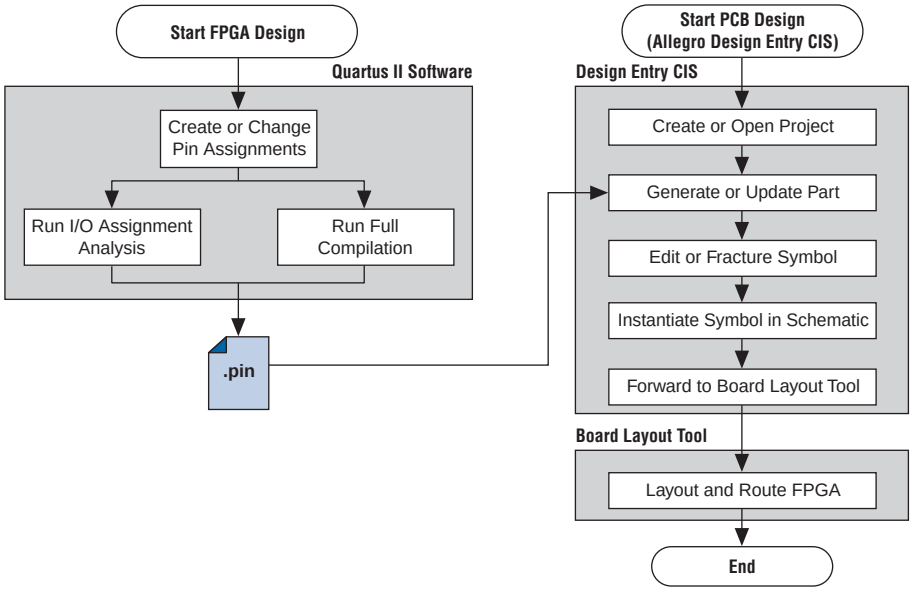


Figure 9–1 and Figure 9–2 show the possible design flows, depending on your tool choice. To create FPGA symbols using the Cadence Allegro PCB Librarian Part Developer tool, you must obtain the Cadence PCB Librarian Expert license. You can update symbols with changes made to the FPGA design using any of these tools.

To integrate an Altera FPGA design starting in the Quartus II software through to a circuit schematic in the Cadence Allegro Design Entry HDL software or the Cadence Allegro Design Entry CIS software, follow these steps:

1. In the Quartus II software, compile your design to generate a Pin-Out File (**.pin**) to transfer the assignments to the Cadence software.
2. If you are using the Cadence Allegro Design Entry HDL software for your schematic design, follow these steps:
   a. Open an existing project or create a new project in the Cadence Allegro Project Manager tool.
   b. Construct a new symbol or update an existing symbol using the Cadence Allegro PCB Librarian Part Developer tool.
   c. With the Cadence Allegro PCB Librarian Part Developer tool, edit your symbol or fracture it into smaller parts (optional).
   d. Instantiate the symbol in your Cadence Allegro Design Entry HDL software schematic and transfer the design to your board layout tool.

   –or

If you are using the Cadence Allegro Design Entry CIS software for your schematic design, follow these steps:

a. Generate a new part in a new or existing Cadence Allegro Design Entry CIS project, referencing the **.pin** output file from the Quartus II software. You can also update an existing symbol with a new **.pin**.

b. Split the symbol into smaller parts as necessary.

c. Instantiate the symbol in your Cadence Allegro Design Entry CIS schematic and transfer the design to your board layout tool.

Performing Simultaneous Switching Noise (SSN) Analysis of Your FPGA

With the Quartus II software, you can extract pin assignment data and perform SSN analysis of your FPGA design for designs targeting the Stratix III device family. You can analyze SSN in your device early in the board layout stage as part of your overall pin planning process; however, you do not have to perform SSN analysis to generate pin assignment data from the Quartus II software. You can use the SSN Analyzer tool to optimize the pin assignments for better SSN performance of your device.

For more information about the SSN Analyzer, refer to *About the SSN Analyzer* in Quartus II Help and the *Simultaneous Switching Noise (SSN) Analysis and Optimizations* chapter in volume 2 of the *Quartus II Handbook*.

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**. The **.pin** is an output file generated by the Quartus II Fitter containing pin assignment information. You can use the Quartus II Pin Planner to set and change the assignments in the **.pin** and then transfer the assignments to the Cadence design tools. You cannot, however, import pin assignment changes from the Cadence design tools into the Quartus II software with the **.pin**.

The **.pin** lists all used and unused pins on your selected Altera device. The **.pin** also provides the following basic information fields for each assigned pin on the device:

- Pin signal name and usage
- Pin number
- Signal direction
- I/O standard
- Voltage
- I/O bank
- User or Fitter-assigned

For more 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 a .pin File

The Quartus II Fitter generates a **.pin** during a full compilation of your FPGA design, or when performing I/O assignment analysis on your design. You can locate the **.pin** in your Quartus II project directory with the name <*project name*>**.pin**.

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 a schematic capture tool and is part of the Cadence 600 series design flow. Use the Cadence Allegro Design Entry HDL 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** generated by the Quartus II software. You can only make signal and pin assignment changes in the Quartus II software and these changes reflect as updated symbols in a Cadence Allegro Design Entry HDL project. For more information about the design flow with the Cadence Allegro Design Entry HDL software, refer to Figure 9–1 on page 9–3.

Routing or pin assignment changes made in a board layout tool or a Cadence Allegro Design Entry HDL software symbol cannot be back-annotated to the Quartus II software.

For more information about the Cadence Allegro Design Entry HDL software and the Cadence Allegro PCB Librarian Part Developer tool, including licensing, support, usage, training, and product updates, refer to the Help in the software or to the Cadence website (www.cadence.com).

Creating Symbols

In addition to circuit simulation, circuit board schematic creation is one of the first tasks required when designing a new PCB. Schematics must understand how the PCB works, and to generate a netlist for a board layout tool for board design and routing. The Cadence Allegro PCB Librarian Part Developer tool allows you to create schematic symbols based on FPGA designs exported from the Quartus II software.

You can create symbols for the Cadence Allegro Design Entry HDL project with the Cadence Allegro PCB Librarian Part Developer tool, which is available in the Cadence Allegro Project Manager tool. Altera recommends using the Cadence Allegro PCB Librarian Part Developer tool to import FPGA designs into the Cadence Allegro Design Entry HDL software.

You must obtain a PCB Librarian Expert license from Cadence to run the Cadence Allegro PCB Librarian Part Developer tool. The Cadence Allegro PCB Librarian Part Developer tool provides a GUI with many options for creating, editing, fracturing, and updating symbols. If you do not use the Cadence Allegro PCB Librarian Part Developer tool, 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 automatically create FPGA symbols using the programmable IC (PIC) design flow found in the Cadence Allegro Project Manager tool. For more information about using the PIC design flow, refer to the Help in the Cadence design tools, or go to the Cadence website (www.cadence.com).

Before creating a symbol from an FPGA design, you must open a Cadence Allegro Design Entry HDL project with the Cadence Allegro Project Manager tool. If you do not have an existing Cadence Allegro Design Entry HDL project, you can create one with the Cadence Allegro Design Entry HDL software. The Cadence Allegro Design Entry HDL project directory with the name <*project name*>**.cpm** contains your Cadence Allegro Design Entry HDL projects.

While the Cadence Allegro PCB Librarian Part Developer tool 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.

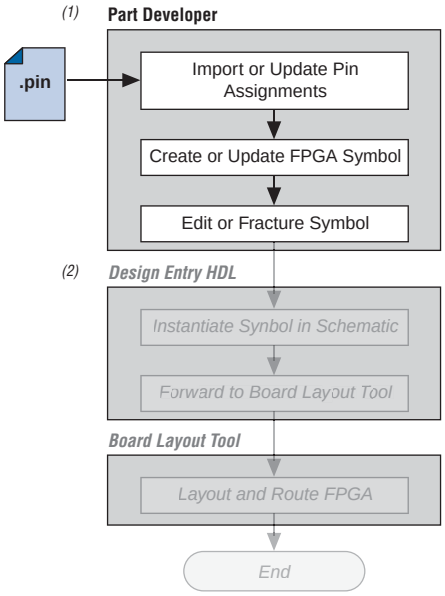
Table 9–2. Symbol Fracture Naming

	Cadence Allegro PCB Librarian Part Developer Tool	Cadence Allegro Design Entry HDL Software	Cadence Allegro Design Entry CIS Software
During symbol generation	Slots	—	Sections
During symbol schematic instantiation	—	Versions	Parts

Cadence Allegro PCB Librarian Part Developer Tool

You can create, fracture, and edit schematic symbols for your designs using the Cadence Allegro PCB Librarian Part Developer tool. Symbols designed in the Cadence Allegro PCB Librarian Part Developer tool can be split or fractured into several functional blocks called slots, allowing multiple smaller part fractures to exist on the same schematic page or across multiple pages. Figure 9–3 shows how the Cadence Allegro PCB Librarian Part Developer tool fits into the design flow.

Figure 9–3. Cadence Allegro PCB Librarian Part Developer Tool in the Design Flow



Notes to Figure 9–3:

(1) For more information about the full design flow flowchart, refer to Figure 9–1 on page 9–3.

(2) Grayed out steps are not part of the FPGA symbol creation or update process.

To run the Cadence Allegro PCB Librarian Part Developer tool, you must open a Cadence Allegro Design Entry HDL project in the Cadence Allegro Project Manager tool. To open the Cadence Allegro PCB Librarian Part Developer tool, on the Flows menu, click **Library Management**, and then click **Part Developer**.

Import and Export Wizard

After starting the Cadence Allegro PCB Librarian Part Developer tool, use the **Import and Export** wizard to import your pin assignments from the Quartus II software.

Altera recommends using your PCB Librarian Expert license file. To point to your PCB Librarian Expert license file, on the File menu, click **Change Product** and then select the correct product license.

To access the Import and Export wizard, follow these steps:

1. On the File menu, click **Import and Export**.
2. Select **Import ECO-FPGA**, and then click **Next**.

3. In the **Select Source** page of the **Import and Export** wizard, specify the following settings:
   a. In the **Vendor** list, select **Altera**.
   b. In the **PnR Tool** list, select **quartusII**.
   c. In the **PR File** box, browse to select the **.pin** in your Quartus II project directory.
   d. Click **Simulation Options** to select simulation input files.
   e. Click **Next.**
4. In the **Select Destination** dialog box, specify the following settings:
   a. Under **Select Component**, click **Generate Custom Component** to create a new component in a library,

   –or

   b. Click **Use standard component** to base your symbol on an existing component.

   ☞ Altera recommends creating a new component if you previously created a generic component for an FPGA device. Generic components can cause some problems with your design. When you create a new component, you can place your pin and signal assignments from the Quartus II software on this component and reuse the component as a base when you have a new FPGA design.

   c. In the **Library** list, select an existing library. You can select from the cells in the selected library. Each cell represents all the symbol versions and part fractures for a particular part. In the **Cell** list, select the existing cell to use as a base for your part.
   d. In the **Destination Library** list, select a destination library for the component. Click **Next**.
   e. Review and edit the assignments you import into the Cadence Allegro PCB Librarian Part Developer tool based on the data in the **.pin** and then click **Finish**. The location of each pin is not included in the **Preview of Import Data** page of the **Import and Export** wizard, but input pins are on the left side of the created symbol, output pins on the right, power pins on the top, and ground pins on the bottom.

Editing and Fracturing Symbol

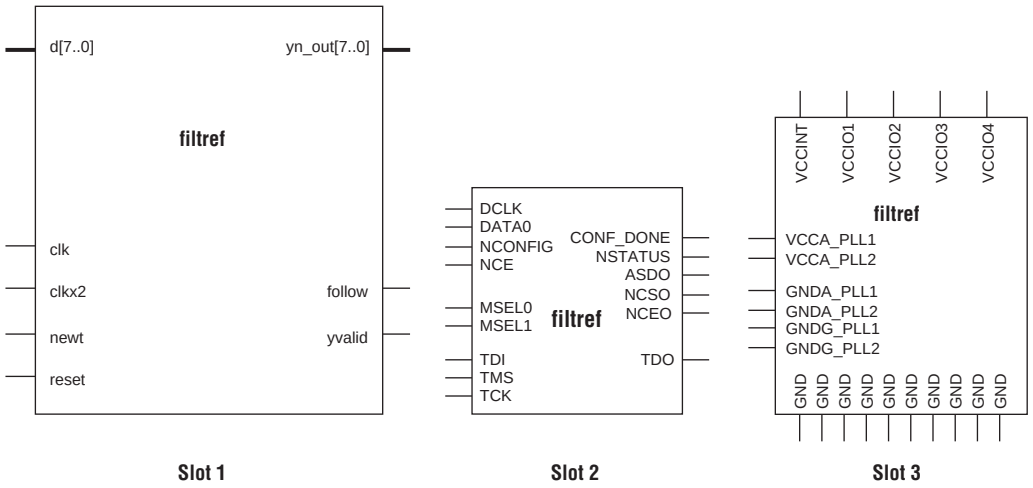
After creating your new symbol in the Cadence Allegro PCB Librarian Part Developer tool, you can edit the symbol graphics, fracture the symbol into multiple slots, and add or change package or symbol properties.

The Part Developer Symbol Editor contains many graphical tools to edit the graphics of a particular symbol. To edit the symbol graphics, select the symbol in the cell hierarchy. The **Symbol Pins** tab appears. You can edit the preview graphic of the symbol in the **Symbol Pins** tab.

Fracturing a Cadence Allegro PCB Librarian Part Developer package into separate symbol slots is 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 can 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 9–4 shows a part fractured into separate slots.

Figure 9–4. Splitting a Symbol into Multiple Slots *(Notes 1)*, *(2)*



Notes to Figure 9–4:

(1) Figure 9–4 represents a Cyclone device with JTAG or passive serial (PS) mode configuration option settings. Symbols created for other devices or other configuration modes may have different sets of configuration pins, but can be fractured in a similar manner.

(2) The power/ground slot shows only a representation of power and ground pins because the device contains a large number of power and ground pins.

To fracture a part into separate slots, or to modify the slot locations of pins on parts fractured in the Cadence Allegro PCB Librarian Part Developer tool, follow these steps:

1. Start the Cadence Allegro Design Project Manager.
2. On the Flows menu, click **Library Management**.
3. Click **Part Developer**.
4. Click the name of the package you want to change in the cell hierarchy.
5. Click **Functions/Slots**. If you are not creating new slots but want to change the slot location of some pins, proceed to Step 6. 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**.
6. Click **Distribute Pins**. Specify the slot location for each pin. Use the checkboxes in each column to move pins from one slot to another. Click **OK**.
7. After distributing the pins, click the **Package Pin** tab and click **Generate Symbol(s)**.

8. Select whether to create a new symbol or modify an existing symbol in each slot. Click **OK**.

The newly generated or modified slot symbols appear as separate symbols in the cell hierarchy. Each of these symbols can be edited individually.

CAUTION

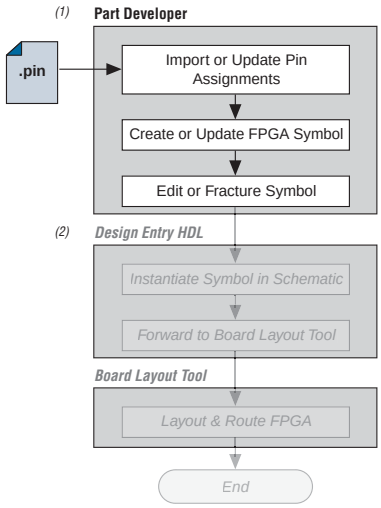
The Cadence Allegro PCB Librarian Part Developer tool allows you to remap pin assignments in the **Package Pin** tab of the main Cadence Allegro PCB Librarian Part Developer window. If signals remap to different pins in the Cadence Allegro PCB Librarian Part Developer tool, the changes reflect only in regenerated symbols for use in your schematics. You cannot transfer pin assignment changes to the Quartus II software from the Cadence Allegro PCB Librarian Part Developer tool, 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 Cadence Allegro PCB Librarian Part Developer tool, and update the symbol as described in the following sections.

For more information about creating, editing, and organizing component symbols with the Cadence Allegro PCB Librarian Part Developer tool, refer to the Part Developer Help.

Updating FPGA Symbols

As the design process continues, you must make logic changes in the Quartus II software, placing signals on different pins after recompiling the design, 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. To ensure signals connect to the correct pins on the FPGA, you must carry forward these types of changes to the circuit schematic and board layout tools. Updating the **.pin** in the Quartus II software facilitates this flow. Figure 9–5 shows this part of the design flow.

Figure 9–5. Updating the FPGA Symbol in the Design Flow



Notes to Figure 9–5:

(1) For more information about the full design flow flowchart, refer to Figure 9–1 on page 9–3.

(2) Grayed out steps are not part of the FPGA symbol update process.

To update the symbol using the Cadence Allegro PCB Librarian Part Developer tool after updating the **.pin**, follow these steps:

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** 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**. Selecting **Generate Custom Component** replaces the cell listed under the **Specify Library and Cell** name header with a new, non fractured cell. 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 appears summarizing the changes made to the cell. To accept the changes and update the cell, click **Finish**.
6. The main Cadence Allegro PCB Librarian Part Developer window appears. You can edit, fracture, and generate the updated symbols as usual from the main Cadence Allegro PCB Librarian Part Developer window.

If the Cadence Allegro PCB Librarian Part Developer tool is not set up to point to your PCB Librarian Expert license file, an error message appears 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.

Instantiating the Symbol in the Cadence Allegro Design Entry HDL Software

To instantiate the symbol in your Cadence Allegro Design Entry HDL schematic after saving the new symbol in the Cadence Allegro PCB Librarian Part Developer tool, follow these steps:

1. In the Cadence Allegro Project Manager tool, switch to the board design flow.
2. On the Flows menu, click **Board Design**.
3. To start the Cadence Allegro Design Entry HDL software, click **Design Entry**.
4. To add the newly created symbol to your schematic, 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 attaches to your cursor for placement in the schematic. To fracture 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 (www.cadence.com).

FPGA-to-Board Integration with Cadence Allegro Design Entry CIS Software

The Cadence Allegro Design Entry CIS software is a schematic capture tool (part of the Cadence 200 series design flow based on OrCAD Capture CIS). Use the Cadence Allegro Design Entry CIS 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 the Cadence Allegro Design Entry CIS software. With the Cadence Allegro Design Entry CIS software, the design flow from FPGA-to-board is unidirectional using only the **.pin** generated by the Quartus II software. You can only make signal and pin assignment changes in the Quartus II software. These changes reflect as updated symbols in a Cadence Allegro Design Entry CIS schematic project. Figure 9–2 on page 9–4 shows the design flow with the Cadence Allegro Design Entry CIS software.

Routing or pin assignment changes made in a board layout tool or a Cadence Allegro Design Entry CIS symbol cannot be back-annotated to the Quartus II 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 (www.cadence.com) or go to the EMA Design Automation website (www.ema-eda.com).

Creating a Cadence Allegro Design Entry CIS Project

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.

To create a new project in the Cadence Allegro Design Entry CIS software, follow these steps:

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 only builds an FPGA logic design in the Cadence Allegro Design Entry CIS software.

Your new project is in the specified location and consists of the following files:

- OrCAD Capture Project File (**.opj**)
- Schematic Design File (**.dsn**)

Generating a Part

After you create a new project or open an existing project in the Cadence 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. The Cadence Allegro Design Entry CIS software stores component symbols in OrCAD Library File (**.olb**). When you place a symbol in a library attached to a project, it is immediately available for instantiation in the project schematic.

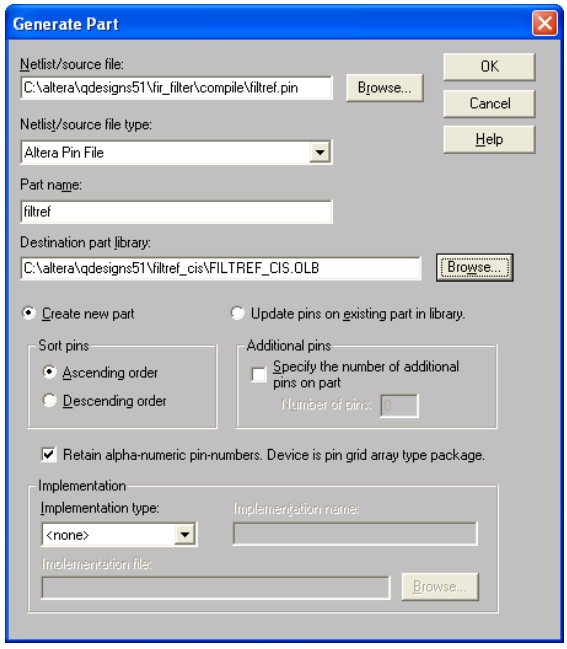
You can add symbols to an existing library or you can create a new library specifically for the symbols generated from your FPGA designs. To create a new library, follow these steps:

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. To specify a desired name and location for the library, right-click the new library and select **Save As**. Saving the new library creates the library file.

You can now create a new symbol to represent your FPGA design in your schematic. To generate a schematic symbol, follow these 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 9–6).

Figure 9–6. Generate Part Dialog Box

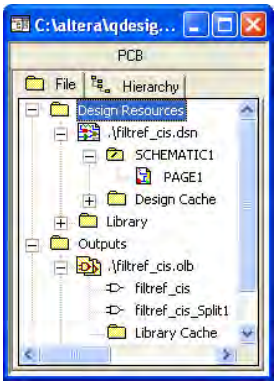


3. To specify the **.pin** from your Quartus II design, in the **Netlist/source file type** field, click **Browse**.
4. In the **Netlist/source file type** list, select **Altera Pin File**.

5. Type the new part name.
6. Specify the **Destination part library** for the symbol. Failing to select an existing library for the part creates a new library with a default name that matches the name of your Cadence Allegro Design Entry CIS project.
7. To create a new symbol for this design, select **Create new part**. If you updated your **.pin** in the Quartus II software and want to transfer any assignment changes to an existing symbol, select **Update pins on existing part in library**.
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** and does not require special implementation. Click **OK**.
9. Review the Undo warning and click **Yes** to complete the symbol generation.

You can locate the generated symbol in the selected library or in a new library found in the **Outputs** folder of the design in the Project Manager window (Figure 9–7). 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 9–7. Project Manager Window

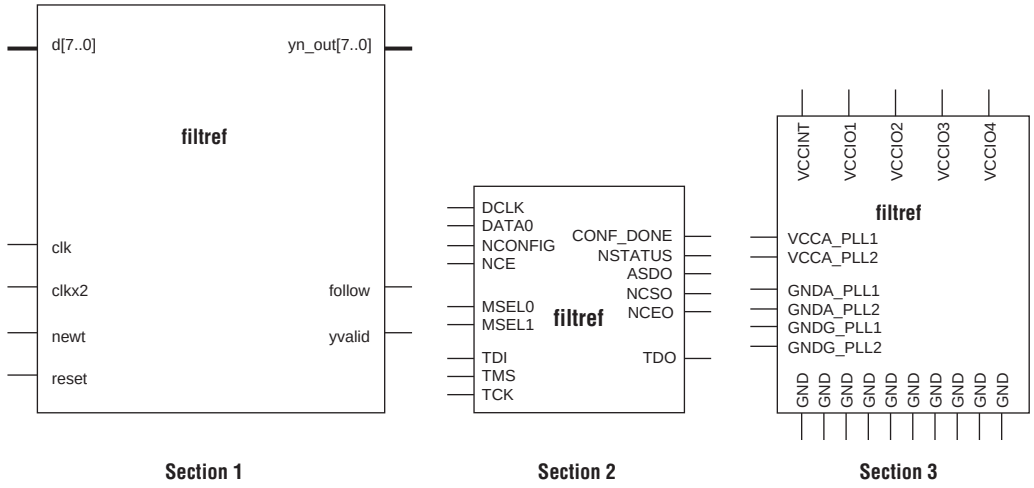


For more information about creating and editing symbols in the Cadence Allegro Design Entry CIS software, refer to the Help in the software.

Splitting a Part

After saving a new symbol in a project library, you can fracture the symbol into multiple parts called sections. Fracturing a part into separate sections is 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 can 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 9–8 shows a part fractured into separate sections.

Figure 9–8. Splitting a Symbol into Multiple Sections *(Notes 1)*, *(2)*



Notes to Figure 9–8:

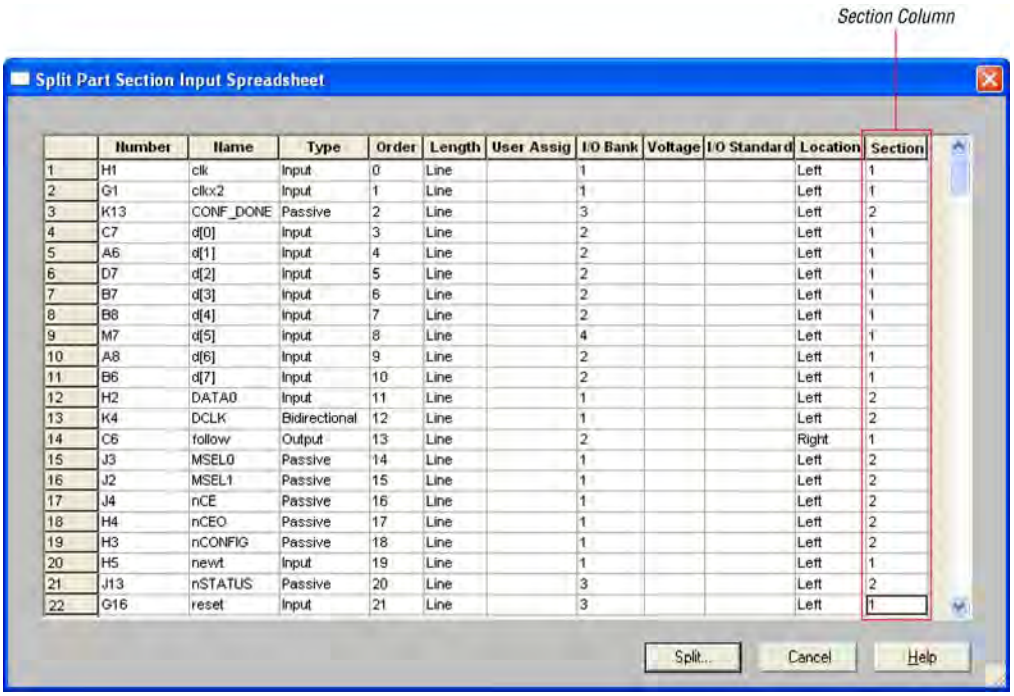
(1) Figure 9–8 represents a Cyclone device with JTAG or passive serial (PS) mode configuration option settings. Symbols created for other devices or other configuration modes might have different sets of configuration pins, but can be fractured in a similar manner.

(2) The power/ground section shows only a representation of power and ground pins because the device contains a high number of power and ground pins.

Although 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.

To split a part into sections, select the part in its library in the Project Manager window of the Cadence Allegro Design Entry CIS software. On the Tools menu, click **Split Part** or right-click the part and choose **Split Part**. The **Split Part Section Input Spreadsheet** appears (Figure 9–9).

Figure 9–9. 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 **Section** column indicates the section of the symbol to which each pin is assigned. You can locate all pins in a new symbol in section 1. You can change the values in the **Section** column to assign pins to various sections of the symbol. You can also specify the side of a section on the location of the pin by changing the values in the **Location** column. When you are ready, click **Split**. A new symbol appears in the same library as the original with the name <*original part name*>**_Split1.**

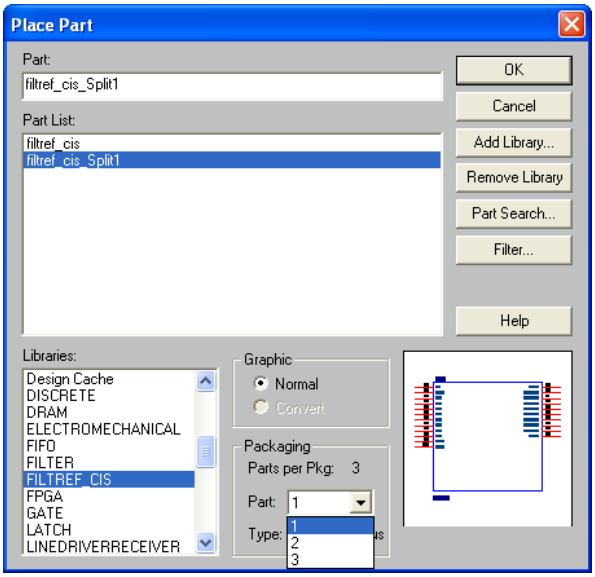
View and edit each section individually. To view the new sections of the part, double-click the part. The Part Symbol Editor window appears and the first section of the part displays for editing. On the View menu, click **Package** to view thumbnails of all the part sections. To edit the section of the symbol, double-click the thumbnail.

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.

Instantiating a Symbol in a Design Entry CIS Schematic

After saving a new symbol in a library in your Cadence Allegro Design Entry CIS project, you can instantiate the new symbol on a page in your schematic. Open a schematic page in the Project Manager window of the Cadence Allegro Design Entry CIS software. To add the newly created symbol to your schematic on the schematic page, on the Place menu, click **Part**. The **Place Part** dialog box appears (Figure 9–10).

Figure 9–10. 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 attaches to your cursor for placement in the schematic. To place the symbol, click on the schematic page.

For more information about using the Cadence Allegro Design Entry CIS software, refer to the Help in the software.

Altera Libraries for the Cadence Allegro Design Entry CIS Software

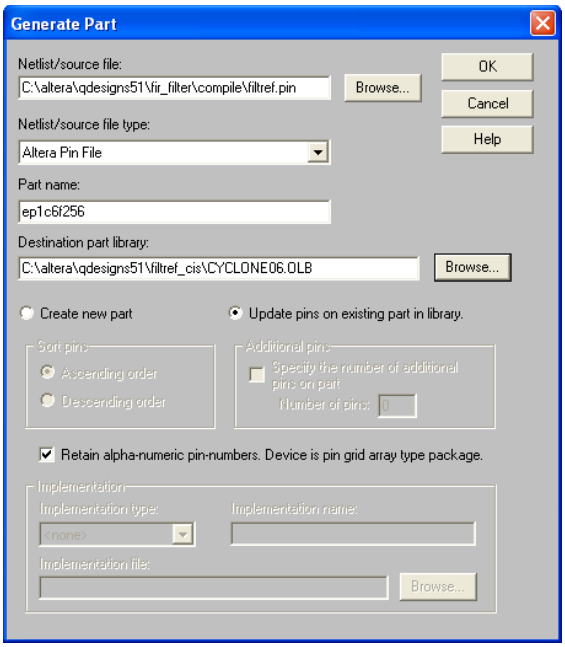
Altera provides downloadable **.olb** for many of its device packages. You can add these libraries to your Cadence Allegro Design Entry CIS project and update the symbols with the pin assignments contained in the **.pin** generated by the Quartus II software. You can use the downloaded library symbols as a base for creating custom schematic symbols with your pin assignments that you can edit or fracture. This method increases productivity by reducing the amount of time it takes to create and edit a new symbol.

To use the Altera-provided libraries with your Cadence Allegro Design Entry CIS project, follow these steps:

1. Download the library of your target device from the Download Center page found through the Support page on the Altera website (www.altera.com).
2. Create a copy of the appropriate **.olb** to maintain the original symbols. Place the copy in a convenient location, such as your Cadence Allegro 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** and add it to your project. You can locate the new library in the list of part libraries for your project.

4. On the Tools menu, click **Generate Part.** The **Generate Part** dialog box appears (Figure 9–11).

Figure 9–11. Generate Part Dialog Box



5. In the **Netlist/source file** field, click **Browse** to specify the **.pin** in your Quartus II design.
6. From the **Netlist/source file type** list, select **Altera Pin File**.
7. For **Part name**, type 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, type the part name to match one of the devices in this library such as `ep1c6f256`. You can rename the symbol in the Project Manager window after updating the part.
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.**
10. 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 fractured into sections, you can edit each section but you cannot further fracture the part. You can 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 and 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.

Document Revision History

Table 9–3 shows the revision history for this chapter.

Table 9–3. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ General style editing. ■ Removed Referenced Document Section. ■ Added a link to Help in "Performing Simultaneous Switching Noise (SSN) Analysis of Your FPGA" on page 9–5.
November 2009	9.1.0	■ Added "Performing Simultaneous Switching Noise (SSN) Analysis of Your FPGA" on page 9–5. ■ General style editing. ■ Edited Figure 9–4 on page 9–10 and Figure 9–8 on page 9–16.
March 2009	9.0.0	■ Chapter 9 was previously Chapter 7 in the 8.1 software release. ■ No change to content.
November 2008	8.1.0	Changed to 8-1/2 x 11 page size.
May 2008	8.0.0	Updated references.

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

Take an online survey to provide feedback about this handbook chapter.

10. Reviewing Printed Circuit Board Schematics with the Quartus II Software

QII52019-10.0.0

This chapter provides guidelines for reviewing printed circuit board (PCB) schematics with the Quartus® II software. Altera FPGAs and CPLDs offer a multitude of configurable options to allow you to implement a custom application-specific circuit on your PCB.

Your Quartus II project provides important information specific to your programmable logic design, which you can use in conjunction with the device literature available on Altera's website to ensure that you implement the correct board-level connections in your schematic.

This chapter highlights the important options in the Quartus II software, including **Settings** dialog box options, the Fitter report, and Messages window to which you should refer when creating and reviewing your PCB schematic. The Quartus II software also provides useful tools, such as the Pin Planner and the SSN Analyzer, to assist you during your PCB schematic review process.

The "Reviewing Quartus II Software Settings"section provides information about the settings you can make in the Quartus II software to help you review your PCB schematic. After verifying options in the Quartus II software, you can compile your design and use the data generated in the Fitter report, which is described in "Reviewing Device Pin-Out Information in the Fitter Report" on page 10–4 to verify settings in your PCB schematic. You should also ensure that you carefully review error and warning messages, as described in "Reviewing Compilation Error and Warning Messages" on page 10–5.

In addition to verifying your settings in the **Settings** dialog box and Fitter report, and checking messages, you can turn on additional settings, as described in "Using Additional Quartus II Software Features" on page 10–6 and "Running the HardCopy Design Readiness Check" on page 10–6.

Finally, Quartus II software tools, such as the Pin Planner and the SSN Analyzer, described in "Using Additional Quartus II Software Tools" on page 10–6, help you to verify proper I/O placement.

You should use this chapter in conjunction with Altera's device family-specific literature.

For more information, refer to the Schematic Review Worksheets and the Pin Connection Guidelines pages, which are available on the Altera.com website.

Reviewing Quartus II Software Settings

The **Device** dialog box in the Quartus II software allows you to specify device-specific assignments and settings. You can use the **Device** dialog box to specify general project-wide options, including specific device and pin options, which help you to implement correct board-level connections in your PCB schematic.

The **Device** dialog box provides project-specific device information, including the target device and any migration devices you specify. Using migration devices can impact the number of available user I/O pins and internal resources, as well as require connection of some user I/O pins to power/ground pins to support migration.

If you want to use vertical migration, which allows you to use different devices with the same package, you can specify your list of migration devices in the **Migration Devices** dialog box. The Fitter places the pins in your design based on your targeted migration devices, and allows you to use only I/O pins that are common to all of the migration devices.

For more information about the **Migration Devices** dialog box in the Quartus II software, refer to *Migration Devices Dialog Box* in Quartus II Help.

If a migration device has pins that are power or ground, but the pins are also user I/O pins on a different device in the migration path, the Fitter ensures that these pins are not used as user I/O pins. You must ensure that these pins are connected to the appropriate plane on the PCB.

If you are migrating from a smaller device with NC (no-connect) pins to a larger device with power or ground pins in the same package, you can safely connect the NC pins to power or ground pins to facilitate successful migration.

Device and Pins Options Dialog Box Settings

You can verify important design-specific data in the **Device and Pin Options** dialog box when reviewing your PCB schematic, including options found on the **Configuration**, **Unused Pin**, **Dual-Purpose Pins**, and **Voltage** pages.

Configuration Page Settings

The **Configuration** page of the **Device and Pin Options** dialog box specifies the configuration scheme and configuration device for the target device. Use the **Configuration** page settings to verify the configuration scheme with the MSEL pin settings used on your PCB schematic and the I/O voltage of the configuration scheme.

Your specific configuration settings may impact the availability of some dual-purpose I/O pins in user mode. Refer to "Dual-Purpose Pins Page Settings" on page 10–3 for more information.

Unused Pin Page Settings

The **Unused Pin** page specifies the behavior of all unused pins in your design. Use the **Unused Pin** page to ensure that unused pin settings are compatible with your PCB. For example, if you reserve all unused pins as outputs driving ground, you must ensure that you do not connect unused I/O pins to VCC pins on your PCB. Connecting unused I/O pins to VCC pins may result in contention that could lead to higher than expected current draw and possible device overstress.

The **Reserve all unused pins** list shows available unused pin state options for the target device. The default state for each pin is the recommended setting for each device family.

When you reserve a pin as output driving ground, the Fitter connects a ground signal to the output pin internally. You should connect the output pin to the ground plane on your PCB, although you are not required to do so. Connecting the output driving ground to the ground plane is known as creating a virtual ground pin, which helps to minimize simultaneous switching noise (SSN) and ground bounce effects.

Dual-Purpose Pins Page Settings

The **Dual-Purpose Pins** page specifies how configuration pins should be used after device configuration completes. You can set the function of the dual-purpose pins by selecting a value for a specific pin in the **Dual-purpose pins** list. Pin functions should match your PCB schematic. The available options on the **Dual-Purpose Pins** page may differ depending on the selected configuration mode.

Voltage Page Settings

The **Voltage** page specifies the default VCCIO I/O bank voltage and the default I/O bank voltage for the pins on the target device. VCCIO I/O bank voltage settings made in the **Voltage** page are overridden by I/O standard assignments made on I/O pins in their respective banks. Refer to the "Reviewing Device Pin-Out Information in the Fitter Report" on page 10–4 for more details about the I/O bank voltages for your design.

Error Detection CRC Page Settings

The **Error Detection CRC** page specifies error detection cyclic redundancy check (CRC) use for the target device. When **Enable error detection CRC** is turned on, the device checks the validity of the programming data in the devices. Any changes made in the data while the device is in operation generates an error.

Turning on the **Enable open drain on CRC error pin** option allows the CRC ERROR pin to be set as an open-drain pin in some devices, which decouples the voltage level of the CRC ERROR pin from VCCIO voltage. You must connect a pull-up resistor to the CRC ERROR pin on your PCB if you turn on this option.

In addition to settings in the **Device** dialog box, you should verify settings in the **Voltage** page of the **Settings** dialog box.

For more information about the **Device and Pins Options** dialog box in the Quartus II software, refer to *Device and Pin Options Dialog Box* in Quartus II Help.

Voltage Page Settings

The **Voltage** page, under **Operating Settings and Conditions** in the **Settings** dialog box, allows you to specify voltage operating conditions for timing and power analyses. Ensure that the settings in the **Voltage** page match the settings in your PCB schematic, especially if the target device includes transceivers.

The **Voltage** page settings requirements differ depending on the settings of the transceiver instances in the design. Refer to the Fitter report for the required settings, and verify that the voltage settings are correctly set up for your PCB schematic.

For more information about voltage settings, refer to the Pin Connection Guidelines page, which is available on the Altera.com website.

Once you verify your settings in the **Device** and **Settings** dialog boxes, you can verify your device pin-out with the Fitter report.

Reviewing Device Pin-Out Information in the Fitter Report

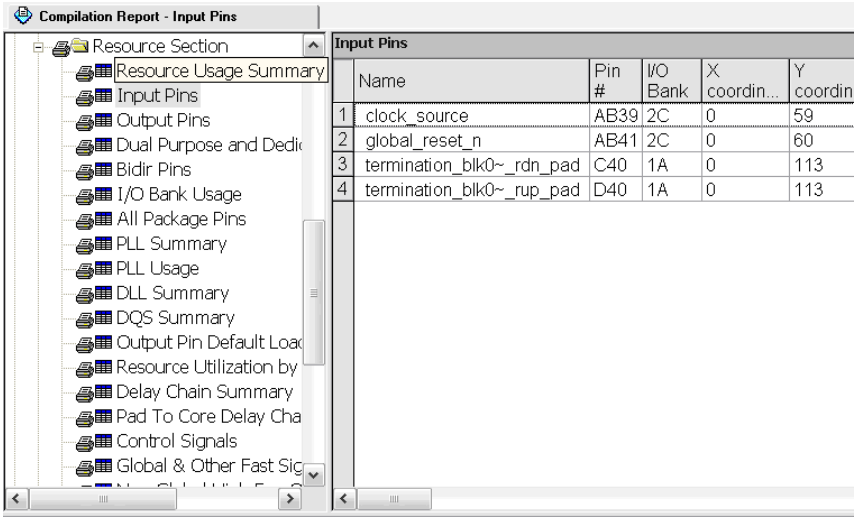
After you compile your design, you can use the reports in the Resource section of the Fitter report to check your device pin-out in detail.

The Input Pins, Output Pins, and Bidirectional Pins reports identify all the user I/O pins in your design and the features enabled for each I/O pin. For example, you can find use of weak internal pull-ups, PCI clamp diodes, and on-chip termination (OCT) pin assignments in these sections of the Fitter report. You can check the pin assignments reported in the Input Pins, Output Pins, and Bidirectional Pins reports against your PCB schematic to determine whether your PCB requires external components.

These reports also identify whether you made pin assignments or if the Fitter automatically placed the pins. If the Fitter changed your pin assignments, you should make these changes user assignments because the location of pin assignments made by the Fitter may change with subsequent compilations.

Figure 10–1 shows the pins the Fitter chose for the OCT external calibration resistor connections (RUP/RDN) and the name of the associated termination block in the Input Pins report. You should make these types of assignments user assignments.

Figure 10–1. Resource Section Report



The I/O Bank Usage report provides a high-level overview of the VCCIO and VREF requirements for your design, based on your I/O assignments. Verify that the requirements in this report match the settings in your PCB schematic. All unused I/O banks, and all banks with I/O pins with undefined I/O standards, default the VCCIO voltage to the voltage defined in the **Voltage** page of the **Device and Pin Options** dialog box.

The All Package Pins report lists all the pins on your device, including unused pins, dedicated pins and power/ground pins. You can use this report to verify pin characteristics, such as the location, name, usage, direction, I/O standard and voltage for each pin with the pin information in your PCB schematic. In particular, you should verify the recommended voltage levels at which you connect unused dedicated inputs and I/O and power pins, especially if you selected a migration device. Use the All Package Pins report to verify that you connected all the device voltage rails to the voltages reported.

Errors commonly reported include connecting the incorrect voltage to the predriver supply (VCCPD) pin in a specific bank, or leaving dedicated clock input pins floating. Unused input pins that should be connected to ground are designated as **GND+** in the **Pin Name/Usage** column in the All Package Pins report.

You can also use the All Package Pins report to check transceiver-specific pin connections and verify that they match the PCB schematic. Unused transceiver pins have the following requirements, based on the pin designation in the Fitter report:

- GXB_GND*—Unused GXB receiver or dedicated reference clock pin. This pin must be connected to GXB_GND through a 10k Ohm resistor.
- GXB_NC—Unused GXB transmitter or dedicated clock output pin. This pin must be disconnected.

Some transceiver power supply rails have dual voltage capabilities, such as VCCA_L/R and VCCH_L/R, that depend on the settings you created for the ALTGX MegaWizard Plug-In Manager. Because these user-defined settings overwrite the default settings, you should use the All Package Pins report to verify that these power pins on the device symbol in the PCB schematics are connected to the voltage required by the transceiver. An incorrect connection may cause the transceiver to function not as expected.

If your design includes a memory interface, the DQS Summary report provides an overview of each DQ pin group. You can use this report to quickly confirm that the correct DQ/DQS pins are grouped together. This section also provides information on DLL usage.

Finally, the Fitter Device Options report summarizes some of the settings made in the **Device and Pin Options** dialog box. Verify that these settings match your PCB schematics.

Reviewing Compilation Error and Warning Messages

If your project does not compile without error or warning messages, you should resolve the issues identified by the Compiler before signing off on your pin-out or PCB schematic. Error messages often indicate illegal or unsupported use of the device resources and IP.

Additionally, you should cross-reference fitting and timing analysis warnings with the design implementation. Timing may be constrained due to nonideal pin placement. You should investigate if you can reassign pins to different locations to prevent fitting and timing analysis warnings. Ensure that you review each warning and consider its potential impact on the design.

Running the HardCopy Design Readiness Check

You should run the HardCopy Design Readiness Check for designs including a HardCopy device in the migration path. This tool checks for issues that must be addressed prior to handing off the design to the Altera HardCopy Design Center for the HardCopy back-end process, including possible issues with device resource and I/O usage. Verify all warning messages generated during the HardCopy Design Readiness Check.

For more information about the HardCopy Design Readiness Check and designing for HardCopy devices in the Quartus II software, refer to the *Quartus II Support for HardCopy Series Devices* chapter in volume 1 of the *Quartus II Handbook*.

Using Additional Quartus II Software Features

You can generate IBIS files, which contain models specific to your design and selected I/O standards and options, with the Quartus II software.

Because board-level simulation is important to verify, you should check for potential signal integrity issues. You can turn on the **Board-Level Signal Integrity** feature in the **EDA Tool Settings** page of the **Settings** dialog box.

For more information about signal integrity analysis in the Quartus II software, refer to the *Signal Integrity Analysis with Third-Party Tools* chapter in volume 3 of the *Quartus II Handbook*.

Additionally, using advanced I/O timing allows you to enter physical PCB information to accurately model the load seen by an output pin. This feature facilitates accurate I/O timing analysis.

For more information about advanced I/O timing, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Using Additional Quartus II Software Tools

This section describes additional tools found in the Quartus II software, specifically the Pin Planner and the SSN Analyzer, and how you can use these tools to assist you with reviewing your PCB schematics.

Pin Planner

The Pin Planner provides a graphical representation of the target device and is a useful visual aid that shows the device package and I/O assignments. The Pin Planner allows you to view the I/O banks, VREF groups, edges and DQ/DQS pin groups, which enables you to verify the expected placement of pin groups.

You can use the Pin Planner to verify the location of clock inputs, and whether they have been placed on dedicated clock input pins, which is recommended when your design uses PLLs.

You can also use the Pin Planner to verify the placement of dedicated SERDES pins. SERDES receiver inputs can be placed only on DIFFIO_RX pins, while SERDES transmitter outputs can be placed only on DIFFIO_TX pins.

The Pin Planner gives a visual indication of signal-to-signal proximity in the Pad View window, and also provides information about differential pin pair placement, such as the placement of pseudo-differential signals.

For more information about the Pin Planner, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

SSN Analyzer

The SSN Analyzer supports pin planning by estimating the voltage noise caused by the simultaneous switching of output pins on the device. Because of the importance of the potential SSN performance for a specific I/O placement, you can use the SSN Analyzer to analyze the effects of aggressor I/O signals on a victim I/O pin.

For more information about the SSN Analyzer, refer to the *Simultaneous Switching Noise (SSN) Analysis and Optimizations* chapter in volume 2 of the *Quartus II Handbook*.

Conclusion

This chapter describes guidelines and descriptions of settings to verify when reviewing your PCB schematic with the Quartus II software. You can use settings in the **Settings** dialog box; information in the Fitter report and Messages window; and the Pin Planner and SSN Analyzer during the PCB schematic review process.

Document Revision History

Table 10–1 shows the revision history for this chapter.

Table 10–1. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	Initial release

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

Take an online survey to provide feedback about this handbook chapter.

Section III. Area, Timing, Power, and Compilation Time Optimization

This section introduces features in the Quartus® II software that you can use to optimize area, timing, power, and compilation time when you design for programmable logic devices (PLDs).

This section includes the following chapters:

- Chapter 11, Design Optimization Overview

  This chapter summarizes features in the Quartus II software that you can use to achieve the highest design performance when you design for PLDs, especially high density FPGAs.

- Chapter 12, Reducing Compilation Time

  This chapter describes techniques for reducing the amount of time it takes to compile and recompile your design, accelerating your design process.

- Chapter 13, Area and Timing Optimization

  This chapter describes a broad spectrum of Quartus II software features and design techniques to reduce resource usage and improve timing performance when designing for Altera® devices. This chapter also explains how and when to use some of the features described in other chapters of the *Quartus II Handbook*.

- Chapter 14, Power Optimization

  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.

- Chapter 15, Analyzing and Optimizing the Design Floorplan

  You can use the Chip Planner to perform design analysis and create a design floorplan. This chapter discusses how to analyze and optimize the design floorplan with the Chip Planner.

- Chapter 16, Netlist Optimizations and Physical Synthesis

  This chapter explains how the physical synthesis optimizations in the Quartus II software can improve your quality of results. This chapter also provides information about preserving and writing out a new netlist, and provides guidelines for applying the various options.

11. Design Optimization Overview

qii52021-10.0.1

This chapter 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.

Introduction

Physical implementation can be an intimidating and challenging phase of the design process. 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 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. .

Physical Implementation

Most optimization issues involve 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.

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.

For more information, refer to the *Design Planning with the Quartus II Software* chapter in volume 1 of the *Quartus II Handbook*.

Trade-Offs and Limitations

Many optimization goals can conflict with one another, so you might need 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, such as 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 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 limitations, timing constraints, or power constraints that cannot be resolved by the Fitter, consider rewriting parts of the HDL code.

For more information, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*

Preserving Results and Enabling Teamwork

For some Quartus II Fitter algorithms, 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 cannot meet timing on a portion of your design, you can partition that 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 optimization techniques often result in longer compilation times, but by applying them only on specific partitions, you can reduce this impact and complete iterations more quickly.

In addition, by physically floorplanning your partitions with LogicLock regions, you can enable team-based flows and allow multiple people to work on different portions of the design.

For more information, refer to *Quartus II Incremental Compilation for Hierarchical and Team-Based Designs* in volume 1 of the *Quartus II Handbook* and *About Incremental Compilation* in Quartus II Help.

Reducing Area

By default, the Quartus II Fitter might phyically spread a design over the entire device 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 reduced area, you can enable certain physical synthesis options to modify your netlist to create a more area-efficient implementation, but at the cost of increased runtime and decreased performance.

For more information, refer to the *Area and Timing Optimization* and *Netlist Optimizations and Physical Synthesis* chapters in volume 2 and the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

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 the specified timing requirements and stops trying when the requirements are met. Therefore, using realistic constraints is important to successfully close timing. If you under-constrain your design, you may 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 may also increase because of excessively tight constraints.

If your resource usage 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 you improve 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 one 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 reduce delay on 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 and resource usage. Alternately, you can modify your HDL to manually duplicate or retime logic.

Reducing Power Consumption

The Quartus II software has features that help reduce design power consumption. The PowerPlay power optimization options control the power-driven compilation settings for Synthesis and the Fitter.

For more information, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*.

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

The following sections describe several Quartus II tools that you can use to help optimize your design.

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 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.

For more information, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Designs* and *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapters in volume 1 and the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Advisors

The Quartus II software includes several advisors to help you optimize your design and reduce compilation time. You can complete your design faster by following the recommendations in the Compilation Time Advisor, Incremental Compilation Advisor, Timing Optimization Advisor, Area Optimization Advisor, Resource Optimization Advisor, and Power Optimization Advisor. These advisors give recommendations based on your project settings and your design constraints.

For more information about advisors, refer to Quartus II Help.

Design Space Explorer

Use the Design Space Explorer (DSE) to find optimal 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, based on your chosen primary optimization goal. You can try different seeds with the DSE if you are fairly close to meeting your timing or area requirements and find one seed that meets timing or area requirements. Finally, the DSE can run the different compilations on multiple computers in parallel, which shortens the timing closure process.

For more information, refer to *About Design Space Explorer* in Quartus II Help.

Conclusion

The Quartus II software includes a number of features and tools that you can use to optimize area, timing, power, and compilation time when you design for programmable logic devices (PLDs).

Document Revision History

Table 11–1 shows the revision history for this chapter.

Table 11–1. Document Revision History

Date	Version	Changes Made
August 2010	10.0.1	■ Corrected link
July 2010	10.0.0	■ Initial release. Chapter based on topics and text in Section III of volume 2.

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

Take an online survey to provide feedback about this handbook chapter.

12. Reducing Compilation Time

QII52022-10.0.0

The Quartus® II software offers a number of features and techniques to help reduce compilation time.

This chapter describes techniques to reduce compilation time when designing for Altera® devices, and includes the following topics:

- "Compilation Time Optimization Techniques"
- "Compilation Time Advisor" on page 12–2
- "Strategies to Reduce the Overall Compilation Time" on page 12–2
- "Reducing Synthesis Time and Synthesis Netlist Optimization Time" on page 12–5
- "Reducing Placement Time" on page 12–6
- "Reducing Routing Time" on page 12–8
- "Setting Process Priority" on page 12–9

Compilation Time Optimization Techniques

In long compilations, most of the time is spent in the Analysis and Synthesis and Fitter modules. Analysis and 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 and Synthesis and Fitter modules. The Fitter Messages report in the **Fitter** section of the Compilation Report shows the time that was spent in placement and the time that 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 uses 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 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 example, *x* indicates the number of hours the process has run.

Compilation Time Advisor

A Compilation Time Advisor is available in the Quartus II software, which helps you to reduce compilation time. 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.

Strategies to Reduce the Overall Compilation Time

This section discusses strategies you can use to reduce overall compilation time, including the following topics:

- "Using Parallel Compilation with Multiple Processors"
- "Using Incremental Compilation" on page 12–3
- "Using the Smart Compilation Setting" on page 12–4
- "Using Rapid Recompile" on page 12–4

Using Parallel Compilation with Multiple Processors

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. Parallel compilation is turned on by default in the Quartus II software and the software can detect if multiple processors are available. 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 16 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.

If you have partitioned your design and enabled parallel compilation, the Quartus II software can use different processors to compile those partitions simultaneously during the Analysis and Synthesis stage. This may result is high peak memory usage during Analysis and Synthesis.

By allowing the Quartus II software to use two processors, you can reduce the compilation time by up to 10% on systems with two processing cores and by up to 20% 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 10% 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 multi-corner optimization turned on benefit more from using multiple processors than do compilations that do not use multi-corner optimization. The runtime requirement is not reduced for some other compilation goals, such as Analysis and Synthesis. The Fitter (`quartus_fit`) and the TimeQuest Timing Analyzer (`quartus_sta`) stages in

the compilation can, in certain cases, 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 HardCopy III, HardCopy IV, Arria® series, Cyclone®, MAX® II, and Stratix® 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.

The Quartus II software can detect the number of processors available on a computer and use available processors to reduce compilation time. You can also control the number of processors used during a compilation on a per user basis.

For more information, refer to *Processing Page (Options Dialog Box)* in Quartus II Help.

For information about how to control the number of processors used during compilation for a specific project, refer to *Compilation Process Settings Page (Settings Dialog Box)* in Quartus II Help.

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 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 `16`.

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 ↵
```

Using Incremental Compilation

The incremental compilation feature can speed up design iteration time by up to 70% for small design changes, and helps you reach design timing closure more efficiently. Design iterations can be made 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.

In the standard incremental compilation design flow, the top-level design is divided into partitions, which can be compiled and optimized together in the top-level Quartus II project. You can preserve fitting results and performance for completed partitions while other parts of the design are changing, which reduces the compilation time for each design iteration because neither synthesis nor fitting is performed for unchanged partitions in the design.

The incremental compilation feature also facilitates team-based incremental compilation design flows that allow designers to create and optimize design blocks independently, when necessary, and support third-party IP integration.

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*.

For additional information about incremental compilation, refer to *About Incremental Compilation* in Quartus II Help.

Using the 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**.

Smart compilation 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.

Using Rapid Recompile

The Rapid Recompile feature maximizes designer productivity when making small engineering change order (ECO)-style design changes after a full compilation, reducing compilation times by an average of 50%. Rapid Recompile also significantly improves designer productivity during timing closure by preserving critical timing during late design changes.

The Rapid Recompile option can be used stand-alone or along with standard incremental flow for compatible nodes in your design. A compatible node is a node that can be matched to a node from previous compilation results. Rapid Recompile allows the Quartus II software to reuse placement and routing resources of compatible nodes from previous results with a high degree of confidence.

If you enable the Rapid Recompile feature, you will see the compile time reduction after a full compilation. Turn on the Rapid Recompile feature in later compilations to see further reductions. The **Incremental Compilation Preservation Summary** section in the Fitter Report provides details about the placement and routing preservation for your design.

The performance of Rapid Recompile is largely dependent on the nature of the design change. If the Quartus II software determines that full optimization is necessary for design performance, you may not see much compilation time reduction. Applying extensive, global optimizations to a small user change may be required to obtain optimal performance. Be sure to select the right flow to achieve your end goals.

If you see the message `Fitter has failed to locate previous placement information` during the compilation of your design, Rapid Recompile will not provide any compile time reduction.

For more information about this feature, refer to *Incremental Compilation Page (Settings Dialog Box)* in Quartus II Help.

Reducing Synthesis Time and Synthesis Netlist Optimization Time

You can reduce synthesis time by reducing your use of netlist optimizations and by using incremental compilation (with Netlist Type set to **Post-Synthesis**) without affecting the Fitter time. For tips for reducing synthesis time when using third-party EDA synthesis tools, refer to your synthesis software's documentation.

Settings to Reduce Synthesis Time and Synthesis Netlist Optimization Time

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. When using Quartus II Integrated Synthesis, you can also enable specific Physical Synthesis Optimizations during Analysis and Synthesis. Using these netlist optimizations can cause the Analysis and Synthesis module to take much longer to run. Read the Analysis and Synthesis messages to find out how much time these optimizations take. The compilation time spent in Analysis and Synthesis is usually 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.

Turn off settings that are not useful. In general, if you carry over compilation settings from a previous project, evaluate all settings and keep only those that you need.

Use Appropriate Coding Style to Reduce Synthesis Time

The way you code your design in HDL can effect the synthesis time. For example, if you want to infer RAM blocks from your code, you must follow the guidelines for inferring RAMs. If not, those blocks are implemented as registers, and if you are trying to infer a large memory, it will use a large amount of resources within the FPGA. This will cause routing congestion and increase compilation time drastically. In general, if you see very high routing utilizations in some blocks, it is a good idea to review the code for such blocks.

For more information about coding guidelines, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Using 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 2.5× faster (on average) than running a full compilation (8.5× faster in best case), although the fit is not fully optimized or routed. Therefore, the timing report is only an estimate. On average, the estimated delays are within 15% of the final timing results as achieved by a full compilation.

You can specify the type of delay estimates to use with Early Timing Estimation. 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 view the critical paths in the design by locating these paths in the Chip Planner from the TimeQuest Timing Report panel. 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 Estimate to quickly assess the impact of any floorplan assignments or logic changes, enabling you to try different design variations and find the best solution.

Reducing 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

The highest fitter effort setting, Standard Fit, takes the most runtime, but does not always yield a better result than using the default **Auto Fit**. For designs with very tight timing requirements, both **Auto Fit** and **Standard Fit** use the maximum effort during optimization. Using **Auto Fit** is recommended for reducing compilation time. If you are certain that your design has only easy-to-meet timing constraints, you can select **Fast Fit** for an even greater runtime savings.

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, click **Settings**. The **Settings** dialog box appears. From the **Category** list, select **Fitter Settings**, and then 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 **Compilation Process 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** physical synthesis effort 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" in the *Area and Timing Optimization* chapter in volume 2 of the Quartus II Handbook.

Preserving Placement, Incremental Compilation, and LogicLock Regions

Preserving information about previous placements can make future placements faster. The incremental compilation feature provides an easy-to-use methodology for preserving placement results. For more information, refer to "Using Incremental Compilation" on page 12–3.

Reducing 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

Identifying Routing Congestion in the Chip Planner

To identify areas of routing congestion 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 **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.

In some cases, routing congestion may be a result of the HDL coding style used in the design. Once you identity areas that are congested using the Chip Planner, review the HDL code for the blocks placed in those areas to see if interconnect usage can be reduced by code changes.

The Quartus II Fitter report contains information about the peak interconnect usage in the design. High peak interconnect usage often results in longer compilation time.

For information about identifying areas of congested routing using the Chip Planner, refer to the "Viewing Routing Congestion" subsection in the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Placement Effort Multiplier Setting

Some designs might be time consuming and difficult to route because the placement is not optimal. In such cases, you can increase the Placement Effort Multiplier to get a better placement. Doing so might increase the placement time, but it can reduce the routing time, and even overall compilation time in some cases.

Preserving Routing Incremental Compilation and LogicLock Regions

Preserving information about the previous routing results for part of the design can reduce future routing time. LogicLock regions used with incremental compilation provides an easy-to-use methodology that preserves placement and routing results. For more information, refer to "Using Incremental Compilation" on page 12–3 and the references listed in the section.

Setting Process Priority

It might be necessary to reduce the computing resources allocated to the compilation at the expense of increased compilation time. It can be convenient to reduce the resource allocation to the compilation with single processor machines if you also have to run other tasks at the same time.

For more information, refer to *Processing Page (Options Dialog Box)* in Quartus II Help.

Conclusion

The Quartus II software provides many features to reduce compilation time and achieve optimal results. Using the recommended techniques described in this chapter can help you reduce compilation time.

Document Revision History

Table 12–1 shows the revision history for this chapter.

Table 12–1. Document Revision History

Date and Document Version	Version	Changes Made
July 2010	10.0.0	Initial release.

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

Take an online survey to provide feedback about this handbook chapter.

13. Area and Timing Optimization

QII52005-10.0.1

This chapter describes techniques to reduce resource usage and improve timing performance when designing for Altera® devices.

Good optimization techniques are essential for achieving the best results when designing for programmable logic devices (PLDs). The optimization features available in the Quartus® II software allow you to meet design requirements by applying these techniques at multiple points in the design process.

This chapter also explains how and when to use some of the features described in other chapters of the *Quartus II Handbook*.

Topics in this chapter include:

- "Optimizing Your Design" on page 13–2
- "Design Analysis" on page 13–9
- "Resource Utilization Optimization Techniques (LUT-Based Devices)" on page 13–16
- "Timing Optimization Techniques (LUT-Based Devices)" on page 13–29
- "Resource Utilization Optimization Techniques (Macrocell-Based CPLDs)" on page 13–46
- "Timing Optimization Techniques (Macrocell-Based CPLDs)" on page 13–52
- "Scripting Support" on page 13–56

The application of these techniques varies from design to design. Applying each technique does not always improve 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 results and to use them only if they improve results for your design.

You can 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 on your design. "Initial Compilation: Required Settings" on page 13–3 provides guidelines for some of the settings and assignments that are recommended for your initial compilation."Initial Compilation: Optional Settings" on page 13–6 describes settings that you might turn on based on your design requirements. "Design Analysis" on page 13–9 explains how to analyze the compilation results.

You can use incremental compilation in the optimization process. Incremental compilation can preserve timing to aid in timing closure, as well as compilation time reduction; however, it can cause a slight increase in resource utilization.

For more details about Quartus II incremental compilation flow, 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 13–16 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 13–58 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 13–59.

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 13–46 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 13–52.

For device-independent techniques to reduce compilation time, refer to the *"Compilation-Time Optimization Techniques"* section in the *Reducing Compilation Time* chapter in volume 2 of the *Quartus II Handbook*.

You can use these techniques in the GUI or with Tcl commands. For more information about scripting techniques, refer to "Scripting Support" on page 13–56.

Initial Compilation: Required Settings

This section describes the basic assignments and settings for your initial compilation. Check the following compilation assignments before compiling the design in the Quartus II software. Significantly varied compilation results can occur depending on the assignments you set.

Verify the following settings:

- "Device Settings"
- "I/O Assignments"
- "Timing Requirement Settings" on page 13–4
- "Device Migration Settings" on page 13–5
- "Partitions and Floorplan Assignments for Incremental Compilation" on page 13–5

Device Settings

Specific device assignments determine 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 the number of resources available in the device.

To select 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 select pin locations automatically. If your pin locations are not fixed due to PCB layout requirements, leave pin locations unconstrained. If your pin locations are already fixed, make pin assignments to constrain the compilation appropriately."Resource Utilization Optimization Techniques (Macrocell-Based CPLDs)" on page 13–46 includes recommendations for making pin assignments that can have a large effect on your 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 *About the Assignment Editor* in Quartus II Help.

Timing Requirement Settings

Using comprehensive timing requirement settings is an important step for achieving the best 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.
- The Quartus II software performs physical synthesis optimizations based on timing requirements (refer to "Physical Synthesis Optimizations" on page 13–35 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 description of the different effort levels, refer to "Fitter Effort Setting" on page 13–8

Use your real requirements to get the best results. If you apply more demanding timing requirements than you actually need, increased resource usage, higher power utilization, increased compilation time, or all of these may result.

The TimeQuest Timing Analyzer checks your design against the timing constraints. 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 create timing constraints for the Quartus II TimeQuest Timing Analyzer, create a Synopsys Design Constraint (**.sdc**) file. You can also enter constraints in the TimeQuest GUI. Use the `write_sdc` command, or, on the Constraints menu in the TimeQuest Timing Analyzer, click **Write SDC File** to write your constraints to an **.sdc** file. You can add an **.sdc** file to your project on the **Quartus II Settings** page under **Timing Analysis Settings**.

If you already have an **.sdc** file in your project, using the **write_sdc** command from the command line or using the **Write SDC File** option from the TimeQuest GUI can be used to create a new **.sdc** file, combining the constraints from your current **.sdc** file and any new constraints added through the GUI or command window, or can be used to overwrite the existing **.sdc** file with your newly applied constraints.

Ensure that every clock signal has an accurate clock setting constraint. 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 constraints. All I/O pins that require I/O timing optimization must be constrained. You should specify both minimum and maximum 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 a global constraint.

Make any complex timing assignments required in the design, including false path 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* chapter in volume 3 of the *Quartus II Handbook* and the *Quartus II TimeQuest Timing Analyzer Cookbook*. For more information about 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 in the **Task** pane or the `report_ucp` Tcl command.

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 the **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 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 can 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 incremental compilation for your design with good design partitioning methodology 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 may also help reduce compilation times.

For more information, refer to the *"Incremental Compilation"* section in the *Reducing Compilation Time* chapter in volume 2 of the *Quartus II Handbook*.

If you want to take advantage of incremental compilation 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 and create a floorplan for your design partitions. Good assignments can improve your results. Assignments can negatively affect a design's results if you do not follow Altera's recommendations.

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

Initial Compilation: Optional Settings

This section describes optional settings that can help to compile your design. You can selectively set all the optional settings that help to improve performance. 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"
- "Optimize Hold Timing" on page 13–6
- "Limit to One Fitting Attempt" on page 13–7
- "Optimize Multi-Corner Timing" on page 13–7
- "Fitter Effort Setting" on page 13–8

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 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.

For more information about the Design Assistant, refer to *About the Design Assistant* and *Analyzing Designs with the Design Assistant* in Quartus II Help.

Optimize Hold Timing

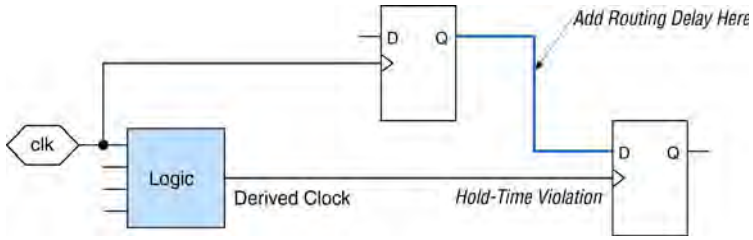
The **Optimize Hold Timing** option directs the Quartus II software to optimize minimum delay timing constraints. This option is available for all Altera device families except MAX 3000 and MAX 7000 series devices. By default, the Quartus II software optimizes hold timing for all paths for designs using devices newer than Arria GX, Stratix III, and Cyclone III. By default, the Quartus II software optimizes hold timing 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** pane, if you select **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 13–1, 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 13–1. 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*.

Limit to One Fitting Attempt

A design might fail to fit for several reasons, such as logic overuse or illegal assignments. For most 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 of these fit attempts takes significantly longer than the previous attempt.

For large designs, you might not want to wait for all three fitting attempts to be completed. To have the Quartus II software issue an error message after the first failed attempt, turn on **Limit to one fitting attempt** on the **Fitter Settings** page.

Refer to "Routing" on page 13–25 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.

Also, because of the small process geometries of the Cyclone III, Stratix III, and newer device families, the slowest circuit performance of designs targeting these devices does not necessarily occur at the highest operating temperature. The temperature at which the circuit is slowest depends on the selected device, the design, and the Quartus II compilation results. Therefore, the Quartus II software provides the Cyclone III series, Stratix III, and newer device families with three different timing corners—Slow 85°C corner, Slow 0°C corner, and Fast 0°C corner. For other device families two timing corners are available—Fast 0° C and Slow 85° C corner.

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 considering both the fast and slow timing corners, at the cost of a slight increase in runtime. By optimizing for both the fast and slow process corners, you can create a design implementation that is more robust across process, temperature, and voltage variations. While optimizing for multi-corner timing, the Fitter chooses one of the two slow corners that is known to have more critical timing (depending on the chosen device), along with the fast corner. This option is available only for Arria, Cyclone, HardCopy, MAX II, and Stratix series of devices.

To turn on the **Optimize multi-corner timing** option, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Fitter Settings**.
3. Turn on **Optimize multi-corner timing**.

Using the 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 multi-corner timing** setting.

Fitter Effort Setting

Fitter effort refers to the amount of effort the Quartus II software uses to fit your design. 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, Cyclone, HardCopy, MAX II, and Stratix series of devices) focuses the full Fitter effort only 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 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, perform a sweep over multiple seeds to ensure that any performance improvements that you see are real gains. For more information, refer to "Fitter Seed" on page 13–41.

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 possible performance if the design includes no timing assignments.

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.

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, and there is no adverse effect on the dynamic power consumption relative to using **Standard Fit**. 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. The **Standard Fit** setting usually increases compilation time relative to **Auto Fit**, because it applies full optimization, regardless of the design requirement. In designs with no timing assignments, on average, using the **Standard Fit** option results in a f_{MAX} about 10% higher than that achieved using the **Auto Fit** option. In designs where timing requirements can be easily met, using the **Standard Fit** option can result in considerably longer compilation times than using the **Auto Fit** option.

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 timing requirements. This section describes how to analyze your design results in the Quartus II software.

Error and Warning Messages

After compiling your design, it is important to evaluate all error and warning messages to see if any design or setting changes are required. If changes are required, make these changes and recompile the design before proceeding with design optimization.

To suppress messages that you have already evaluated and do not want to see again, 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 Constraints

The Quartus II software ignores illegal, obsolete, and conflicting constraints.

You can view a list of ignored constraints by clicking **Report Ignored Constraints** in the Reports menu in the TimeQuest GUI or by typing the following command to generate a list of ignored timing constraints:

```
report_sdc -ignored -panel_name "Ignored Constraints" ↵
```

If any constraints were ignored, analyze why they were ignored. If necessary, correct the constraints and recompile the design before proceeding with design optimization.

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 might introduce fitting difficulties. Also, review the resource utilization information to determine if it is impacting timing performance.

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, 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 series, Stratix II, Stratix III, Stratix IV, and Stratix V devices, a device with low utilization does not necessarily 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 routing 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. In the Compilation Report, look for errors and warnings that indicate these types of problems.

Refer to "Limit to One Fitting Attempt" on page 13–7 for more information about how to get a quick error message on hard-to-fit designs.

You can use the Chip Planner to find areas of the device that have routing congestion. If you find areas with very high congestion, analyze the cause of the congestion. Issues such as high fan-out nets not using global resources, an improperly chosen optimization goal (speed versus area), very restrictive floorplan assignments, or the coding style can cause routing congestion. After you identify the cause, modify the source or settings to reduce routing congestion.

For details about using the Chip Planner tool, 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 the TimeQuest Timing Analyzer 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 to specify the data arrival time at an output port's receiver with respect to a given clock. You can use the `report_timing` Tcl command to generate the I/O timing reports.

The I/O paths that do not meet the required timing performance are reported as having negative slack and are highlighted in red in the TimeQuest Timing Analyzer **Report** pane. In cases where you do not apply an explicit I/O timing constraint 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.

For more information about how timing numbers are calculated, refer to 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" on page 13–14
- "Tips for Analyzing Failing Clock Paths that Cross Clock Domains" on page 13–15

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.

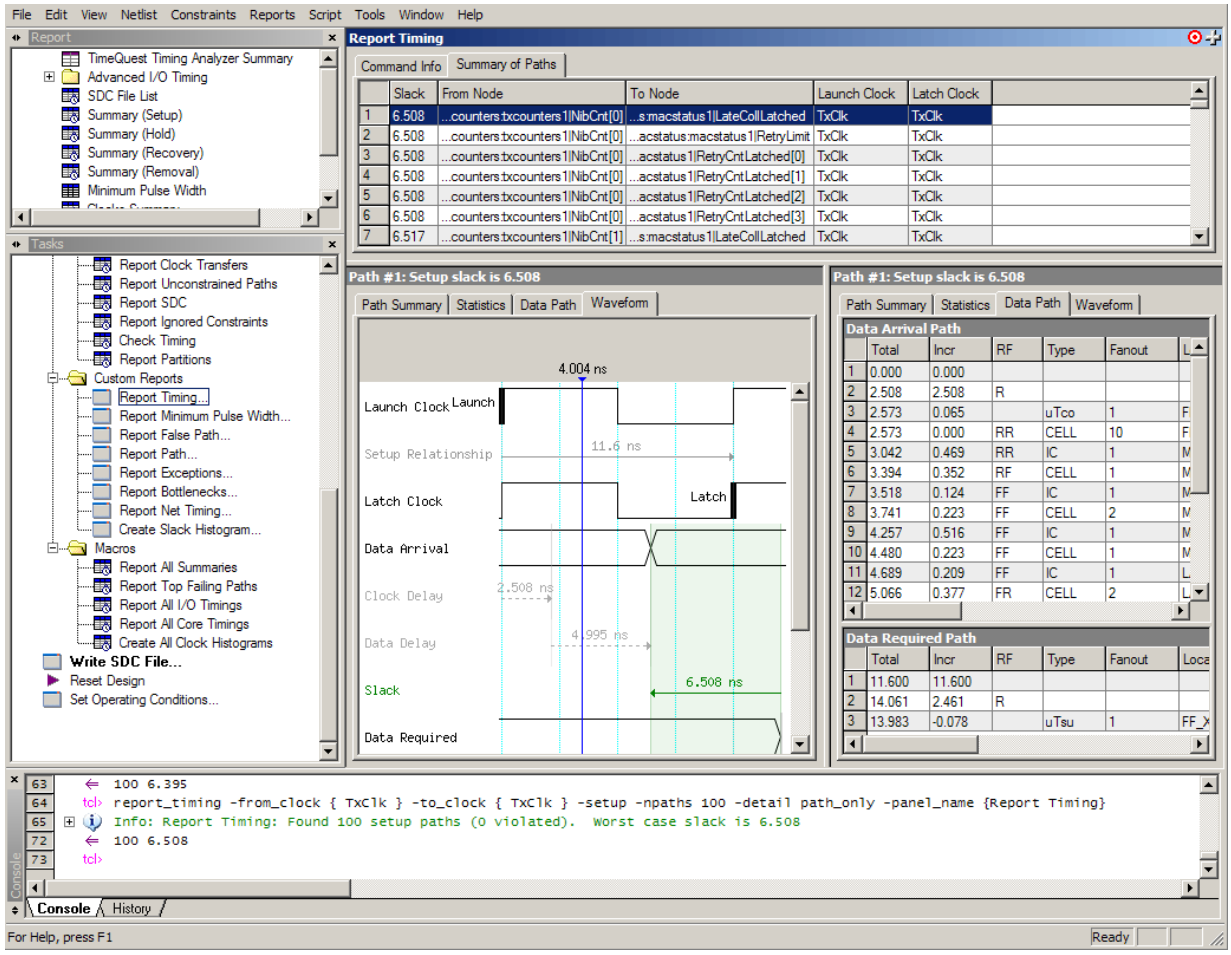
All paths that do not meet the timing requirement are shown with a negative slack and appear in red in the TimeQuest Timing Analyzer GUI.

When you select a path listed in the TimeQuest **Report Timing** pane, the tabs in the corresponding path detail pane show a path summary of source and destination registers and their timing, statistics about the path delay, detailed information about the complete data path with all nodes in the path and the waveforms of the relevant signals (Figure 13–2). To locate a selected path in the Chip Planner or the Technology Map Viewer by using the shortcut menu, right-click on a path, point to **Locate**, and click **Locate in Chip Planner**. The Chip Planner appears with the path highlighted. Similarly, if you know that a path is not a valid path, you can set it to be a false path using the shortcut menu.

To see the path details of any selected path, click on the **Data Path** tab in the path details pane. 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.

You can locate critical paths in the Chip Planner from the TimeQuest timing analysis report panel.

Figure 13–2. TimeQuest Timing Analyzer GUI



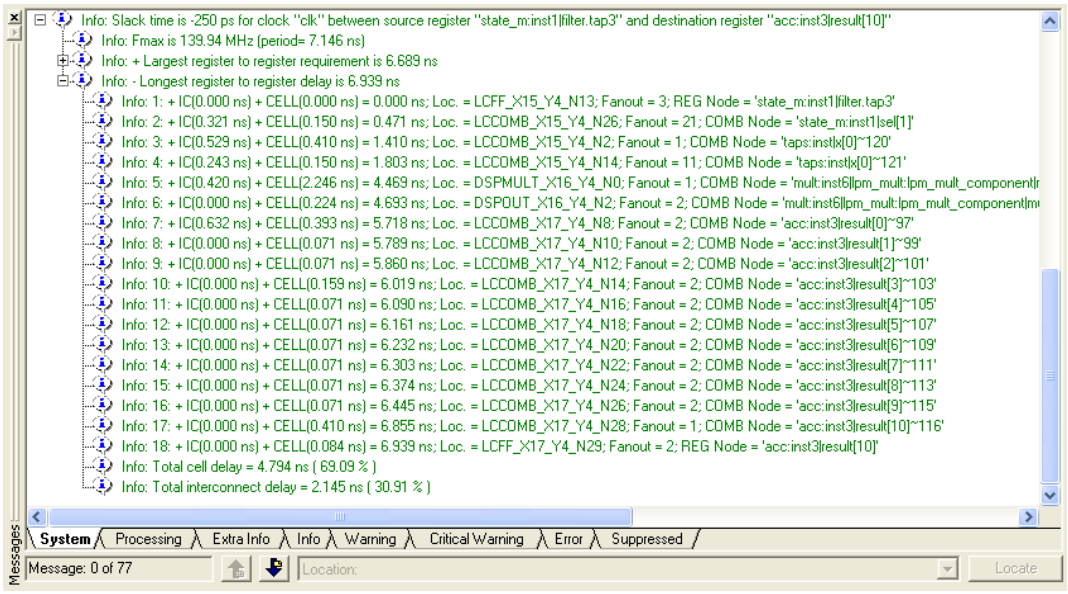
Timing Analysis with the Classic Timing Analyzer

If you are using the Quartus II Classic Timing Analyzer, in the Compilation Report window, refer to the **Timing Analyzer** section to determine whether register-to-register timing requirements are met. The **Clock Setup** folder displays setup slacks between registers on each clock domain in the design. The paths that do not meet timing requirements have 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 13–3). Click the "+" icon at the beginning of the line to see 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 13–3. 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**. The Chip Planner appears with the path highlighted.

For more information about how timing analysis results are calculated, refer to the *Quartus II TimeQuest Timing Analyzer* chapter or the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

You also can see the logic in a particular path by locating the logic in 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 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, examine reports and waveforms to determine if the correct constraints are being applied, and add multicycle or false paths as appropriate.

Focus on improving the paths that show the worst slack. The Fitter works hardest on paths with the worst 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 to see if the PLL phase shift is reducing the setup requirement. You might be able to adjust this using PLL parameters and settings.

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 TimeQuest Timing Analyzer* chapter or the *Quartus II Classic 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. Use the **Global Signal** setting in the Assignment Editor to control global routing resources.

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 13–3 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 13–58. 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 conflict with 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 13–18).
- Issues relating to routing (refer to "Routing" on page 13–25).

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 select 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 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, select **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 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.

For more information about this option, refer to *Restructure Multiplexers logic option* in Quartus II Help.

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*.

Perform WYSIWYG Primitive Resynthesis with Balanced or Area Setting

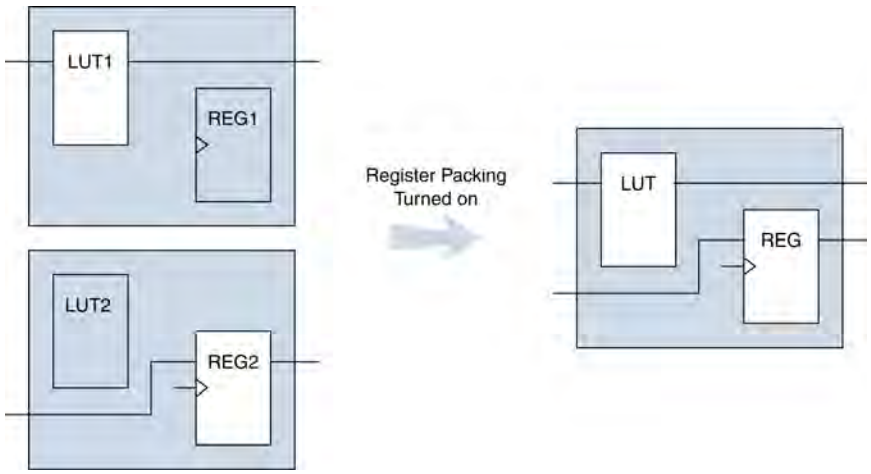
For information about this logic option, refer to *Perform WYSIWYG Primitive Resynthesis logic option* in Quartus II Help.

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 cases. Performing WYSIWYG resynthesis for area in this way typically reduces register-to-register timing performance.

Use Register Packing

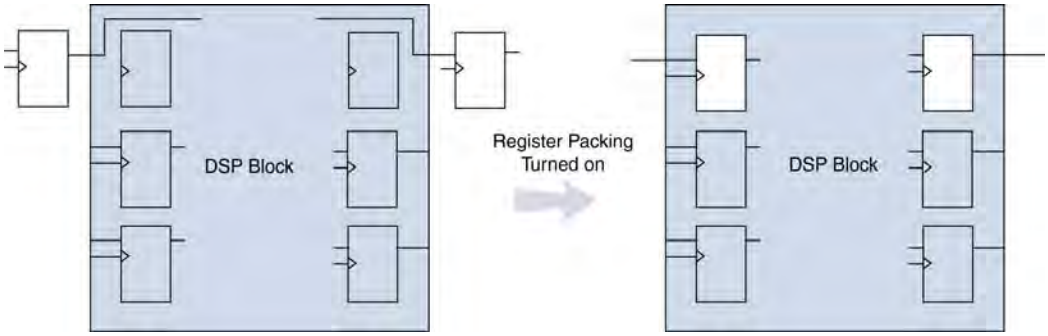
The **Auto Packed Registers** option implements the functions of two cells into one logic cell by combining the register of one cell in which only the register is used with the LUT of another cell in which only the LUT is used. Figure 13–4 shows register packing and the gain of one logic cell in the design.

Figure 13–4. Register Packing



Registers can also be packed into DSP blocks (Figure 13–5).

Figure 13–5. 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 supported device families):

- **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 the Arria, Cyclone, and Stratix series of devices.
- **Minimize Area**—Aggressively packs registers to reduce area, even at the cost of design performance.
- **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, Cyclone, MAX II, and Stratix series of devices.
- **Normal**—Packs registers when this is not expected to adversely affect timing results.
- **Off**—Does not pack registers.

- **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 series, Cyclone II, Cyclone III, Stratix II, Stratix III, Stratix IV, and Stratix V 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 series, Cyclone II, Cyclone III, Stratix II, Stratix III, and Stratix IV, and Stratix V devices.

For more information, refer to *Auto Packed Registers logic option* in Quartus Help.

Remove Fitter Constraints

A design with conflicting constraints or constraints that are difficult to meet may 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** task in the Chip Planner 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 or the Chip Planner. To remove LogicLock assignments in the Chip Planner, 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** task in the Chip Planner, refer to *Analyzing and Optimizing the Design Floorplan* in volume 2 of the *Quartus II Handbook*.

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 may 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, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select A**nalysis & Synthesis Settings**.
3. Click **More Settings**. The **More Analysis & Synthesis Settings** page appears.
4. Turn on the **Minimal Bits** setting for the **State Machine Processing** option.

You can also 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 may require a certain type of memory the device does not have. For example, a design that requires two M-RAM blocks cannot 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.

For more information, refer to *Auto RAM Replacement logic option*, *Auto ROM Replacement logic option*, and *Auto Shift Register Replacement logic option* in Quartus II Help.

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. 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.

The following physical synthesis optimizations for fitting are available:

- Physical synthesis for combinational logic
- Map logic into memory

To apply physical synthesis options, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, expand **Compilation Process Settings** and select **Physical Synthesis Optimization**. The **Physical Synthesis Optimization** page appears.
3. 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, perform the following steps:

1. Indicate the module instance you want to apply the setting to in the **To** tab.
2. Select the required physical synthesis assignment in the Assignment **Name** tab.
3. In the **Value** tab, select **ON**.
4. 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 entire project. On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, click **More 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 *Auto DSP Block Replacement* and *DSP Block Balancing* in 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 Multiplier and DSP Functions from HDL Code" 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*.

For additional HDL coding examples, refer to *AN 584: Timing Closure Methodology for Advanced FPGAs.*

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 Packed Registers to Sparse or Sparse Auto

This option is useful for reducing LE or ALM count in a design. This option is available for all Altera devices supported by the Quartus II software.

This option can be set in the Assignment Editor, or you can set this option by clicking **More Settings** on the **Fitter Settings** page in the **Settings** dialog box

For more information, refer to *Auto Packed Registers* in Quartus II Help.

Set Fitter Aggressive Routability Optimizations to Always

If routing resources are resulting in no-fit errors, use this option to reduce routing wire utilization. Perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Fitter Settings**.
3. Click **More Settings**.
4. 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, this option can save up to 6% wire utilization, but can also reduce performance by up to 4%, depending on the device.

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. To increase the placement effort multiplier, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Fitter Settings**.
3. Click **More Settings**.
4. 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 improve 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 (values up to 3 are generally considered 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 entering the following Tcl command:

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

<*value*> can be any positive, non-zero number.

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** task in the Chip Planner 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 or the Chip Planner. Remove LogicLock assignments in the Chip Planner, 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** task in the Chip Planner, 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 select **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.

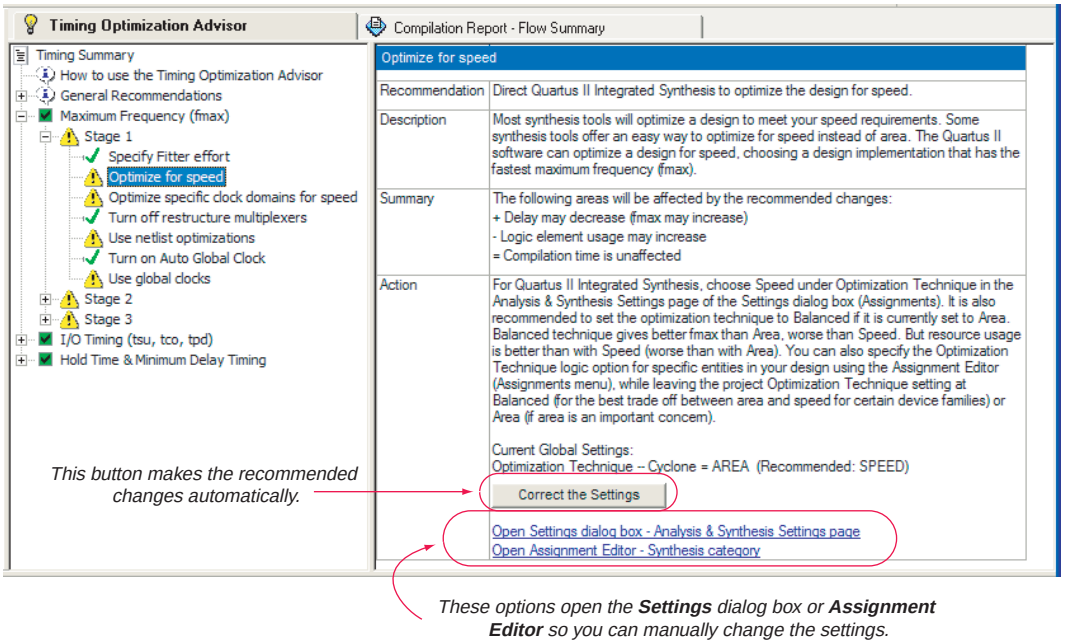
Timing Optimization Advisor

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.

The example in Figure 13–6 shows the Timing Optimization Advisor after compiling a design that meets its frequency requirements, but requires setting changes to improve the timing.

Figure 13–6. 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 13–6, 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.

I/O Timing Optimization

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 13–3, and that the resource utilization is satisfactory before proceeding with I/O timing optimization. The suggestions provided 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 13–34.

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 13–1 shows the recommended order in which to use techniques to reduce t_{SU} and t_{CO} times. Checkmarks indicate which timing parameters are affected by each technique. Reducing t_{SU} times increases hold (t_H) times.

Table 13–1. 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 13–3)	✓	✓
Use timing-driven compilation for I/O (page 13–32)	✓	✓
Use fast input register (page 13–32)	✓	—
Use fast output register, fast output enable register, and fast OCT register (page 13–32)	—	✓
Decrease the value of **Input Delay from Pin to Input Register** or set **Decrease Input Delay to Input Register = ON** (page 13–33)	✓	—
Decrease the value of **Input Delay from Pin to Internal Cells**, or set **Decrease Input Delay to Internal Cells = ON** (page 13–33)	✓	—

Table 13–1. Improving Setup and Clock-to-Output Times *(Note 1)* (Part 2 of 2)

Technique	Affects t_{SU}	Affects t_{CO}
Decrease the value of **Delay from Output Register to Output Pin**, or set **Increase Delay to Output Pin = OFF** (page 13–33)	—	✓
Increase the value of **Input Delay from Dual-Purpose Clock Pin to Fan-Out Destinations** (page 13–33)	✓	—
Use PLLs to shift clock edges (page 13–33)	✓	✓
Use the Fast Regional Clock (page 13–34)	—	✓
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 13–34)	✓	—
Increase the value of **Delay to output enable pin** or set **Increase delay to output enable pin** (page 13–33)	—	✓

Note to Table 13–1:

(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 option 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.

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 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**) can also 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 the 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, Cyclone, MAX II, and Stratix series devices, the Quartus II software automatically adjusts the applicable programmable delays to help meet timing requirements. 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 implemented delay values for 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.

Although the programmable delays in newer devices such as Arria II GX, Stratix III, and Stratix IV 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*.

For details about the programmable delay logic options available for Altera devices, refer to the following Quartus II Help topics.

Decrease Input Delay to Input Register

Input Delay from Pin to Input Register

Decrease Input Delay to Internal Cells

Input Delay from Pin to Internal Cells

Decrease Input Delay to Output Register

Increase Delay to Output Enable Pin

Output Enable Pin Delay

Increase Delay to Output Pin

Delay from Output Register to Output Pin

Increase Input Clock Enable Delay

Input Delay from Dual-Purpose Clock Pin to Fan-Out Destinations

Increase Output Clock Enable Delay

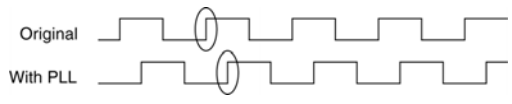
Increase Output Enable Clock Enable Delay

Increase t_{ZX} Delay to Output Pin

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_H at the expense of t_{SU}, while shifting it forward gives a better t_{SU} at the expense of t_H. Refer to Figure 13–7 This technique can be used only in devices that offer PLLs with the phase shift option.

Figure 13–7. Shift Clock Edges Forward to Improve t_{SU} at the Expense of t_H



You can achieve the same type of effect in certain devices by using the programmable delay called **Input Delay from Dual Purpose Clock Pin to Fan-Out Destinations**, described in *Input Delay from Dual-Purpose Clock Pin to Fan-Out Destinations* in Quartus II Help.

Use Fast Regional Clock Networks and Regional Clocks Networks

Altera devices have a variety of hierarchical clock structures. 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 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.

For details, refer to *Guarantee I/O Paths Have Zero Hold Time at Fast Corner logic option* in Quartus II Help.

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.

Coding style affects the performance of your design to a greater extent than other changes in settings. Always evaluate your code and make sure to use synchronous design practices.

For more details about synchronous design practices and coding styles, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

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 design. 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 and correct. For details, refer to "Timing Requirement Settings" on page 13–4.

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 13–9 for details and fix any of these problems before proceeding with optimization.
3. Apply netlist synthesis optimization options.

   Apply the following synthesis options to optimize for speed:

   - "Optimize Synthesis for Speed, Not Area" (page 13–38)
   - "Flatten the Hierarchy During Synthesis" (page 13–39)
   - "Set the Synthesis Effort to High" (page 13–40)
   - "Change State Machine Encoding" (page 13–40)
   - "Prevent Shift Register Inference"(page 13–40)
   - "Use Other Synthesis Options Available in Your Synthesis Tool" (page 13–41)
4. Apply the following options for physical synthesis optimization:
   - Perform physical synthesis for combinational logic
   - Perform automatic asynchronous signal pipelining
   - Perform register duplication
   - Perform register retiming
   - Perform logic to memory mapping
5. Try 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.
6. Make LogicLock assignments (page 13–43) 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 13–42).
8. Make location assignments, or as a last resort, perform manual placement by back-annotating the design (page 13–45).

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 *About Design Space Explorer* in Quartus II Help.

If these techniques do not achieve performance requirements, additional design source code modifications might be required (page 13–42).

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 and effort level.

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, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Analysis & Synthesis Settings**.
3. 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, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Analysis & Synthesis Settings**.
3. Turn on **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**, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Fitter Settings**.
3. 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).

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.

The Quartus II software generally does not retime register paths that cross clock domains. However, if you are using the Classic Timing Analyzer and have a universal f_{MAX} specified for your compilation, the Quartus II software considers all clocks as related to each other, and might retime paths between clock domains. To avoid the retiming of paths, specify individual f_{MAX} requirements for each of the clock domains in your design when using the Classic Timing Analyzer.

You can perform physical synthesis during the fitting 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 the 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.

In many designs, using WYSIWIG primitive resynthesis can reduce area or improve f_{MAX}. By using other physical synthesis options for combinational logic and registers, you might be able to achieve an additional increase in f_{MAX}.

Compilation time might increase considerably when you use high physical synthesis effort levels. The optimizations are design dependent, and some designs might not improve much with physical synthesis.

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

Perform the following steps:

1. In the Assignment Editor, indicate the module instance you want to apply to the specific physical synthesis setting in the **To** tab.
2. Select the required physical synthesis assignment in the **Assignment Name** tab.
3. In the **Value** tab, select **ON**.
4. In the **Enabled** tab, select **Yes**.

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**.

For more information, refer to *PowerPlay power optimization (Analysis & Synthesis Settings Page) logic option* in Quartus II Help.

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, perform the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Analysis & Synthesis Settings**.
3. 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 in the Assignment Editor 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.

For more information about the Design Space Explorer, refer to *About Design Space Explorer* in Quartus II Help.

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.

For more information, refer to *State Machine Processing logic option* in Quartus II Help.

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.

For more information, refer to *Manual Logic Duplication logic option* in 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. Each seed value results in a somewhat different fit, and 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 cannot 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 your design with different seeds using the Design Space Explorer (DSE seed sweep), refer to *About Design Space Explorer* in Quartus II Help.

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**.

For more information, refer to *Router Timing Optimization Level logic option* in Quartus II Help.

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, Cyclone III, Stratix III, and Stratix IV devices. This is called beneficial skew optimization and is enabled by default for devices that support this feature.

For more information, refer to *Enable Beneficial Skew Optimization logic option* in Quartus II Help.

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*.

For additional HDL coding examples. refer to *AN 584: Timing Closure Methodology* for Advanced FPGAs.

LogicLock Assignments

Using LogicLock assignments to improve timing performance is only recommended for older Altera devices, such as the MAX II family. For other device families, especially for larger devices such as the Arria and Stratix series of devices, using LogicLock assignments to improve timing performance is not recommended. For these devices, the LogicLock feature is intended to be used for performance preservation and to floorplan your design.

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. The register-to-register timing paths in the Timing Analyzer section of the Compilation Report help 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 13–8, which has failing paths in the timing analysis results similar to those shown in Table 13–2, `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 13–8. Design Hierarchy

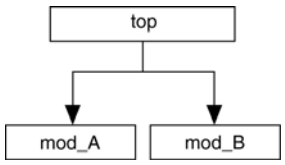


Table 13–2 shows the failing paths connecting two regions together within `mod_A` listed in the timing analysis report.

Table 13–2. 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 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 13–9 and Table 13–3, 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 can also 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 13–9. Failing Paths in Timing Analysis

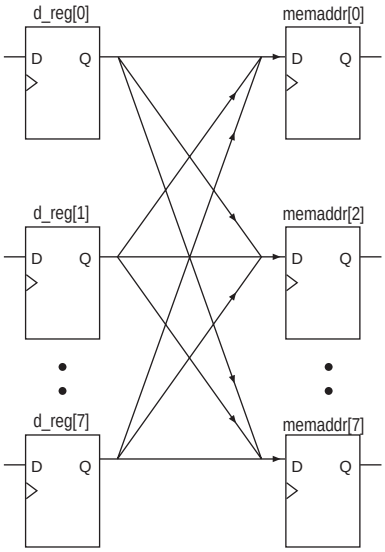


Table 13–3 shows the failing paths listed in the timing analysis report.

Table 13–3. 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 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 Arria and Stratix 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.

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.

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 volume 1 of 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.

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 devices 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 13–3. 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, select 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, click **More Settings**, and in the **Existing Option settings** section, select **Auto Global 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 select 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 *Pin Advisor Command* in 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

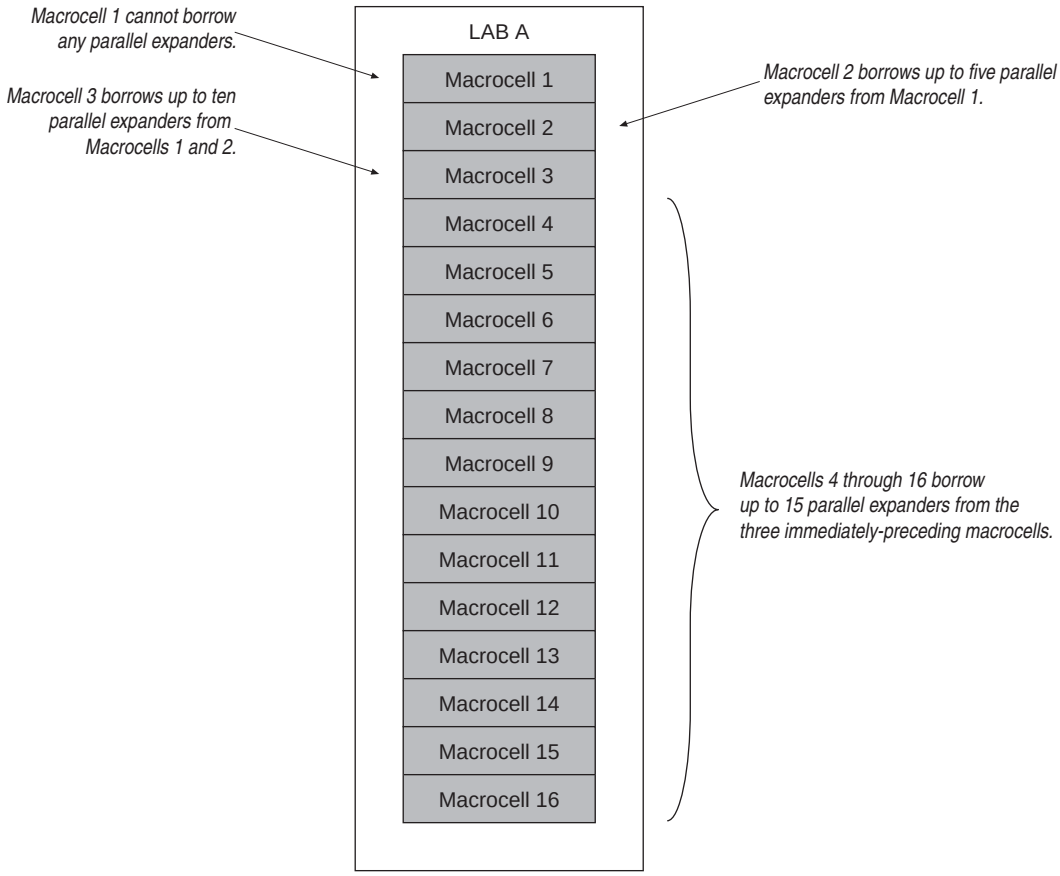
Macrocells with high fan-in can cause more placement problems for the Quartus II Fitter than those with low fan-in. The maximum fan-in per LAB should not exceed 36 in MAX 7000 and MAX 3000 devices. Therefore, estimate the fan-in of logic (such as an *x*-input AND gate) that feeds each output pin. If 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 13–10 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 13–10, 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 13–10. 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**, click **More 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 13–49.
- 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**, click **More Options**, 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 13–47.

Using LCELL Buffers to Reduce Required Resources

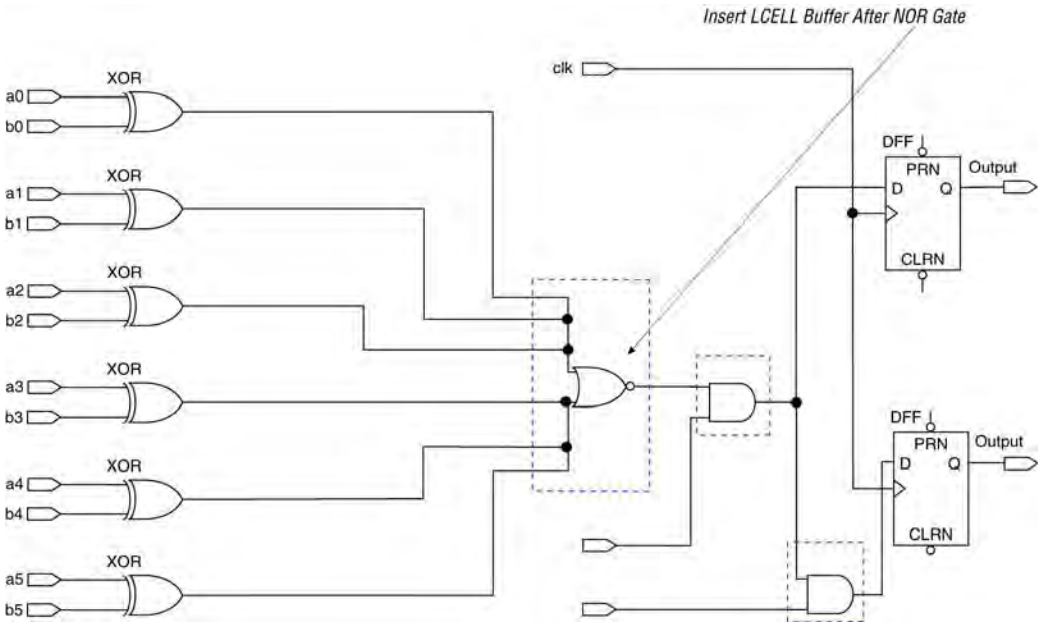
Complex logic, such as multilevel XOR gates, are often implemented with more than one macrocell. When this occurs, the Quartus II software automatically allocates shareable expanders—or additional macrocells (called synthesized logic cells)—to supplement the logic resources that are available in a single 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 13–11). The design in Figure 13–11 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 13–11. Reducing the Average Fan-In by Inserting LCELL Buffers



Timing Optimization Techniques (Macrocell-Based CPLDs)

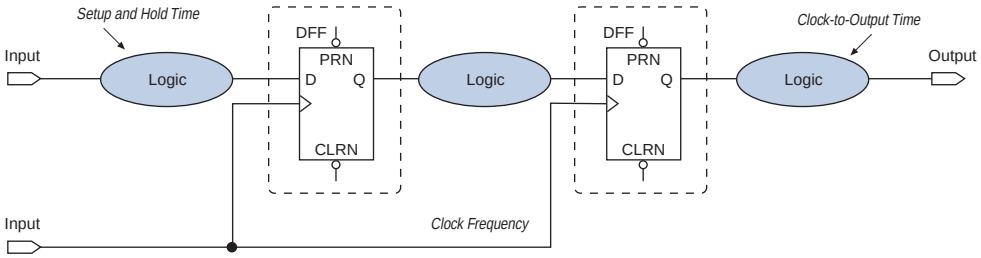
After resource optimization, design optimization focuses on timing. Ensure that you have made the appropriate assignments as described in "Initial Compilation: Required Settings" on page 13–3, and that the resource utilization is satisfactory before proceeding with timing optimization.

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 13–12 shows the parts of the design that determine the t_{SU}, t_H, t_{CO}, t_{PD}, and f_{MAX} timing parameters.

Figure 13–12. 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 13–9.

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 13–4 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 13–4. 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 13–49.

- 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 increase your design's clock speed (f_{MAX}).

- On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**, click **More 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 13–49.
- 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, select **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 additional register stages (pipeline registers) consumes more device resources, but it also lowers the propagation delay between registers, allowing you to maintain high system clock speed.

Refer to the application note *AN 584: Timing Closure Methodology for Advanced FPGA Designs* for more information about pipelining registers and other examples of optimizing source code.

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 Design Space Explorer (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 *About Design Space Explorer* in Quartus II Help.

Other Optimization Advisors

The Power Optimization Advisor provides guidance for reducing power consumption. In addition, the Incremental Compilation Advisor provides suggestions to improve your 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*. For 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 ↵
```

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.

Table 13–5 lists the **.qsf** file variable name and applicable values for the settings discussed in "Initial Compilation: Required Settings" on page 13–3. Table 13–6 shows the list of advanced compilation settings.

Table 13–5. 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 13–6. 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 13–7 lists the **.qsf** file variable name and applicable values for the settings discussed in "Resource Utilization Optimization Techniques (LUT-Based Devices)" on page 13–16.

Table 13–7. 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 13–7. 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
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 13–8 lists the **.qsf** file variable name and applicable values for the I/O timing optimization settings.

Table 13–8. 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 13–9 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 13–34.

Table 13–9. 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 13–1.

Example 13–1. 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 links 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.

Document Revision History

Table 13–10 shows the revision history for this chapter.

Table 13–10. Document Revision History (Part 1 of 2)

Date	Version	Changes Made
August 2010	10.0.1	■ Corrected link
July 2010	10.0.0	■ Moved Compilation Time Optimization Techniques section to new *Reducing Compilation Time* chapter ■ Removed references to Timing Closure Floorplan ■ 'Moved Smart Compilation Setting and Early Timing Estimation sections to new *Reducing Compilation Time* chapter ■ Added Other Optimization Resources section ■ Removed outdated information ■ Changed references to DSE chapter to Help links ■ Linked to Help where appropriate ■ Removed Referenced Documents section
November 2009	9.1.0	■ Removed unsupported Timing Closure Floorplan references ■ Removed references to unsupported device families ■ Added several notes ■ Minor text edits
March 2009	9.0.0	■ 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

Table 13–10. Document Revision History (Part 2 of 2)

Date	Version	Changes Made
November 2008	8.1.0	■ 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.
May 2008	8.0.0	■ Updated links ■ Updated the following sections: →Other Optimization Resources] →Setting Process Priority →Location Assignment and Back-Annotation →Fitter Effort Setting →Synthesis Netlist Optimizations and Physical Synthesis Optimizations →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

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

Take an online survey to provide feedback about this handbook chapter.

14. Power Optimization

QII52016-10.0.0

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. Arria series, Stratix II, Stratix III, Stratix IV, and Stratix V device families include 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 a device-specific architecture, refer to the device handbook, available from the Literature and Technical Documentation page on the Altera website.

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 III and Stratix IV devices. These techniques include:

- Selectable Core Voltage (available only for Stratix III devices)
- Programmable Power Technology
- Device Speed Grade Selection

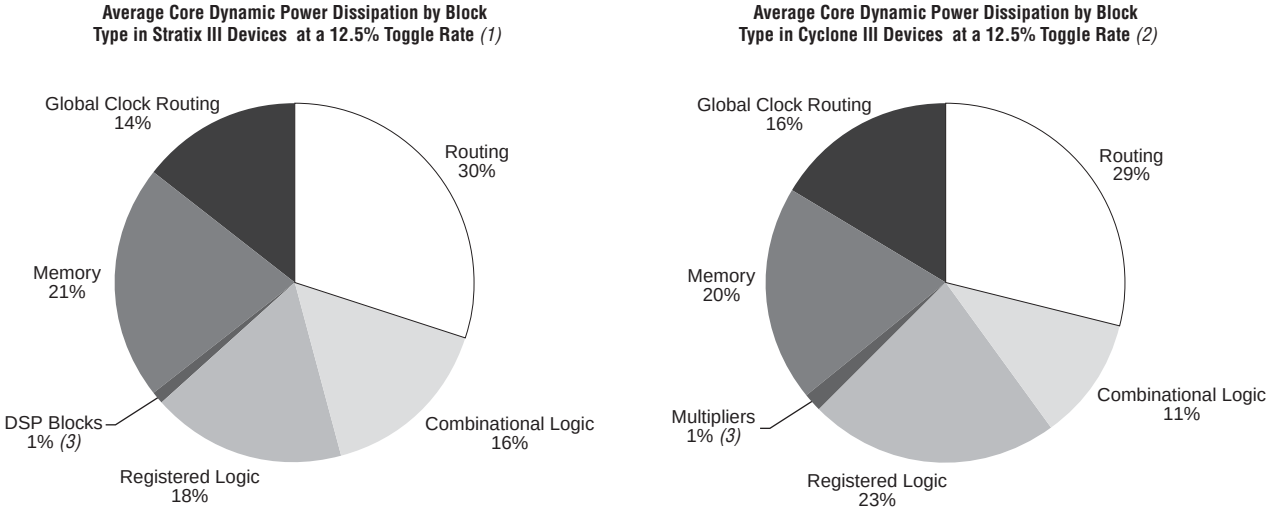
For more information about power optimization techniques available for Stratix III devices, refer to *AN 437: Power Optimization in Stratix III FPGAs*. For more information about power optimization techniques available for Stratix IV devices, refer to *AN 514: Power Optimization in Stratix IV FPGAs*.

Power Dissipation

This section describes the sources of power dissipation in Stratix III and Cyclone III devices. You can refine techniques that reduce power consumption in your design by understanding the sources of power dissipation.

Figure 14–1 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 14–1. Average Core Dynamic Power Dissipation



Notes to Figure 14–1:

(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 14–1, 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 II, Cyclone III and Cyclone IV GX devices, it is the logic element (LE).

For more information about ALMs and LEs in Cyclone II, Cyclone III, Cyclone IV GX, Stratix II, Stratix III, Stratix IV, and Stratix V, devices, refer to the respective device handbook.

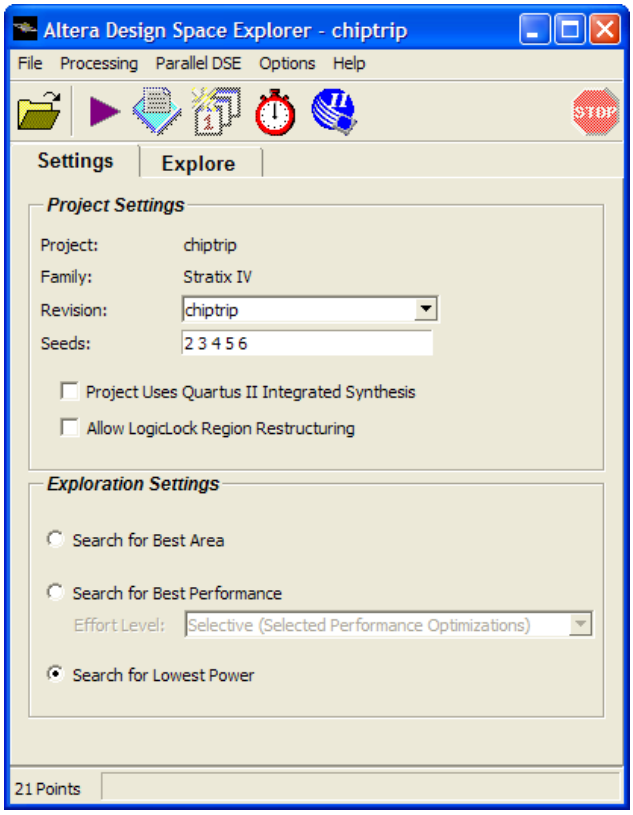
Memory and clock resources are other major consumers of power in FPGAs. Stratix II devices feature the TriMatrix memory architecture. TriMatrix memory includes 512-bit M512 blocks, 4-Kbit M4K blocks, and 512-Kbit M-RAM blocks, which are configurable to support many features. Stratix IV and Stratix III 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. Stratix III, Stratix IV, and Stratix V devices feature Programmable Power Technology, an advanced architecture that enables a smooth trade-off between speed and power. The core of each Stratix III, Stratix IV, and Stratix V device is divided into tiles, each of which may be put into a high-speed or low-power mode. The primary benefit of Programmable Power Technology is to reduce static power, with a secondary benefit being a small reduction in dynamic power. Cyclone II devices have 4-Kbit M4K memory blocks, and Cyclone III and Cyclone IV GX devices have 9-Kbit M9K memory blocks.

Design Space Explorer

Design Space Explorer (DSE) is a simple, easy-to-use, design optimization utility that is included in the Quartus II software. 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 DSE to implement the techniques described in this chapter.

Figure 14–2 shows the DSE user interface. The **Settings** tab is divided into **Project Settings** and **Exploration Settings**.

Figure 14–2. 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.

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 *About Design Space Explorer* in Quartus II Help.

Power-Driven Compilation

The standard Quartus II compilation flow consists of Analysis and Synthesis, placement and routing, Assembly, and Timing Analysis. Power-driven compilation takes place at the Analysis and Synthesis and Place-and-Route stages.
Quartus II software settings that control power-driven compilation are located in the **PowerPlay power optimization** list on the **Analysis & Synthesis Settings** page, and the **PowerPlay power optimization** list on the **Fitter Settings** page. The following sections 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 all devices supported by the Quartus II software except MAX® 3000 and MAX 7000 devices. (Figure 14–3).

Figure 14–3. Analysis & Synthesis Settings Page

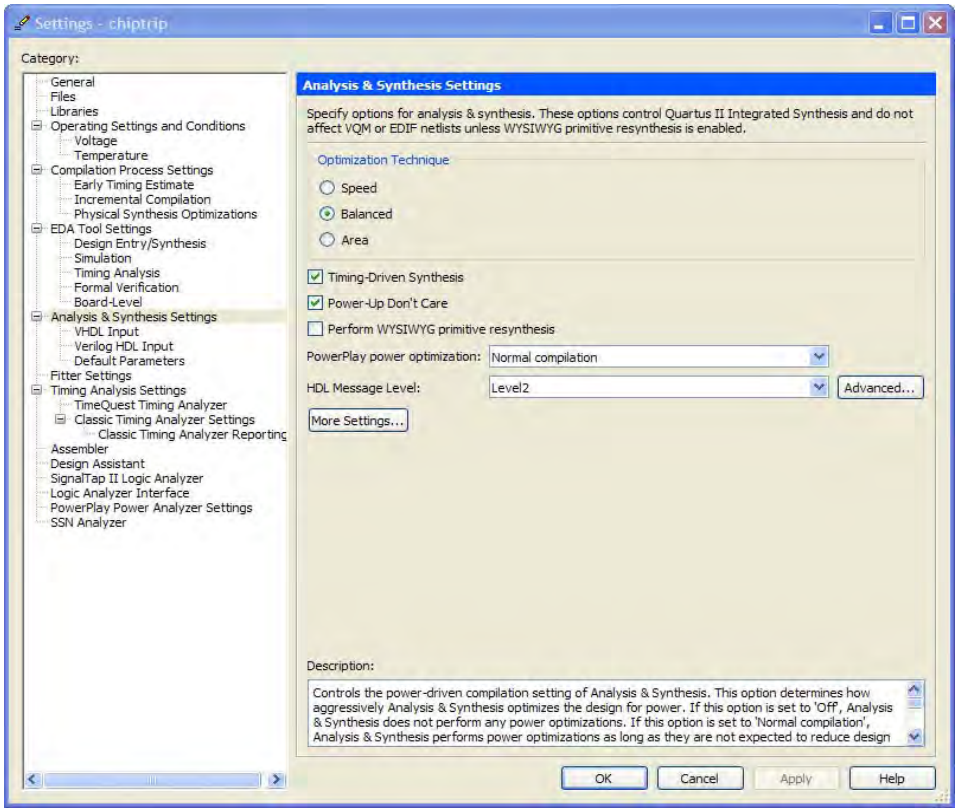


Table 14–1 shows the settings in the **PowerPlay power optimization** list. You can apply these settings on a project or entity level.

Table 14–1. 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 14–13. 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 14–4).

Figure 14–4. Memory Transformation

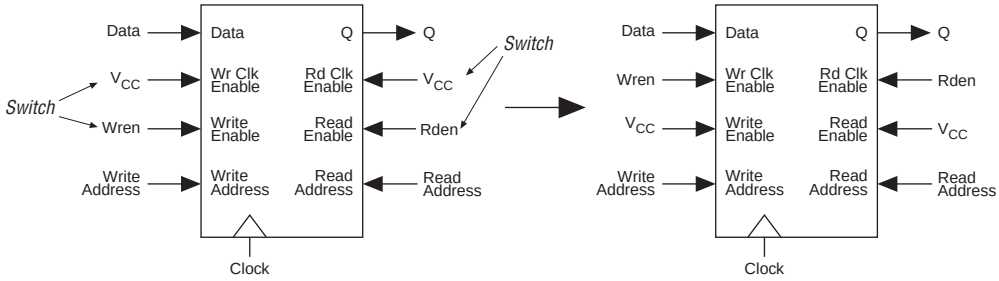


Figure 14–4 shows a default implementation of a simple dual-port memory block in which write-clock enable signals and read-clock enable signals are connected to V_{CC}, making both read and write memory ports active during each clock cycle. Memory transformation effectively moves the read-enable and 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 14–13. For Stratix III, Stratix IV, and Stratix V devices, the memory transformation takes place at the Fitter level by selecting the **Normal compilation** settings for the power optimization option.

In Stratix III, Cyclone III, Cyclone IV GX, and Stratix 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.

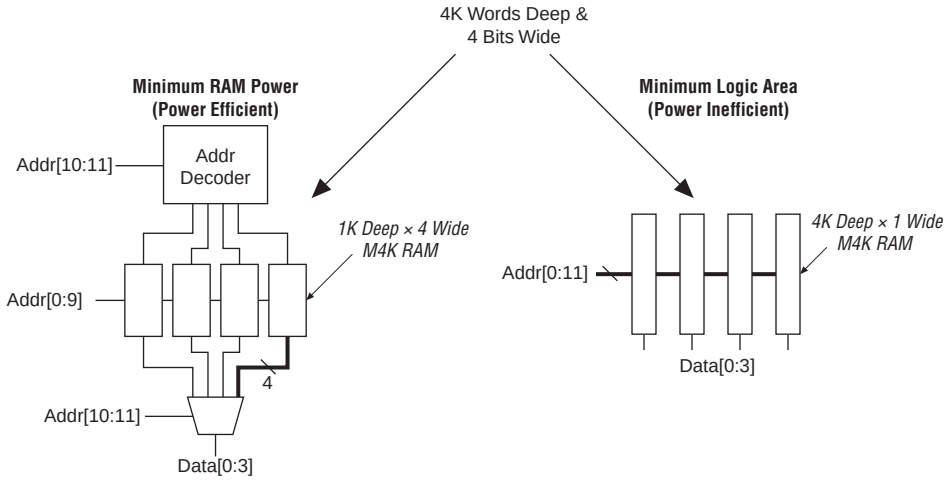
The **Extra effort** setting performs the functions of the **Normal compilation** setting and other memory optimizations to further reduce memory power by shutting down memory blocks that are not accessed. This level of memory optimization can require extra logic, which can reduce design performance.

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 14–5 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 14–5. 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 14–13. The MAXIMUM_DEPTH parameter always takes precedence over the **Optimize Power for Synthesis** options for power optimization on memory optimization.

For step-by-step instructions on how to perform power-driven synthesis, refer to *Running a Power-Optimized Compilation* in Quartus II Help.

Power-Driven Fitter

The **Fitter Settings** page enables you to specify options for fitting (Figure 14–6). The **PowerPlay power optimization** option is available for Arria GX, Arria II GX, Cyclone II, Cyclone III, Cyclone IV, HardCopy series, Stratix II, Stratix II GX, Stratix III, Stratix IV, and Stratix V devices.

Figure 14–6. Fitter Settings Page

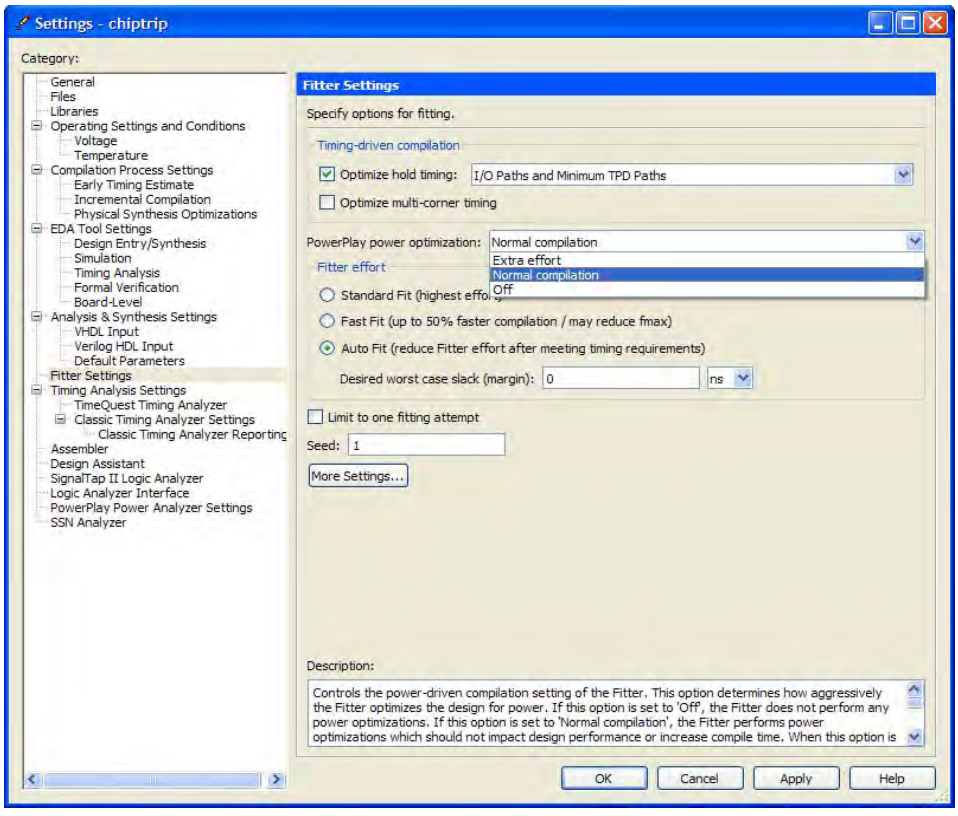


Table 14–2 lists the settings in the **PowerPlay power optimization** list. These settings can only be applied on a project-wide basis. The **Extra effort** setting for the Fitter requires extensive effort to optimize the design for power and can increase the compilation time.

Table 14–2. 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, Stratix IV, and Stratix V 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 14–4.

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 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 **.vcd** file, 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. 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.

Only the **Extra effort** setting in the **PowerPlay power optimization** list for the Fitter option uses the signal activities (from **.vcd** files) 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 **.vcd** files and how to create them, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

For step-by-step instructions on how to perform power-driven fitting, refer to *Running a Power-Optimized Compilation* in Quartus II Help.

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.

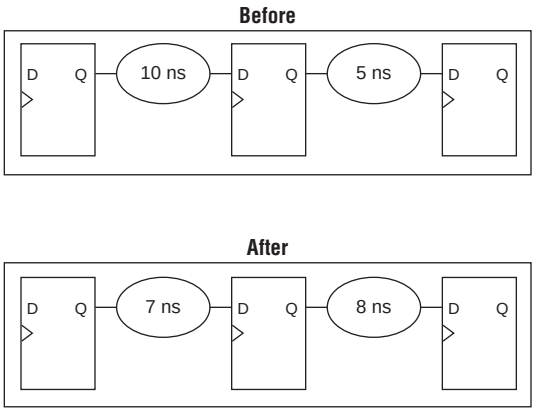
The benchmark data shows that the area-driven technique can reduce power consumption by as much as 31% in Stratix devices and as much as 15% in Cyclone devices.

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 noncritical timing paths.

Retiming uses fewer registers than pipelining. Figure 14–7 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 14–7. Gate-Level Register Retiming



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 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 Cyclone II, Cyclone III, Cyclone IV GX, Stratix II, and Stratix III devices that affect overall design power. The results of these techniques might be different from design to design.

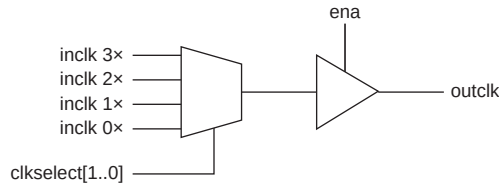
Clock Power Management

Clocks represent a significant portion of dynamic power consumption due to their high switching activity and long paths. Figure 14–1 on page 14–2 shows a 14% average contribution to power consumption for global clock routing in Stratix III devices and 16% 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 enables only 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, Arria II GX, Cyclone III, Cyclone IV, Stratix II, Stratix III, Stratix IV, and Stratix V 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, Arria II GX, Cyclone III, Cyclone IV, Stratix II, Stratix III, Stratix IV, and Stratix V devices provide clock control blocks for global clock networks. In addition, Stratix II, Stratix III, Stratix IV, and Stratix V 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 14–8 shows a 4-input clock control block diagram.

Figure 14–8. Clock Control Block Diagram

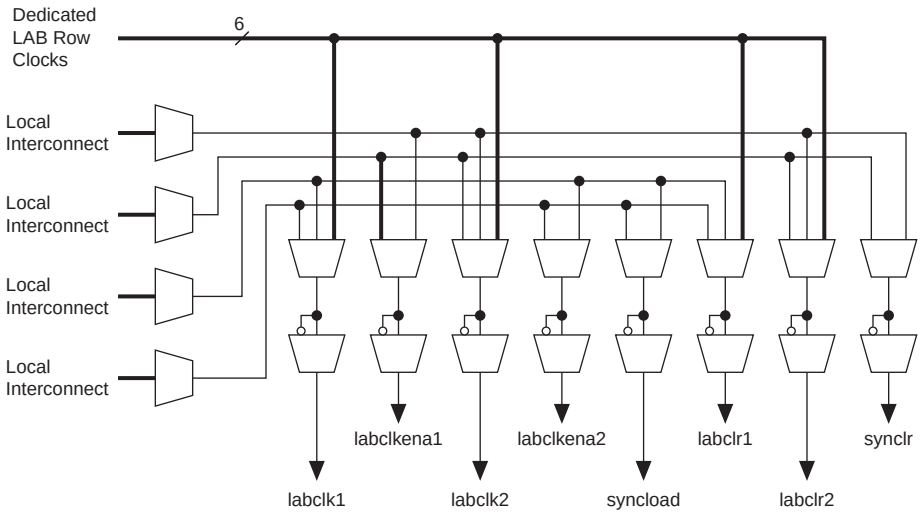


The enable signal is applied to the clock signal before being distributed to global routing. Therefore, the enable signal can either have a significant timing slack (at least as large as the global routing delay) or it can reduce the f_{MAX} of the clock signal.

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 devices, each LAB can use two clocks and two clock enable signals, as shown in Figure 14–9. 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 14–9. 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 14–1 makes use of a LAB-wide clock enable. This clock-gating logic is automatically turned into an LAB-level clock enable signal.

Example 14–1.

```
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.

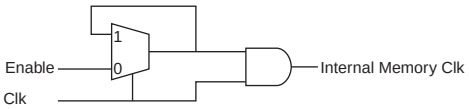
Reducing Memory Power Consumption

The memory blocks in FPGA devices can represent a large fraction of typical core dynamic power. Memory consumes approximately 20% of the core dynamic power in typical Cyclone III and Stratix III device designs. 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.

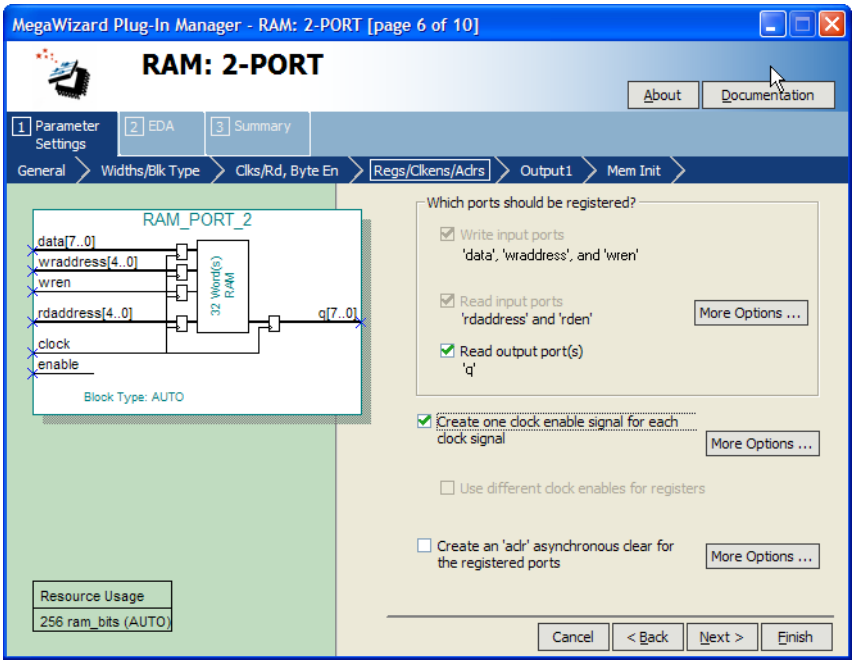
The key to reducing memory power consumption is to reduce the number of memory clocking events. You can achieve this through clock network-wide gating described in "Clock Power Management" on page 14–11, or on a per-memory basis through use of the clock enable signals on the memory ports. Figure 14–10 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 14–10. 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 14–11).

Figure 14–11. MegaWizard Plug-In Manager RAM 2-Port Clock Enable Signal Selectable Option

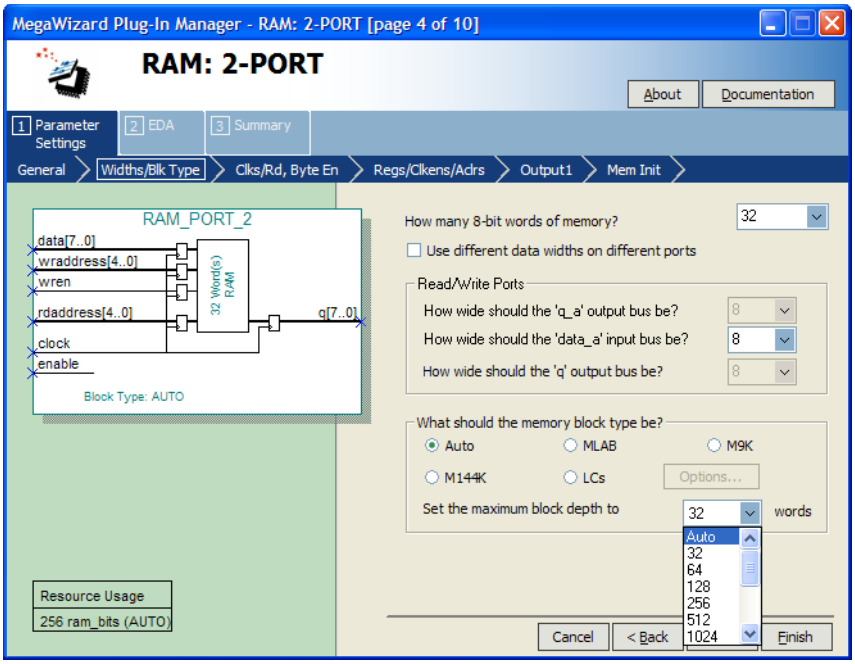


For example, consider a design that contains a 32-bit-wide M4K memory block in ROM mode that is running at 200 MHz. Assuming that the output of this block is only required approximately every four cycles, this memory block will consume 8.45 mW of dynamic power according to the demands of the downstream 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 Cyclone II, Cyclone III, Cyclone IV GX, Stratix II, Stratix III, Stratix IV, and Stratix V 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 14–12). The Quartus II software automatically chooses the best design memory configuration for optimal power, as described in "Power-Driven Compilation" on page 14–4.

Figure 14–12. MegaWizard Plug-In Manager RAM 2-Port Maximum Depth Selectable Option



Memory Power Reduction Example

Table 14–3 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 14–3. 4K × 36 Simple Dual-Port Memory Implemented Using Multiple M4K Blocks (Part 1 of 2)

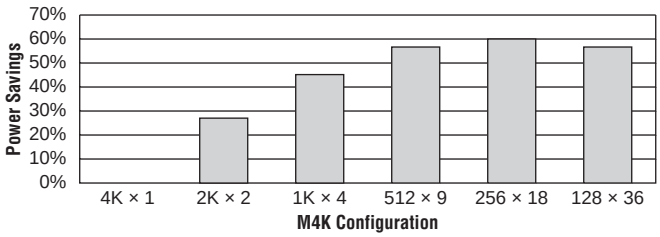
M4K Configuration	Number of M4K Blocks	ALUTs
4K × 1 (Default setting)	36	0
2K × 2	36	40
1K × 4	36	62

Table 14–3. 4K × 36 Simple Dual-Port Memory Implemented Using Multiple M4K Blocks (Part 2 of 2)

M4K Configuration	Number of M4K Blocks	ALUTs
512 × 9	32	143
256 × 18	32	302
128 × 36	32	633

Figure 14–13 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 14–13. Power Savings Using the MAXIMUM_DEPTH Parameter



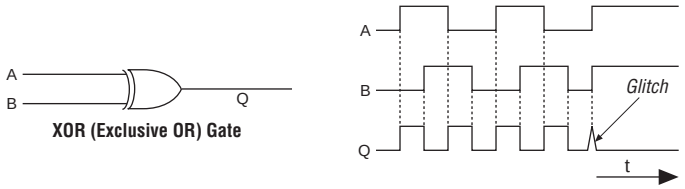
As the memory depth becomes more shallow, memory dynamic power decreases because unaddressed M4K blocks can be shut off using a decoded combination of address bits and the read enable signal. For a 128-deep memory block, power used by the extra LEs starts to outweigh the power gain achieved by using 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.

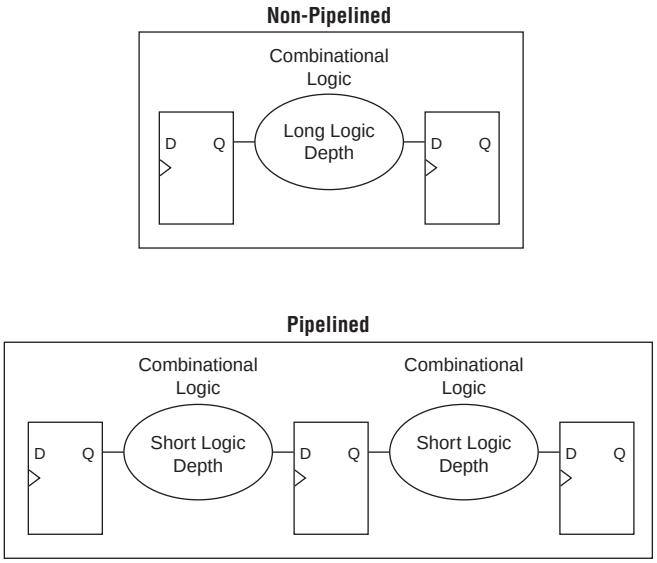
For example, consider an input change on one input of a 2-input XOR gate from `1` to `0`, followed a few moments later by an input change from `0` to `1` on the other input. For a moment, both inputs become `1` (high) during the state transition, resulting in `0` (low) at the output of the XOR gate. 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 14–14. 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 14–14. 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 14–15 shows an example where pipelining is applied to break up a long combinational path.

Figure 14–15. 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 30% in Cyclone and Stratix 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

Nonterminated 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 14–1:

Equation 14–1. Capacitive loading at the output pin

$$P = 0.5 \times F \times C \times V^2$$

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.

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 nonterminated 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 14–4 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.

For this configuration, nonterminated 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 nonterminated outputs increases faster than the power of terminated outputs.

Table 14–4. I/O Power for Different I/O Standards in Stratix II Devices

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
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 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*, the *Stratix III Device I/O Features* chapter in volume 1 of the *Stratix III Device Handbook*, the *I/O Features in Stratix IV Devices* in volume 1 of the *Stratix IV Device Handbook*, or the *Selectable I/O Standards in Cyclone II Devices* chapter in the *Cyclone II Device Handbook*, the *Cyclone III Device Handbook*, or the *Cyclone IV GX 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 devices, 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 Cyclone III, Cyclone IV GX, Stratix III, Stratix IV, and Stratix V, devices, Advanced I/O Timing, which uses the full board trace model, is always used.

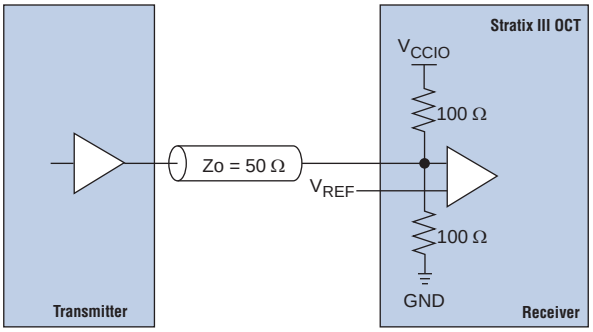
For information about using Advanced I/O Timing and configuring a board trace model, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

Dynamically Controlled On-Chip Terminations

Stratix V, 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 V, 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 14–16).

Figure 14–16. 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.

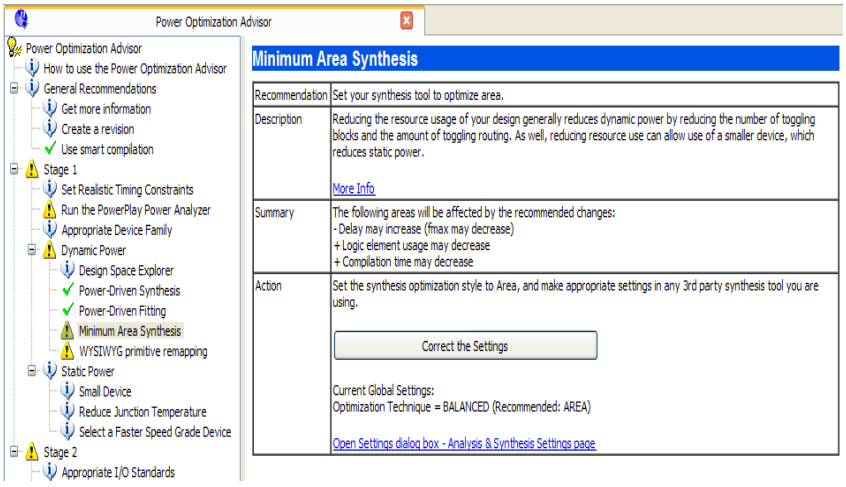
Power Optimization Advisor

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 14–17 shows the Power Optimization Advisor after compiling a design that is not fully optimized for power.

Figure 14–17. Power Optimization Advisor

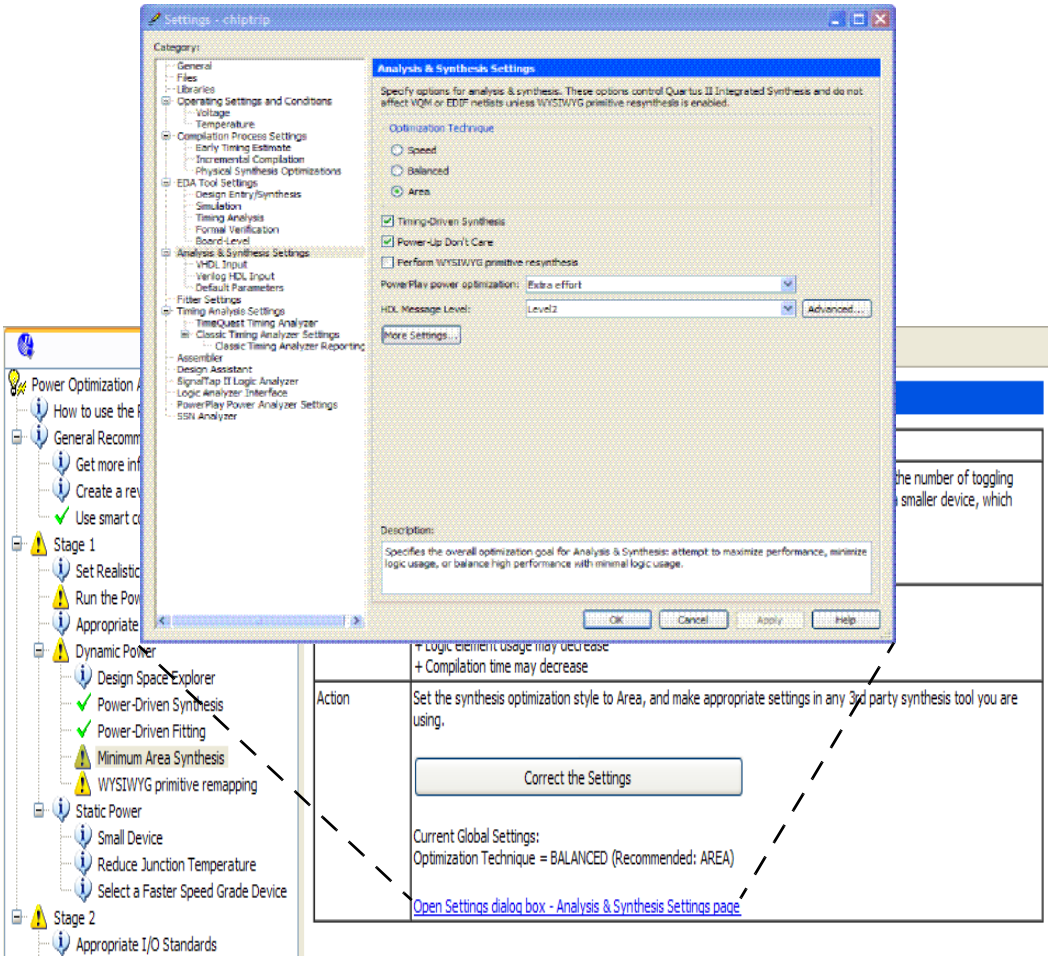


The Power Optimization Advisor shows the recommendations that can reduce power in your design. The recommendations are split into stages to show the order in which you should apply the recommended settings. The first stage shows mostly CAD setting options that are easy to implement and highly effective in reducing design power. An icon indicates whether each recommended setting is made in the current

project. In Figure 14–17, the checkmark icons 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 effect of the recommendation, and the action required to make the appropriate setting.

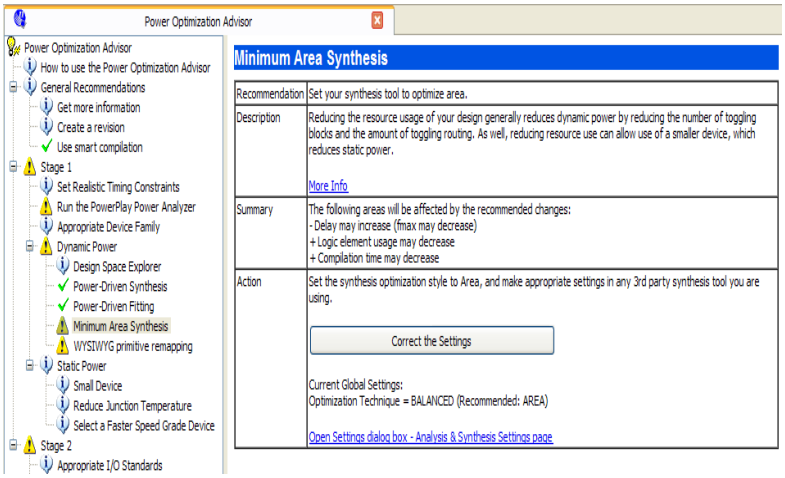
There is a link from each recommendation to the appropriate location in the Quartus II user interface where you can change the setting. You can change the **Power-Driven Synthesis** setting by clicking **Open Settings dialog box - Analysis & Synthesis Settings page** (Figure 14–18). The **Settings** dialog box is shown with the **Analysis & Synthesis Settings** page selected, where you can change the **PowerPlay power optimization** settings.

Figure 14–18. Analysis & Synthesis Settings Page



After making the recommended changes, recompile your design. The Power Optimization Advisor indicates with green check marks that the recommendations were implemented successfully (Figure 14–19). You can use the PowerPlay Power Analyzer to verify your design power results.

Figure 14–19. 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.

Document Revision History

Table 14–5 shows the revision history for this chapter.

Table 14–5. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Was chapter 11 in the 9.1.0 release ■ Updated Figures 14-2, 14-3, 14-6, 14-18, 14-19, and 14-20 ■ Updated device support ■ Minor editorial updates
November 2009	9.1.0	■ Updated Figure 11-1 and associated references ■ Updated device support ■ Minor editorial update

Table 14–5. Document Revision History

Date	Version	Changes Made
March 2009	9.0.0	■ Was chapter 9 in the 8.1.0 release ■ Updated for the Quartus II software release ■ Added benchmark results ■ Removed several sections ■ Updated Figure 14–1, Figure 14–17, Figure 14–18, and Figure 14–19
November 2008	8.1.0	■ Changed to 8½" × 11" page size ■ Changed references to altsyncram to RAM ■ Minor editorial updates
May 2008	8.0.0	■ Added support for Stratix IV devices ■ 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

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

Take an online survey to provide feedback about this handbook chapter.

15. Analyzing and Optimizing the Design Floorplan

QII52006-10.0.0

As FPGA designs grow larger in density, the ability to analyze the design for performance, routing congestion, and logic placement to meet the design requirements becomes critical. This chapter discusses how to analyze the design floorplan with the Chip Planner.

You can perform design analysis and create and optimize the design floorplan with the Chip Planner. To make I/O assignments, use the Pin Planner.

For information about the Pin Planner, refer to the *I/O Management* chapter in volume 2 of the *Quartus II Handbook*.

You can use the Design Partition Planner with the Chip Planner to customize the floorplan of 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 Partitions and Floorplan Assignments* chapters in volume 1 of the *Quartus II Handbook*.

This chapter includes the following topics:

- "Chip Planner Overview" on page 15–1
- "LogicLock Regions" on page 15–4
- "Using LogicLock Regions in the Chip Planner" on page 15–12
- "Design Floorplan Analysis Using the Chip Planner" on page 15–13
- "Scripting Support" on page 15–26

The Chip Planner supports the Arria, Cyclone, Hardcopy, and Stratix series devices, as well as MAX II and MAX II Z devices.

In the Quartus II software version 10.0, the Chip Planner provides read-only support for Stratix V devices.

Chip Planner Overview

The Chip Planner provides a visual display of chip resources. The Chip Planner can show logic placement, LogicLock regions, relative resource usage, detailed routing information, fan-in and fan-out connections between nodes, timing paths between registers, delay estimates for paths, and routing congestion information.

You can also make assignment changes with the Chip Planner, such as creating and deleting resource assignments, and you can perform post-compilation changes such as creating, moving, and deleting logic cells and I/O atoms. With the Chip Planner, you can view and create assignments for a design floorplan, perform power and design analyses, and implement ECOs. With the Chip Planner and 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.

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

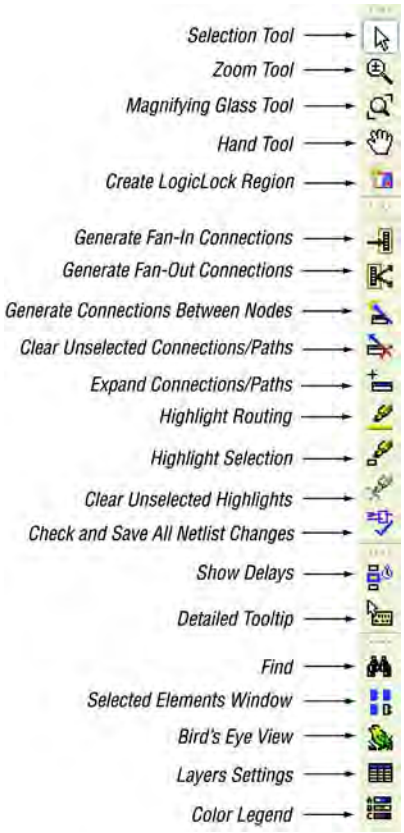
To start the Chip Planner, on the Tools menu, click **Chip Planner (Floorplan & Chip Editor)**. You can also start the Chip Planner by the following methods:

- Click the Chip Planner icon on the Quartus II software toolbar
- On the Shortcut menu in the following tools, click **Locate** and then click **Chip Planner**:
  - 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

Chip Planner Toolbar

The Chip Planner provides powerful tools for design analysis with a GUI. You can access Chip Planner commands from the View menu and the Shortcut menu, or by clicking the icons on the toolbar. Figure 15–1 shows an example of the Chip Planner toolbar and provides descriptions for commonly used icons.

Figure 15–1. Chip Planner Toolbar



Chip Planner Tasks, Layers, and Editing Modes

The Chip Planner models types of resource objects as unique display layers, and uses tasks— which are predefined sets of layer settings—to control the display of resources. The Chip Planner provides a set of default tasks, and you can create custom tasks to customize the display for your particular needs. The Basic, Detailed, and Floorplan Editing tasks provided with the Chip Planner are useful for general ECO and assignment-related activities, while the Partition Planner, Power, and Routing Congestion tasks are optimized for specific activities.

The Chip Planner has two editing modes, which determine the types of operations that you can perform. The Assignment editing mode allows you to make assignment changes that are applied by the Fitter during the next place and route operation. The ECO editing mode allows you to make post-compilation changes, commonly referred to as engineering change orders (ECOs).

You should choose the editing mode appropriate for the work that you want to perform, and a task that displays the resources that you want to view, in a level of detail appropriate for your design.

For more information about tasks, layers, and editing modes in the Chip Planner, refer to *About the Chip Planner* and to *Layers Settings Dialog Box* in Quartus II Help. 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*.

LogicLock Regions

LogicLock regions allow you to constrain resource placement in specified areas on the device. A LogicLock region is defined by its height, width, and location; 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 contents of the region and the timing requirements of the module.

You can use LogicLock regions to create a floorplan for your design. Your floorplan can contain several LogicLock regions. Table 15–1 describes the options for creating LogicLock regions.

Table 15–1. Types of LogicLock Regions

Properties	Values	Behavior
State	Floating (default), Locked	Allows the Quartus II software to determine the location of the region on the device. Locked regions are areas that you define and are shown with a solid boundary in the floorplan. A locked region must have a fixed size.
Size	Auto (default), Fixed	Allows the Quartus II software to determine the appropriate size of a region given its contents. Fixed regions have a shape and size that you define.
Reserved	Off (default), On	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.
Origin	Any Floorplan Location	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 the size of a region 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

You can create LogicLock Regions from the Project Navigator, the LogicLock Regions window, or the Chip Planner.

Creating LogicLock Regions with the Project Navigator

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, right-click on any design entity in the design, and click **Create New LogicLock Region** to create a LogicLock region.

Creating LogicLock Regions with the Chip Planner

To create a LogicLock region in the Chip Planner, click the **Create LogicLock Region** command on the View menu, then click and drag on the Chip Planner floorplan to create a region of your preferred location and size.

Placing LogicLock Regions

A fixed region must contain all resources required for the design block for which you define the region. 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, as follows:

- 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 can overlap with their ancestors or descendants, but not with other floating LogicLock regions.
- 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. (You can determine this from the applicable device handbook.) 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.

Placing Device Resources 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 contents of the LogicLock region.

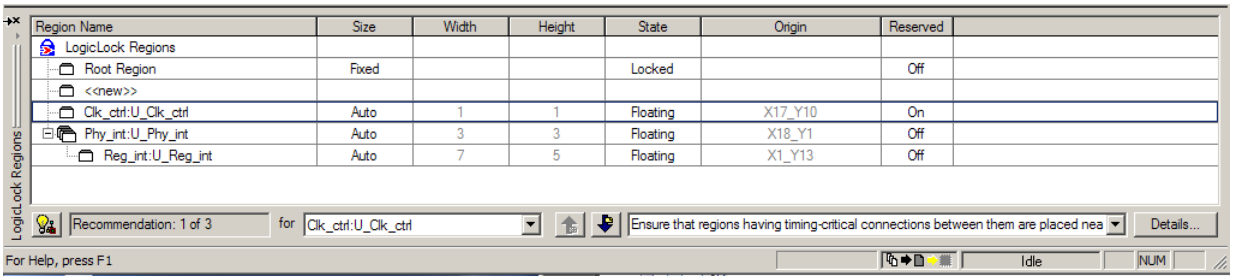
Only one LogicLock region can claim a device resource. If a LogicLock region boundary includes part of a device resource, the Quartus II software allocates the entire resource to that LogicLock region.

Pin assignments to LogicLock regions are effective only in fixed and locked regions. Pin assignments to floating regions do not influence the placement of the region.

LogicLock Regions Window

The LogicLock window consists of the LogicLock Regions window (Figure 15–2) 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 the properties of a LogicLock region such as 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. You can customize the LogicLock Regions window by dragging and dropping the columns to change their order; you can also hide columns.

Figure 15–2. LogicLock Regions Window



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 which resources are required. The **LogicLock Region Properties** dialog box shows the properties of the current selected regions. You can also modify the settings for LogicLock regions in the **LogicLock Region Properties** 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**.

For designs that target Arria GX, Cyclone, Stratix, 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 is locked and fixed.

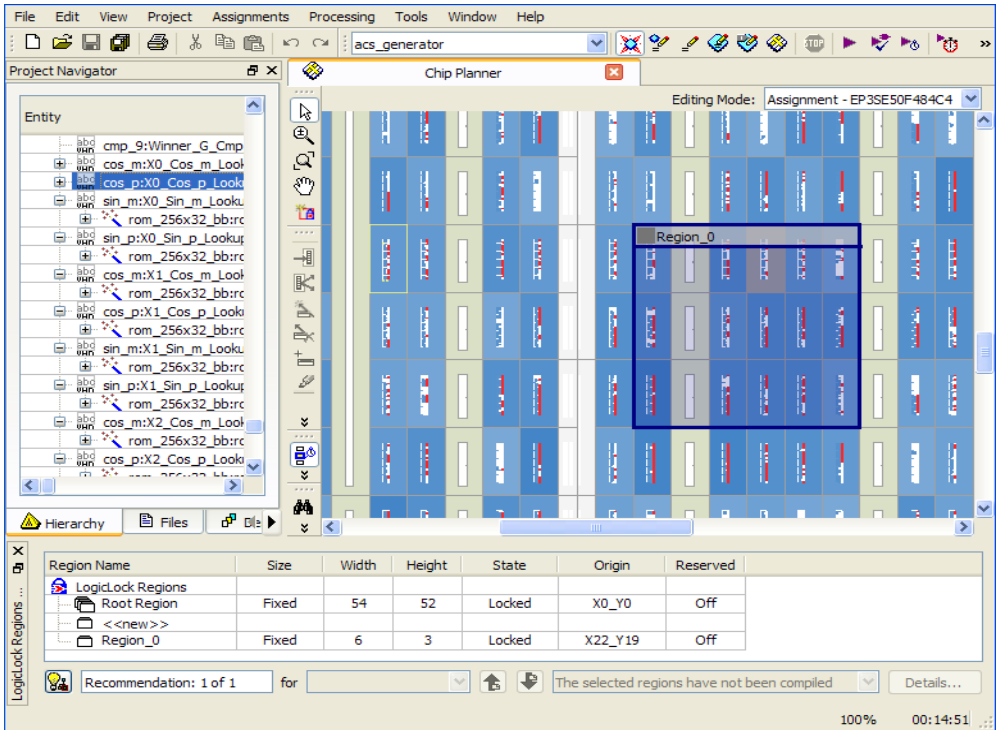
For Arria GX, Cyclone, Stratix, and MAX II devices, the origin of the LogicLock region is located at the lower-left corner of the region. For all other supported devices, the origin is located at the upper-left corner of the region.

Assigning LogicLock Region Content

After you have created a LogicLock region, you must assign resources to the region 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 15–3 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 15–3. Drag and Drop Logic in the Chip Planner



You can also drag logic from the **Hierarchy** tab of the Project Navigator and drop the logic in the LogicLock Regions **Properties** dialog box. You can also drop logic into the **Design Element Assigned** column of the **Contents** tab of the **LogicLock Region Properties** box.

You must manually assign pins to a LogicLock region. 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 Cyclone, Stratix, and MAX II series of devices, the locked regions must include the I/O pins as resources.

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 the performance of a module by constraining the nodes in the critical path of the module.

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 the new child region into 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. The location of a floating child region can float within its parent. If a child region is 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 you change the parent's location, 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. The levels of hierarchy in LogicLock regions are unlimited, but 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 **Reserved** on the **General** tab of the **LogicLock Region Properties** dialog box. When you turn on this option, your LogicLock region contains only the entities and nodes that you specifically assigned to your LogicLock region. When you set the reserved property for a LogicLock region, the Fitter does not place logic from the immediate parent LogicLock region in the assigned LogicLock area, but it might place logic from other parts of your design in that area.

Creating Non-Rectangular LogicLock Regions

When you create a floorplan for your design, you may want to create non-rectangular LogicLock regions to make some device resources accessible to design blocks outside a LogicLock region. You might also create a non-rectangular LogicLock region to place certain parts of your design around specific device resources to improve performance.

To create a non-rectangular region with the **Merge LogicLock Region** command, follow these steps:

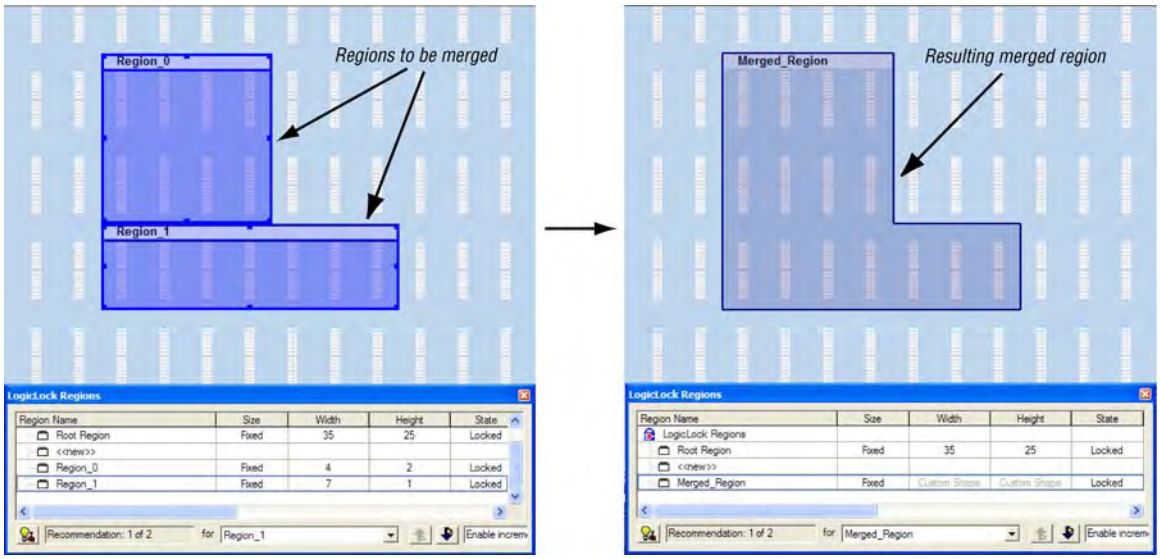
1. In the Chip Planner, create two or more contiguous or non-contiguous rectangular regions as described in "Creating LogicLock Regions" on page 15–5.

2. Arrange the regions that you have created into the locations where you want the non-rectangular region to be.
3. Select all the individual regions that you want to merge by clicking each of them while holding the Shift key.
4. Right-click the title bar of any of the LogicLock regions that you want to merge, point to **LogicLock Regions**, and then click **Merge LogicLock Region**. The individual regions that you select merge to create a single new region.

By default, the new LogicLock region has the same name as the component region containing the greatest number of resources; however, you can rename the new region. In the **LogicLock Regions Window**, the new region is shown as having a custom shape.

Figure 15–4 illustrates two autonomous LogicLock regions combined using the **Merge LogicLock Region** command to form a new non-rectangular region.

Figure 15–4. Using the **Merge** command to create a non-rectangular region



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 you set the region's **Reserved** property to **On**. To indicate to 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** window, the Quartus II software has additional features to help you design with LogicLock regions.

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. You can use this feature when you manually create LogicLock regions.

Quartus II Revisions Feature

When you evaluate different LogicLock regions in your design, you might want to experiment with different configurations to achieve your desired results. The Quartus II Revisions feature allows you to organize the same project with different settings until you find an optimum configuration.

To use the Revisions feature, on the Project menu, click **Revisions**. In the **Revisions** dialog box, you can create and specify revisions. You can base revision on the current design or any previously created revisions. Each revision can have an associated description. You can use revisions to organize the placement constraints created for your LogicLock regions.

LogicLock Assignment Precedence

You can encounter conflists 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:

- Exact node-level assignments
- Path-based and wildcard assignments
- Hierarchical assignments

For more information about LogicLock assignment precedence, refer to *Understanding Assignment Priority* in Quartus II Help.

Open the **Priority** dialog box by selecting **Priority** on the **General** tab of the **LogicLock Regions Properties** dialog box. You can change the priority of path-based and wildcard assignments with the **Up** and **Down** buttons in the **Priority** dialog box. To prioritize assignments between regions, you must select multiple LogicLock regions and then open the **Priority** dialog box from the **LogicLock Regions Properties** dialog box.

Virtual Pins

Usually, when you compile a design in the Quartus II software, all I/O ports are mapped directly to pins on the targeted device. However, you may not want to map all I/O ports to the device pins; use the Virtual Pin assignment then.

A virtual pin is an I/O element that you do not intend to connect to the chip pins. You create a virtual pin by assigning the Virtual Pin logic option to an I/O element. When you compile a design with some I/O elements assigned as virtual pins, those I/O elements become mapped to a logic element and not to a pin during compilation, and are then implemented as a LUT. You might use virtual pin assignments when you compile a partial design, because not all the I/Os from a partial design may drive chip pins at the top level.

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 you designate as a virtual pin becomes mapped to either an LCELL or an adaptive logic module (ALM), depending on the target device.

You cannot use virtual pins for bidirectional, registered I/O pins, or I/O pins with output enable signals.

In the top-level design, you connect these virtual pins to an internal node of another module. By making assignments to virtual pins, you can place those pins in the same location or region on the device as that of the corresponding internal nodes in the top-level module. You can use the Virtual Pin option when compiling a LogicLock module with more pins than the target device allows. The Virtual Pin option can enable timing analyses that more closely match the performance of the LogicLock module when you integrate it into the top-level design.

Apply the following guidelines when creating virtual pins in the Quartus II software:

- Do not declare clock pins as virtual pins
- You should not declare nodes or signals that drive physical device pins in the top-level design as virtual pins

In the Node Finder, you can set **Filter Type** to **Pins: Virtual** 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 LogicLock regions, and assign resources to them, in the Chip Planner. You can also create new LogicLock regions using the Chip Planner.

Viewing Connections Between LogicLock Regions in the Chip Planner

You can view and edit LogicLock regions using the Chip Planner. Select the **Floorplan Editing** task, or any task that has the **User-assigned LogicLock regions** setting enabled to manipulate LogicLock regions.

The Chip Planner shows the connections between LogicLock regions. By default, you can view each connection as an individual line drawn between LogicLock regions. You can choose to display connections between LogicLock regions as a single bundled connection rather than as individual connection lines. To use this option, open the Chip Planner and on the View menu, click **Inter-region Bundles**.

In the **Inter-region Bundles** dialog box, specify the **Maximum source node to region fanout** and the **Minimum Bundle Width** values.

For more information about the **Inter-region Bundles** dialog box, refer to *Inter-region Bundles Dialog Box* in Quartus II Help.

Using LogicLock Regions with the Design Partition Planner

You can help optimize timing in a design by placing entities that share significant logical connectivity close to each othert on the device. By default, the Fitter usually places closely connected entities in the same area of the device; however, you can use LogicLock regions, together with the Design Partition Planner and the Chip Planner, to help ensure that logically connected entities retain optimal placement from one compilation to the next.

You can view the logical connectivity between entities with the Design Partition Planner, and the physical placement of those entities with the Chip Planner. In the Design Partition Planner, you can identify entities that are highly interconnected, and place those entities in a partition. In the Chip Planner, you can create LogicLock regions and assign each partition to a LogicLock region, thereby preserving the placement of the entities.

For more information about using LogicLock regions with design partitions, 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*. For more information about using the Design Partition Planner with the Chip Planner, refer to *About the Design Partition Planner* and *Using the Design Partition Planner* in Quartus II Help.

Design Floorplan Analysis Using the Chip Planner

The Chip Planner helps you visually analyze 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, which you can use to change the properties and parameters of device resources, and modify connectivity between certain types of device resources. The Change Manager records any changes that you make to your design floorplan, so that you can selectively undo 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*, and to *About the Resource Property Editor* and *About the Change Manager* in Quartus II Help.

The following sections present Chip Planner floorplan views and design analysis procedures which you can use with any predefined task—unless a procedure requires a specific 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 zoom in, the level of abstraction decreases, revealing more detail about your design.

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

Bird's Eye View

The Bird's Eye View 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.

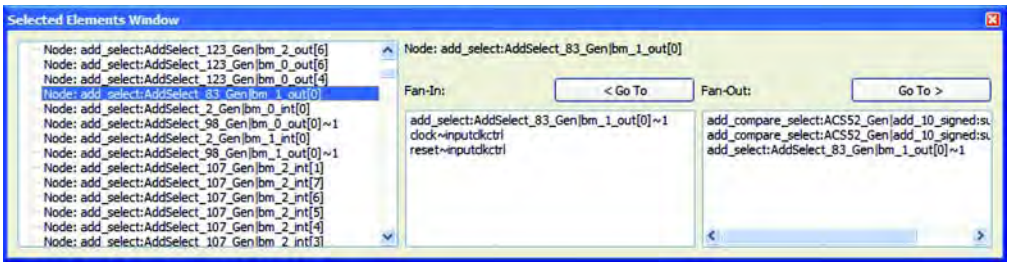
The Bird's Eye View is particularly useful when the parts of your design that you want to view are at opposite ends of the chip and you want to quickly navigate between resource elements without losing your frame of reference.

For more information about the Bird's Eye View, refer to *Bird's Eye View* and *Displaying Resources and Information* in Quartus II Help.

Selected Elements Window

The Selected Elements Window lists the objects (such as atoms, paths, LogicLock regions, or routing elements) currently selected in the Chip Planner. To display the Selected Elements Window, click **Selected Elements Window** on the **View** menu in the Chip Planner.

Figure 15–5. Selected Elements Window



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 logic element (LE) configuration in 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 ALM configuration in 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 device I/O resource usage. 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 phase-locked loop (PLL) configuration in 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 device properties 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.

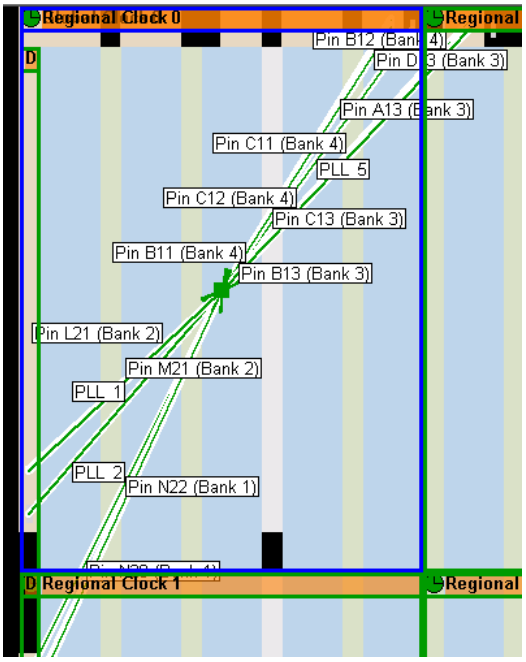
Viewing Available Clock Networks in the Device

When you select a task with clock region layers enabled, you can display the areas of the chip that are driven by global and regional clock networks. This global clock display feature is available for Arria GX, Arria II GX, Cyclone II, Cyclone III, HardCopy II, HardCopy III, Stratix II, Stratix II GX, Stratix III, Stratix IV, and Stratix V device families.

Depending on the clock layers activated in the selected task, the Chip Planner displays regional and global clocks and the regions they cover in the device, and the connectivity between clock regions, pins, and PLLs. Clock regions appear as rectangular overlay boxes with labels indicating the clock type and index.You can select each clock network region by clicking on the clock region. The clock-shaped icon at the top-left corner indicates that the region represents a clock network region. You can change the color in which the Chip Planner displays clock regions on the **Options** page of the Tools menu.

The Layer Settings dialog box lists layers for different clock region types; when the selected device does not contain a given clock region, the option for that category is unavailable in the dialog box. You can customize the Chip Planner's display of clock regions by creating a custom task with selected clock layers enabled in the Layers Settings dialog box. Figure 15–6 shows the potential fan-in in the Chip Planner.

Figure 15–6. Potential Fan-In



For more information about displaying potential fan-ins or fan-outs relative to a clock region, refer to *Displaying Resources and Information* in Quartus II Help.

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 from registers to device I/Os. The slack of a path determines its criticality; slack appears in the timing analysis report. Design analysis for timing closure is a fundamental requirement for optimal performance in highly complex designs. The analytical capability of the Chip Planner helps you close timing on complex designs.

Viewing critical paths in the Chip Planner helps you understand why a specific path is failing. You can see if any modification in the placement can 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 **Expand Connections/Paths** in the toolbar, or by clicking on the "+/-" on the label.

You can locate failing paths from the timing report in the TimeQuest Timing Analyzer. To locate the critical paths, run the Report Timing task from the Custom Reports group in the Tasks pane of the TimeQuest Timing Analyzer. From the View pane, which lists the failing paths, right-click on any failing path or node, and select **Locate Path**. From the Locate dialog box, select **Chip Planner** to see the failing path in the Chip Planner.

To display paths in the floorplan, you must first make timing settings and perform a timing analysis.

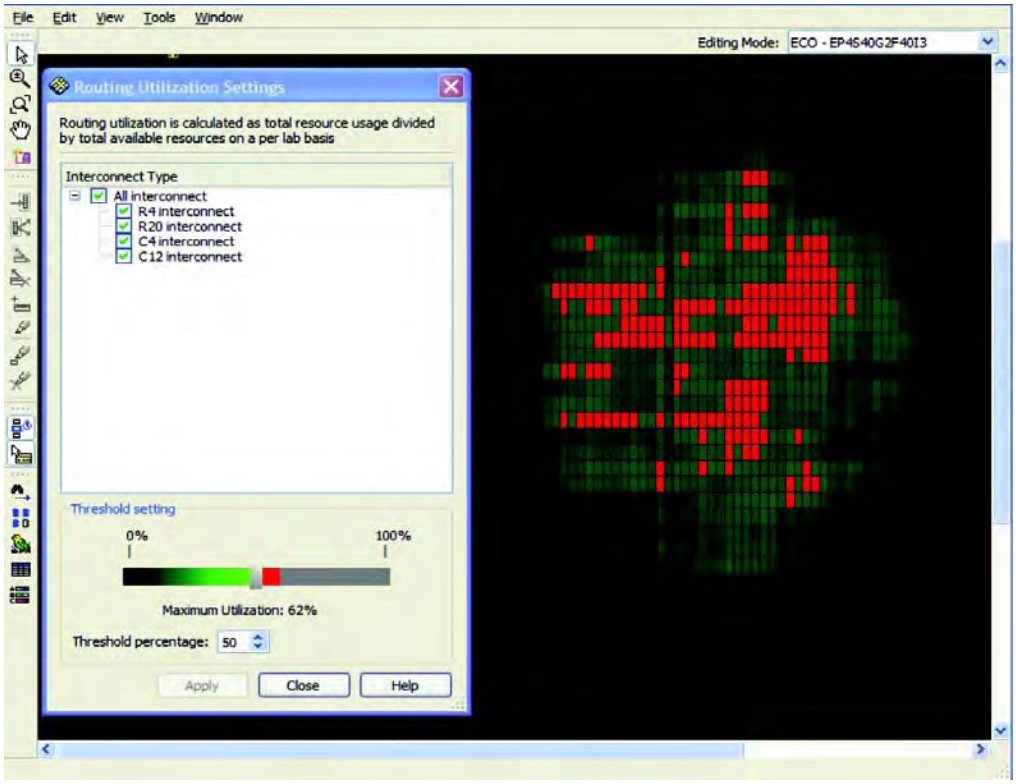
For more information about performing static timing analysis with the Quartus II TimeQuest Timing Analyzer, refer to the *The Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Viewing Routing Congestion

The **Routing Congestion** task allows you to determine the percentage of routing resources in use following a compilation. This feature identifies where there is a lack of routing resources, and helps you make design changes that might ease routing congestion and help meet design requirements.

To view routing congestion in the Chip Planner, select the **Routing Congestion** task. The **Routing Utilization Settings** dialog box appears whenever you select the **Routing Congestion** task; this dialog box allows you to set a congestion threshold value, and to specify the types of routing interconnects of interest (Figure 15–7).

Figure 15–7. Routing Utilization Settings dialog box



For more information about displaying routing congestion, refer to *Displaying Resources and Information* in Quartus II Help.

The routing congestion map uses the color and shading of logic resources to indicate relative resource utilization; darker shading represents a greater utilization of routing resources. Areas where routing utilization exceeds the threshold value specified in the Routing Utilization Settings dialog box appear in red. The congestion map can help you determine whether you can modify the floorplan, or make changes to the RTL to reduce routing congestion.

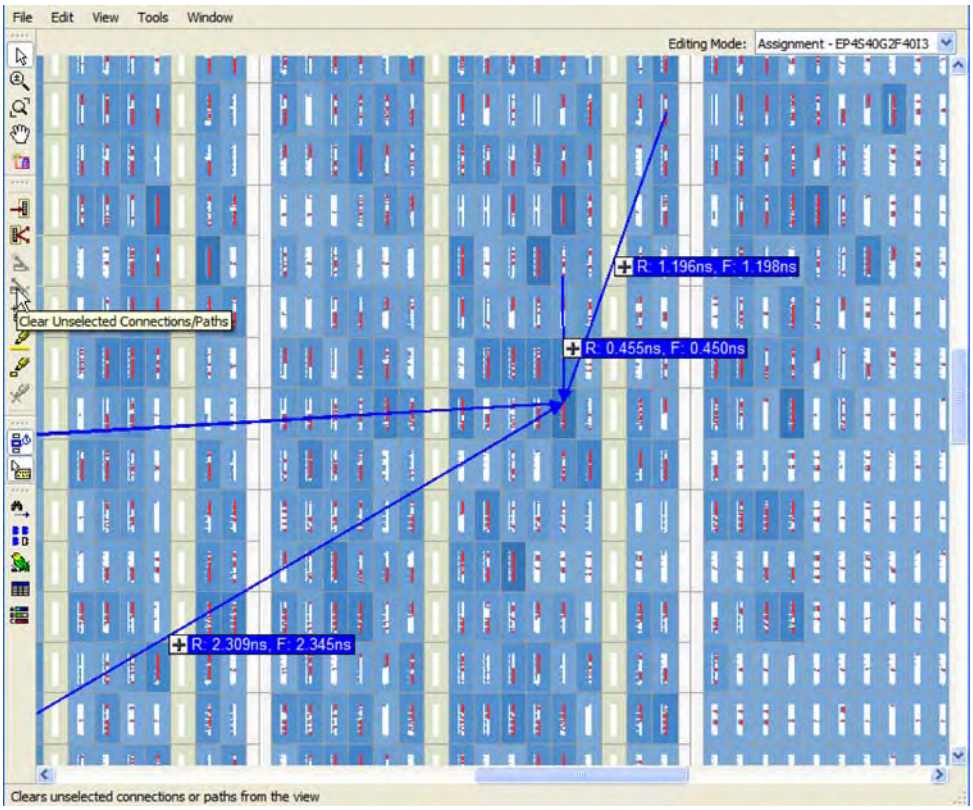
Viewing I/O Banks

The Chip Planner can show all of the I/O banks of the device. To see the I/O bank map of the device, turn on the I/O Banks layer in the Layers Settings dialog box.

Generating Fan-In and Fan-Out Connections

The ability to display fan-in and fan-out connections 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 15–8 shows the fan-in connections for the selected resource.

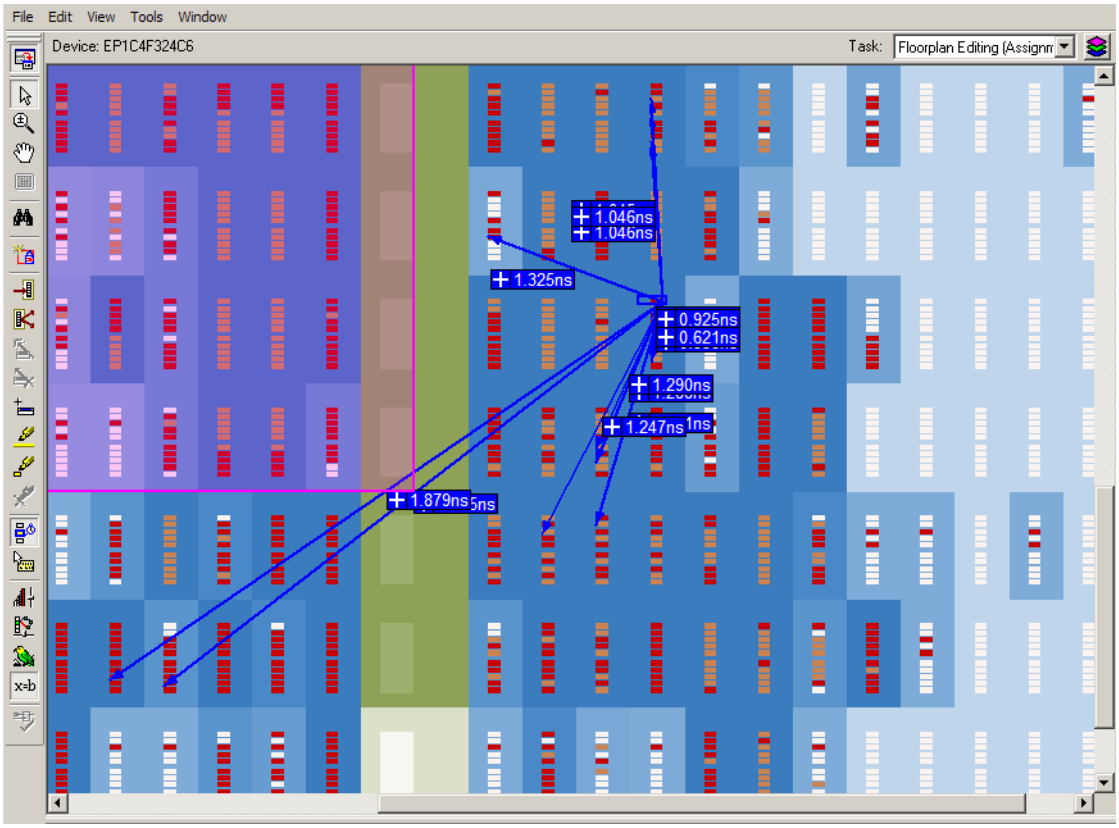
Figure 15–8. Generated Fan-In



Generating Immediate Fan-In and Fan-Out Connections

The ability to display immediate fan-in and fan-out connections enables you to view the resource that is the immediate fan-in or fan-out connection for the selected atom. For example, if you select a logic resource and choose to view the immediate fan-in for that resource, you can 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 displayed connections from the screen, click the Clear Connections icon in the toolbar. Figure 15–9 shows the immediate fan-out connections for the selected resource.

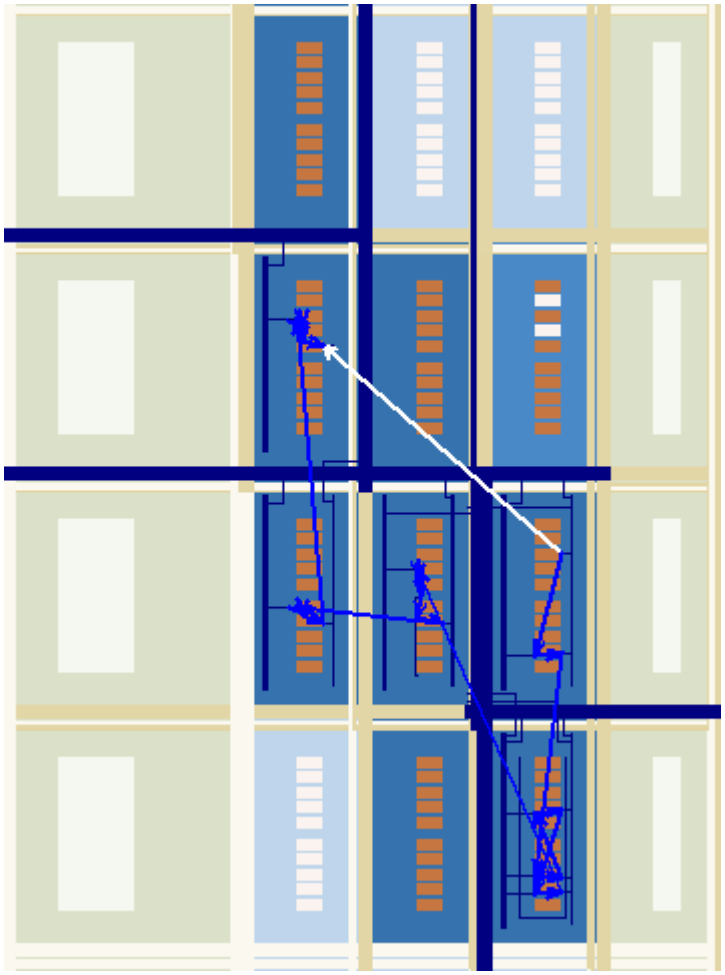
Figure 15–9. Immediate Fan-Out Connection



Highlight Routing

The **Highlight Routing** command enables you to highlight the routing resources used by a selected path or connection. Figure 15–10 shows the routing resources in use between two logic elements.

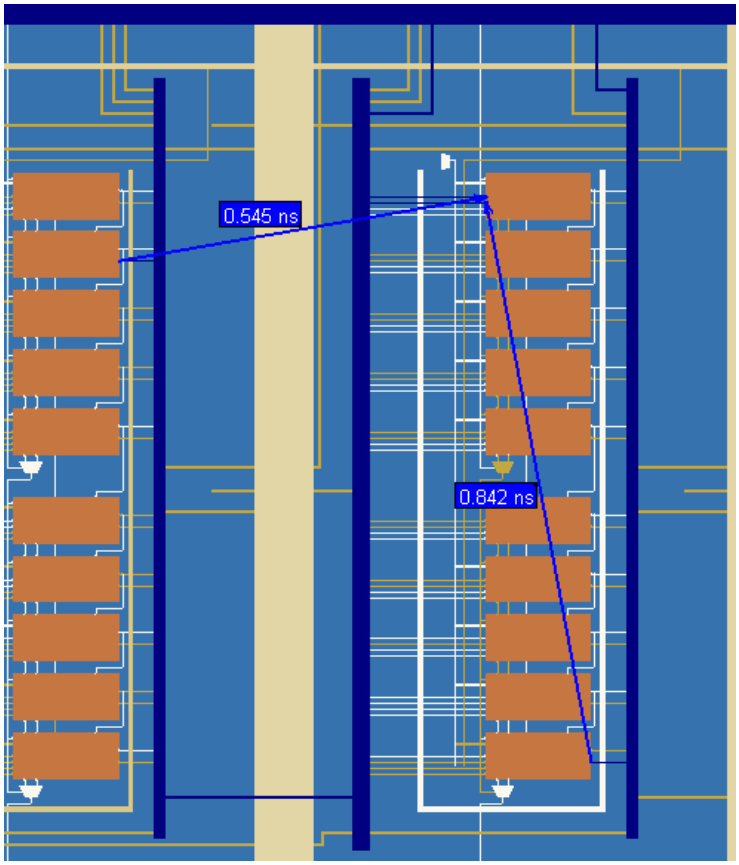
Figure 15–10. Highlight Routing



Show Delays

With the Show Delays command, you can view timing delays for connections displayed in the Chip Planner. For example, you can view the delay between two logic resources or between a logic resource and a routing resource. Figure 15–11 shows the delays between several logic elements.

Figure 15–11. 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

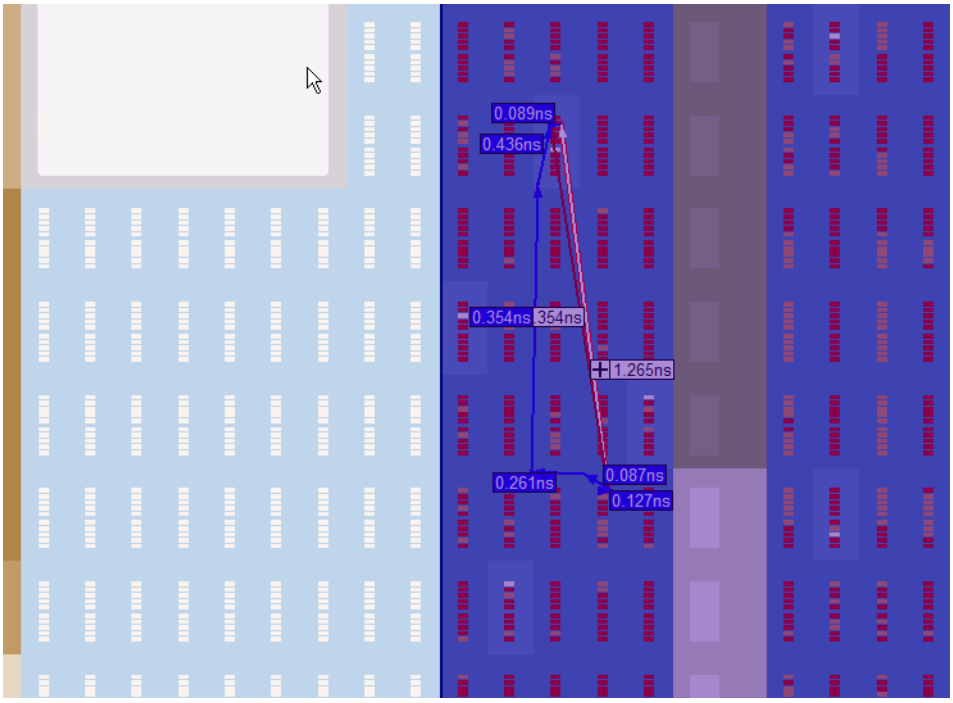
To locate a path from the Timing Analysis report to the Chip Planner, perform the following steps:

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)**.
3. 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 15–12 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 15–12. Path Analysis

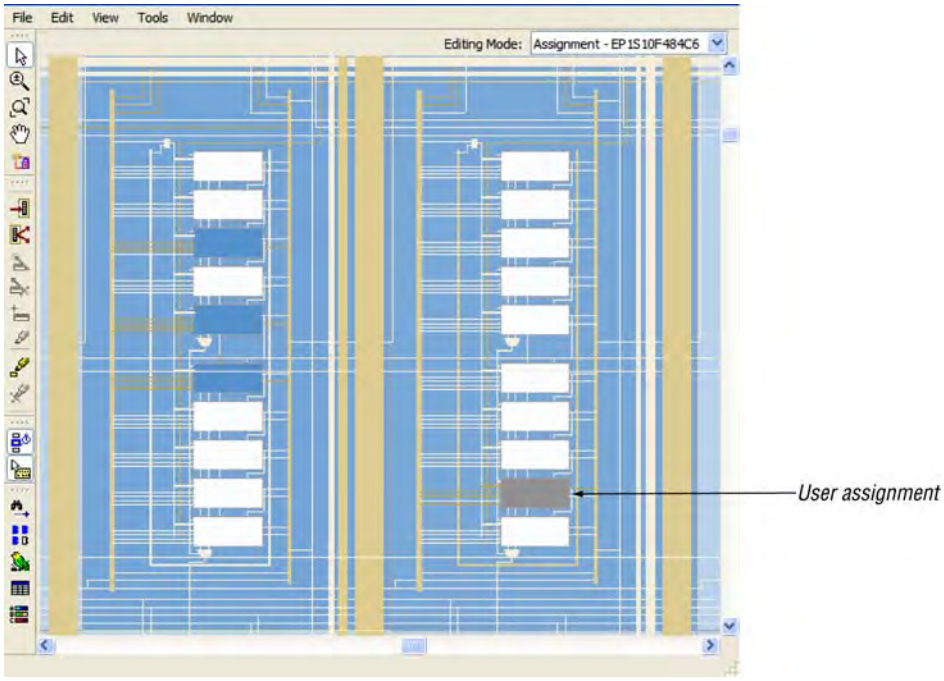


Viewing Assignments in the Chip Planner

You can view location assignments by selecting the appropriate layer set in the Chip Planner. To view location assignments, select the **Floorplan Editing** task or any custom task that displays block utilization, and the Assignment editing mode. See Figure 15–13.

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 15–13. Viewing Assignments in the Chip Planner



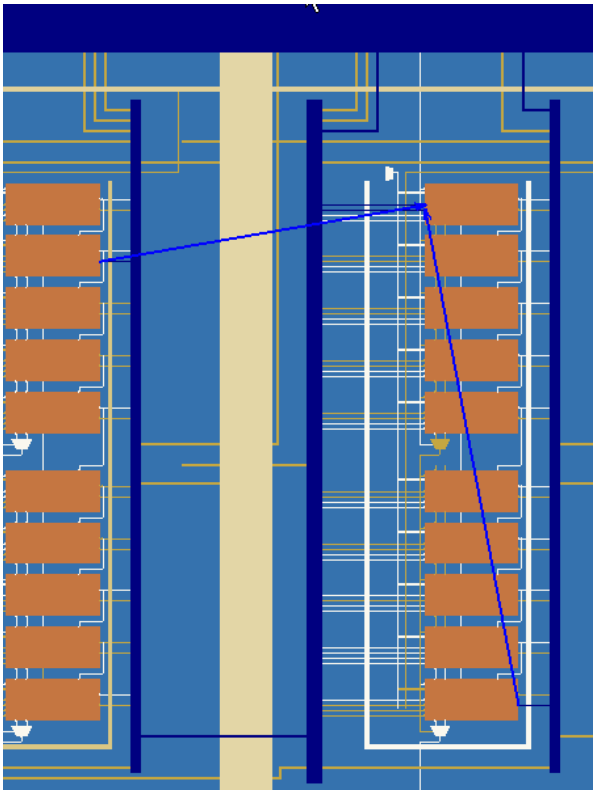
You can make node and pin location assignments and assignments to LogicLock regions and custom regions using the drag-and-drop method in the Chip Planner. The Fitter applies the assignments that you create during the next place-and-route operation.

For more information about managing assignments in the Chip Planner, refer to *Working With Assignments in the Chip Planner* in 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 15–14 shows the routing channels used for the selected path in the Chip Planner.

Figure 15–14. Highlight Routing

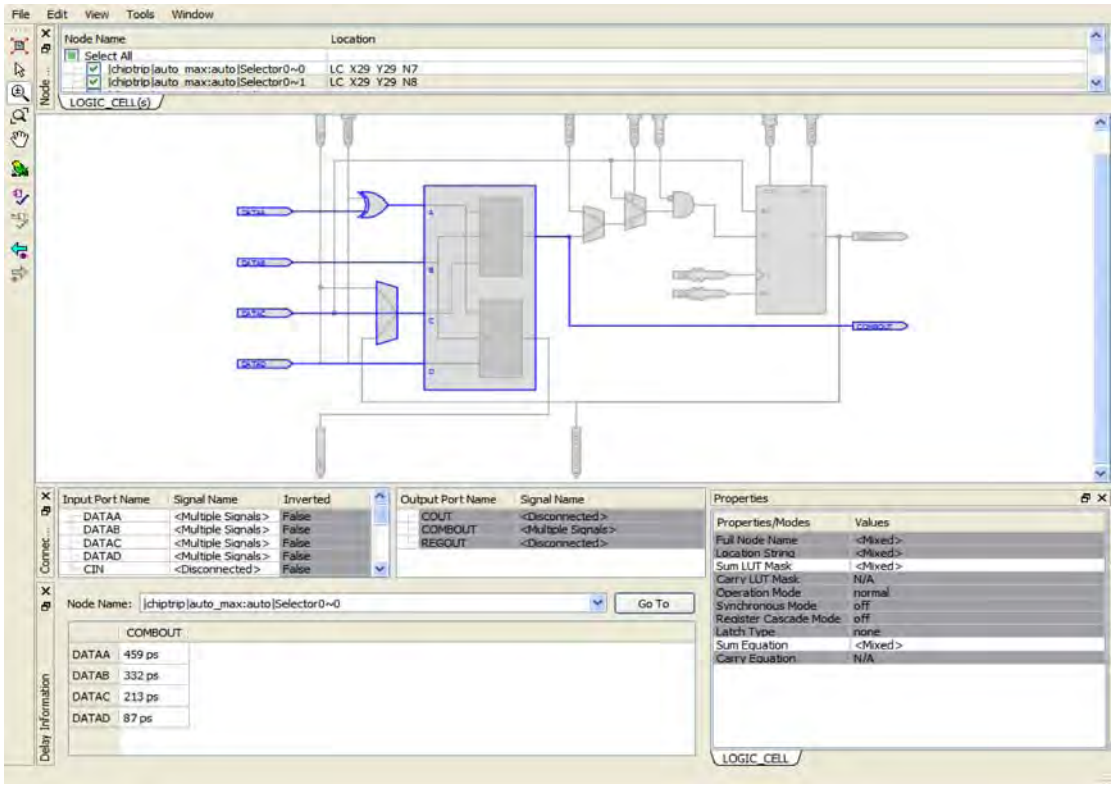


You can view and edit resources in the FPGA using the Resource Property Editor. For more information, refer to the *Engineering Change Management with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Delay Information Table

In the Resource Property Editor, you can view the propagation delay from all inputs to all outputs for any LE in your design. To see the delay information for an atom, right-click the atom in the Chip Planner, click **Locate,** and then click **Locate in Resource Property Editor**. In the Resource Property Editor, click **View Delay Information** on the **View** menu to display the delay information. The Delay Information Table shows you the propagation delay from all inputs to all outputs. Figure 15–15 shows the Resource Property Editor, including the Delay Information Table.

Figure 15–15. Delay Information Table



Timing numbers are displayed only when a direct path exists 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 appears as "N/A."

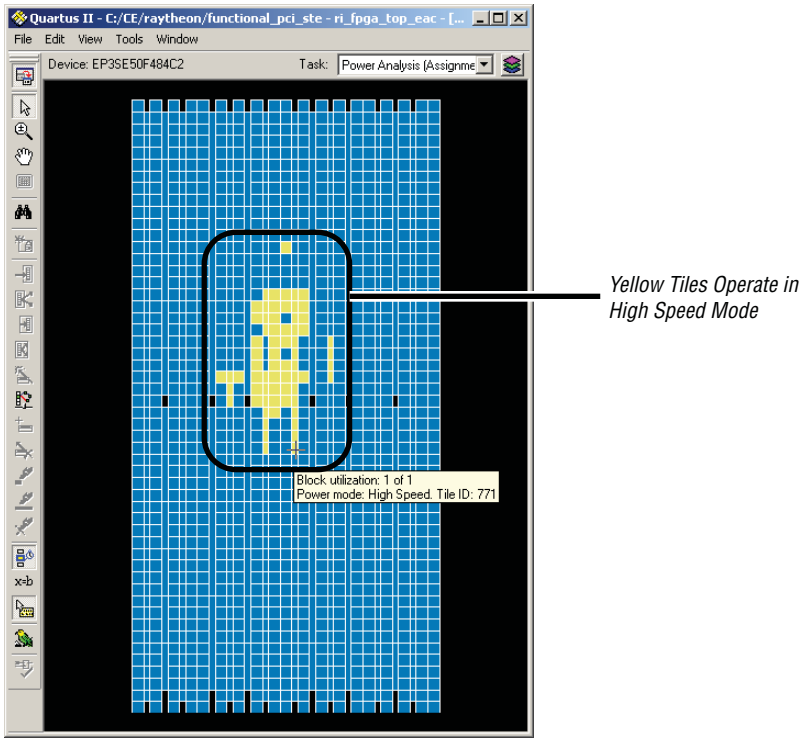
Viewing High-Speed and Low-Power Tiles in the Chip Planner

The Chip Planner has a predefined task, **Power**, which shows the power map of Stratix III, Stratix IV, and Stratix V devices; these 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*. To learn more about power analyses and optimizations in Stratix IV devices, refer to *AN 514: Power Optimization in Stratix IV FPGAs*.

When you select the **Power** task in the Chip Planner for Stratix III, Stratix IV, or Stratix V devices, the Chip Planner displays low-power and high-speed tiles in contrasting colors; yellow tiles operate in a high-speed mode, while blue tiles operate in a low-power mode (see Figure 15–16). When you select the **Power** task, you can perform all floorplanner-related functions for this task; however, you cannot edit tiles to change the power mode.

Figure 15–16. Viewing High-Speed and Low Power Tiles in a Stratix III Device



Scripting Support

You can run procedures and specify 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 ↵
```

Information about scripting command options is also available in *API Functions for Tcl* in Quartus II Help.

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>
```

The command in the above example sets the size of the region to auto and the state 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 "Creating LogicLock Regions" on page 15–5.

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 15–7.

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.

The saving of a node-level netlist to a persistent source file is not supported for designs targeting newer devices such as Arria GX, Arria II GX, Cyclone III, Stratix III, Stratix IV, or Stratix V.

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
```

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 15–11.

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 optimal performance in highly complex designs. With analysis capability, the Quartus II Chip Planner helps you close timing quickly on your designs. Using the Chip Planner 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.

Document Revision History

Table 15–2 shows the revision history for this chapter.

Table 15–2. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Updated device support information ■ Removed references to Timing Closure Floorplan; removed "Design Analysis Using the Timing Closure Floorplan" section ■ Added links to online Help topics ■ Added "Using LogicLock Regions with the Design Partition Planner" section ■ Updated "Viewing Critical Paths" section ■ Updated several graphics ■ Updated format of Document revision History table
November 2009	9.1.0	■ Updated supported device information throughout ■ Removed deprecated sections related to the Timing Closure Floorplan for older device families. (For information on using the Timing Closure Floorplan with older device families, refer to previous versions of the *Quartus II Handbook*, available in the Quartus II Handbook Archive.) ■ Updated "Creating Non-Rectangular LogicLock Regions" section ■ Added "Selected Elements Window" section ■ Updated table 12-1
March 2009	9.0.0	■ Was chapter 10 in the 8.1.0 release.
November 2008	8.1.0	■ 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
May 2008	8.0.0	■ 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

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

Take an online survey to provide feedback about this handbook chapter.

16. Netlist Optimizations and Physical Synthesis

QII52007-10.0.0

The Quartus® II software offers physical synthesis optimizations to improve your design beyond the optimization performed in the normal course of the Quartus II compilation flow.

Physical synthesis optimizations can help improve the performance of your design regardless of the synthesis tool used, although the effect of physical synthesis optimizations depends on the structure of your design.

Netlist optimization options work with the atom netlist of your design, which describes a design in terms of Altera®-specific primitives. An atom netlist file can be 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 to improve the quality of results. This chapter also provides information about preserving 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 physical synthesis optimizations, you should 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 Logic Analyzer, formal verification, or the LogicLock based optimization flow (for legacy devices), you must turn off 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 either an **.edf** or **.vqm** atom netlist file using Altera-specific primitives. When you turn on the **Perform WYSIWYG primitive resynthesis** 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**).

The **Perform WYSIWYG primitive resynthesis** option has no effect if you are using Quartus II integrated synthesis to synthesize your design.

To turn on the **Perform WYSIWYG primitive resynthesis** 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. Turn on **Perform WYSIWYG Primitive Resynthesis**, 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 logic option is available for all Altera devices supported by the Quartus II software except MAX 3000 and MAX 7000 devices.

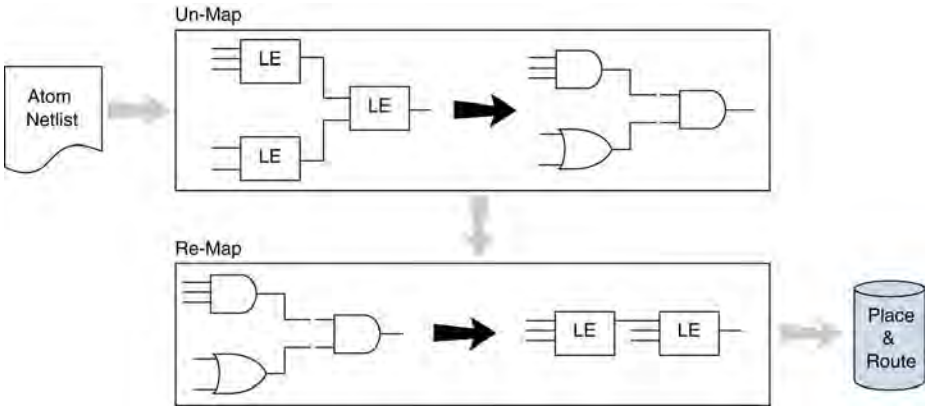
The results of the remapping depend on the **Optimization Technique** you choose. To select an **Optimization Technique**, perform the following steps:

1. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
2. Under **Optimization Technique**, select **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 of the design 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 16–1 shows the Quartus II software flow for the WYSIWYG primitive resynthesis feature.

Figure 16–1. WYSIWYG Primitive Resynthesis



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 remapped. Logic specified in an encrypted **.vqm** file or an **.edf** file, such as third-party intellectual property (IP), is not touched.

The **Perform WYSIWYG primitive resynthesis** option can change 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 by the Quartus II software. The remapping process removes duplicate registers, but registers that are not removed retain 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. You can use the Assignment Editor to apply the **Netlist Optimizations** logic option. 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

The Quartus II design flow involves 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, the physical synthesis optimizations available in the Quartus II software take those routing delays into consideration and focus timing-driven optimizations at those 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 series, Cyclone, HardCopy, and Stratix series device families.

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 both device architectures individually because doing so results in a different post-fitting netlist for each device.

For more information about physical synthesis optimizations, refer to *Physical Synthesis Optimizations Page (Settings Dialog Box)* in Quartus II Help. 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 under the Compilation Process Settings page in the Settings dialog box. 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; however, you can select the **Fast** effort level if you want to limit the increase in compilation time. When you select the **Fast** effort level, the Quartus II software performs limited register retiming operations during fitting. The **Extra** effort level runs additional algorithms to get the best circuit performance, but results in increased compilation time.

To optimize 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, select **Physical Synthesis Optimizations** under **Compilation Process Settings.** 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.

All Physical Synthesis optimizations write results to the **Netlist Optimizations** report, which provides a list of atom netlist files that were modified, created, and deleted during physical synthesis. To access the **Netlist Optimizations** report, perform the following steps:

1. On the Processing menu, click **Compilation Report**.
2. In the **Compilation Report** list, select **Netlist Optimizations** under **Fitter.**

Similarly, physical synthesis optimizations performed during synthesis write results to the synthesis report. To access this report, perform the following steps:

1. On the Processing menu, click **Compilation Report**.
2. In the **Compilation Report** list, select **Analysis & Synthesis.**
3. In the **Optimization Results** folder, select **Netlist Optimizations**. The **Physical Synthesis Netlist Optimizations** table appears, listing the physical synthesis netlist optimizations performed during synthesis.

Nodes or entities that have the **Netlist Optimizations** logic option set to **Never Allow** are not affected by the physical synthesis algorithms. You can use the Assignment Editor to apply the **Netlist Optimizations** logic option. 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.

The **Perform automatic asynchronous signal pipelining** 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. These additional registers add register delays (adds latency) to the reset, adding the same number of register delays for each destination using the reset. The additional register delays can change the behavior of the signal in the design; therefore, you should use this option only if additional latency on the 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 turned on. 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

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 critical paths, you can turn on the **Perform physical synthesis for combinational logic** option, which swaps the look-up table (LUT) ports within LEs so that the critical path has fewer layers through which to travel. The **Perform physical synthesis for combinational logic** option also allows the duplication of LUTs to enable further optimizations on the critical path.

For more information about using the **Perform physical synthesis for combinational logic** option, refer to *Physical Synthesis Optimizations Page (Settings Dialog Box)* and to *Setting Up and Running the Fitter* in Quartus II Help.

The **Perform 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 want to 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.

For more information about the **Perform register duplication** option, refer to *Physical Synthesis Optimizations Page (Settings Dialog Box)* and to *Setting Up and Running the Fitter* in Quartus II Help.

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*.

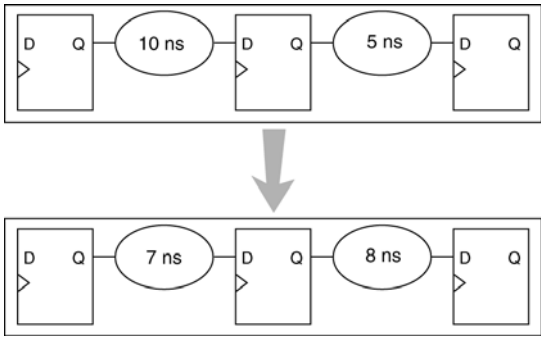
If you want to consider logic cells that meet any of these conditions for physical synthesis, you can override these rules by setting the **Netlist Optimizations** logic option to **Always Allow** on a given set of nodes.

Physical Synthesis for Registers—Register Retiming

The **Perform 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.

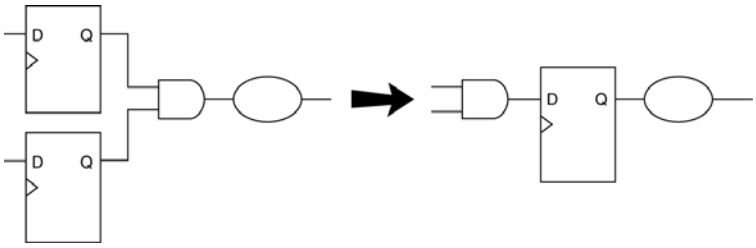
Figure 16–2 shows an example of register retiming in which the 10-ns critical delay is reduced by moving the register relative to the combinational logic.

Figure 16–2. 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 16–3).

Figure 16–3. 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, select **Physical Synthesis Optimizations** under **Compilation Process Settings**. The **Physical Synthesis Optimizations** page appears.
3. Specify your preferred option under **Optimize for performance (physical synthesis)** and **Effort level**.
4. Click **OK**.

For more information about the **Optimize for performance (physical synthesis)** options and effort levels, refer to *Physical Synthesis Optimizations Page (Settings Dialog Box)* in Quartus II Help.

If you want to prevent register movement during register retiming, you can set the **Netlist Optimizations** logic option to **Never Allow**. You can apply this option to either 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 does not move them, 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.

The Quartus II software assumes that a synchronization register chain consists of two registers. If your design has synchronization register chains with more than two registers, you must indicate the number of registers in your synchronization chains so that they are not affected by register retiming. 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.

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

The Quartus II software does not usually retime registers across different clock domains; however, if you use the Classic Timing Analyzer and specify a global f_{MAX} requirement, the Quartus II software interprets all clocks as related. Consequently, the Quartus II software might try to retime register-to-register paths associated with different clocks.

To avoid this circumstance, provide individual f_{MAX} requirements to each clock when using Classic Timing Analysis. When you constrain each clock individually, the Quartus II software assumes no relationship between different clock domains and considers each clock domain to be asychronous to other clock domains; hence no register-to-register paths crossing clock domains are retimed.

When you use the TimeQuest Timing Analyzer, register-to-register paths across clock domains are never retimed, because the TimeQuest Timing Analyzer treats all clock domains as asychronous to each other unless they are intentionally grouped.

- 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, hence you do not need to preserve your results from compilation to compilation. When you make changes 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*, and to *About Incremental Compilation* in Quartus II Help.

You can preserve the resulting nodes from physical synthesis in older devices that do not support incremental compilation. You might need to preserve nodes if you use the LogicLock flow to back-annotate placement, import one design into another, or both. For all device families that support incremental compilation, use that feature to preserve results.

To preserve the nodes from Quartus II physical synthesis optimization options for older devices that do not support incremental compilation (such as Max II devices), 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 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 you 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 you have finalized the design. 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, you must 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 Arria, Cyclone, or Stratix series, use incremental compilation to preserve compilation results instead of using logic back-annotation.

Physical Synthesis Options for Fitting

The Quartus II software provides physical synthesis optimization options for improving fitting results. To access these options, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Physical Synthesis Optimizations** under **Compilation Process Settings**. 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 16–1).

Table 16–1. 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.

For more information about physical synthesis optimization options, refer to *Physical Synthesis Optimizations Page (Settings Dialog Box)* in Quartus II Help.

Applying Netlist Optimization Options

The improvement in performance when using netlist optimizations is design dependent. If you have restructured your design to balance critical path delays, netlist optimizations might yield minimal improvement in performance. You may have to experiment with available options to see which combination of settings works best for a particular design. Refer to the messages in the compilation report to see the magnitude of improvement with each option, and to help you decide whether you should turn on a given option or specific effort level.

Turning on more netlist optimization options can result in more changes to the node names in the design; bear this in mind if you are using a verification flow, such as the SignalTap II 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. The WYSIWYG primitive resynthesis option 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 **DSE**, refer to *About Design Space Explorer* in Quartus II Help.

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 about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook* and *API Functions for Tcl* in Quartus II Help. 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 specify many of the options described in this section on either an instance or 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 16–2 lists the Quartus II Settings File (**.qsf**) variable names and applicable values for the settings discussed in "WYSIWYG Primitive Resynthesis" on page 16–1. 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 16–2. 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	*<file name>*	
Allow Netlist Optimizations	ADV_NETLIST_OPT_ALLOWED	"ALWAYS ALLOW", DEFAULT, "NEVER ALLOW"	Instance

Physical Synthesis Optimizations

Table 16–3 lists the **.qsf** file variable name and applicable values for the settings discussed in "Performing Physical Synthesis Optimizations" on page 16–3. 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 16–3. Physical Synthesis Optimizations and Associated Settings (Part 1 of 2)

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

Table 16–3. Physical Synthesis Optimizations and Associated Settings (Part 2 of 2)

Setting Name	Quartus II Settings File Variable Name	Values	Type
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	<file name>	

Incremental Compilation

For information about scripting and command line usage for incremental compilation as mentioned in "Preserving Your Physical Synthesis Results" on page 16–10, 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 16–10.

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. You can take advantage of these Quartus II netlist optimizations to help improve your quality of results.

Document Revision History

Table 16–4 shows the revision history for this chapter.

Table 16–4. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Added links to Quartus II Help in several sections. ■ Removed Referenced Documents section. ■ Reformatted Document Revision History
November 2009	9.1.0	■ Added information to "Physical Synthesis for Registers—Register Retiming" ■ Added information to "Applying Netlist Optimization Options" ■ Made minor editorial updates
March 2009	9.0.0	■ 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
November 2008	8.1.0	Changed to 8½" × 11" page size. No change to content.
May 2008	8.0.0	Updated "Physical Synthesis Optimizations for Performance on page 11-9 Added Physical Synthesis Options for Fitting on page 11-16

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

Take an online survey to provide feedback about this handbook chapter.

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 17, Engineering Change Management with the Chip Planner

  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.

17. Engineering Change Management with the Chip Planner

QII52017-10.0.0

The Chip Planner allows you to make small changes to your design after the design is fully compiled. 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 making ECOs, and describes how you can use the Chip Planner to make ECOs.

The Chip Planner does not support the MAX® 7000 device family.

In addition to making 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 more 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*.

ECOs directly apply to atoms in the target device. As such, performing an ECO relies on your understanding of the device architecture of the target device. For more information about the architecture of your device, refer to the appropriate device handbook on the Literature page of the Altera website.

Engineering Change Orders

ECOs are small targeted changes that fine-tune the functionality of a design. In the context of an FPGA design, an ECO is applied directly to a physical resource on the device to modify its behavior. ECOs are typically made 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. Making 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 Preservation"
- "Compilation Time"
- "Verification"
- "Change Modification Record"

Performance Preservation

You can preserve the results of previous design optimizations when you are making changes to an existing design with one of the following methods:

- Incremental compilation
- Rapid recompile
- ECOs

Choose the method to modify your design based on the scope of the change. The methods above are arranged from the larger scale change to the smallest targeted change to a compiled design.

The incremental compilation feature allows you to preserve compilation results at an RTL component or module level. After the initial compilation of your design, you can assign modules in your design hierarchy to partitions, and preservation level to each partition. Upon subsequent recompiles, only changed modules are recompiled. Unmodified modules are preserved according to their preservation level.

The rapid recompilation feature leverages results from the latest post-fit netlist to determine the changes required to honor modifications you have made to the source code. If you turn on the rapid recompilation feature, the compiler attempts to refit only the portion of the netlist that is related to the code modification.

ECOs provide a finer granularity of control compared to the incremental compilation and the rapid recompilation feature. All modifications are performed directly on the architectural elements of the device architecture. You should use ECOs for targeted changes to the post-fit netlist.

In the Quartus II software versions 10.0 and later, ECO modifications to the netlist are not preserved when you recompile a design with the incremental compilation feature turned on. You can reapply ECO changes made during a previous compilation with the Change Manager.

For more information about the 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*.

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. Because 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 on your design. To verify that your changes do not violate timing requirements, you can perform static timing analysis with the TimeQuest Timing Analyzer after you check and save your netlist changes in the Chip Planner.

For more information about the Quartus II TimeQuest analyzer, refer to the *Quartus II TimeQuest 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 with the post place-and-route netlist generated by the Quartus II software.

Change Modification Record

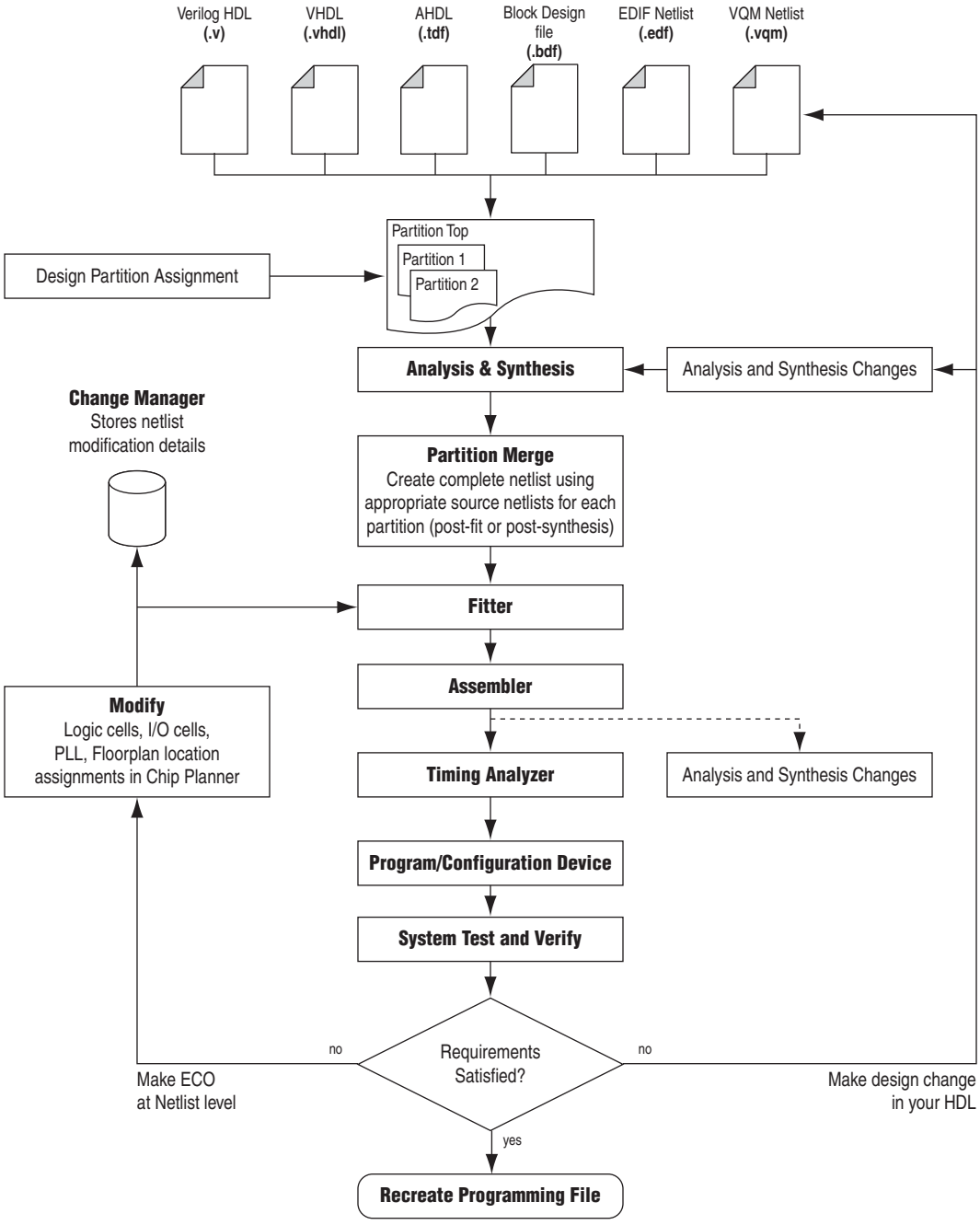
All ECOs made with the Chip Planner are logged in the Change Manager to provide a track record of all changes. With the Change Manager, you can easily revert to the original post-fit netlist or you can pick and choose which ECOs you want to apply.

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 revert back to previous project revisions when required.

ECO Design Flow

Figure 17–1 shows the design flow for making ECOs.

Figure 17–1. Design Flow to Support ECOs



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 time to generate a programming file for debugging the design.

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. By making small changes with the Chip Planner you can modify the post place-and-route netlist directly, without 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

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 blocks, or I/O elements.

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 with 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 resource atoms. 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, you can open the Chip Planner by cross-probing from the shortcut menu in the following tools:

- Compilation Report
- Project Navigator window
- RTL source code

- Node Finder
- Simulation Report
- RTL Viewer

The Chip Planner Tasks and Layers

The Chip Planner allows 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.

For more information about tasks and layers in the Chip Planner, refer to *About the Chip Planner* in Quartus II Help. For more information about creating assignments and performing 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 making ECOs with the ECO mode, refer to "Performing ECOs with the Chip Planner (Floorplan View)" on page 17–10.

The Chip Planner Floorplan Views

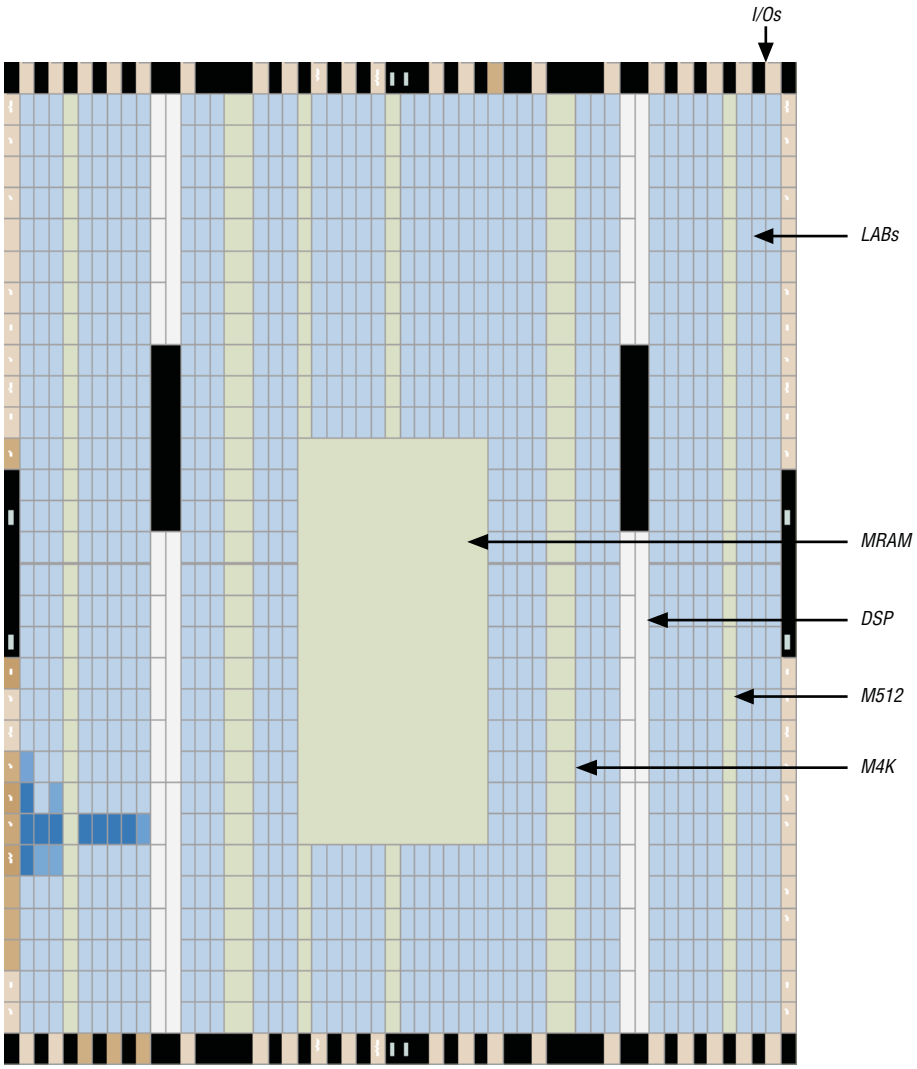
The Chip Planner uses a hierarchical zoom viewer that shows various abstraction levels of the targeted Altera device. As you increase the zoom level, the level of abstraction decreases, thus revealing more detail about your design.

First-Level View

The first zoom level provides a high-level view of the entire device floorplan. You can locate and view the placement of any node in your design.

Figure 17–2 shows the Chip Planner Floorplan first-level view of a Stratix® device.

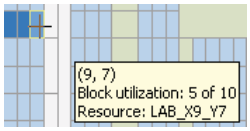
Figure 17–2. The Chip Planner First-Level (Highest) Floorplan View (Stratix Family Device)



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 17–3).

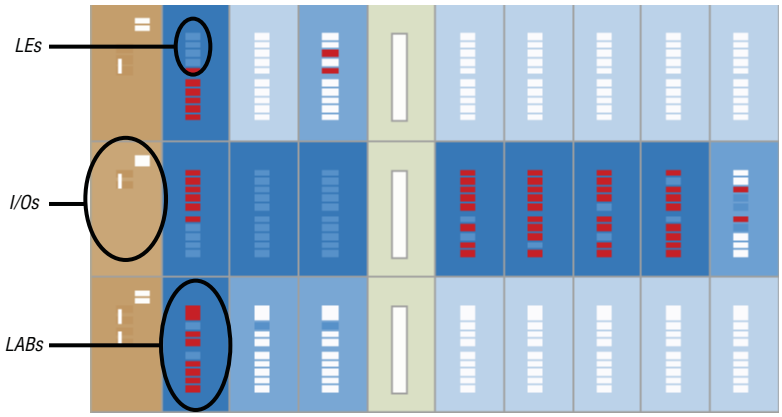
Figure 17–3. Tooltip Message: First-Level View



Second-Level View

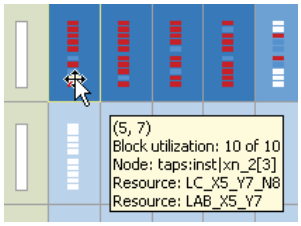
As you zoom in, you see an increase in the level of detail. Figure 17–4 shows the second-level view of the Chip Planner Floorplan for a Stratix device.

Figure 17–4. 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 17–5) 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.

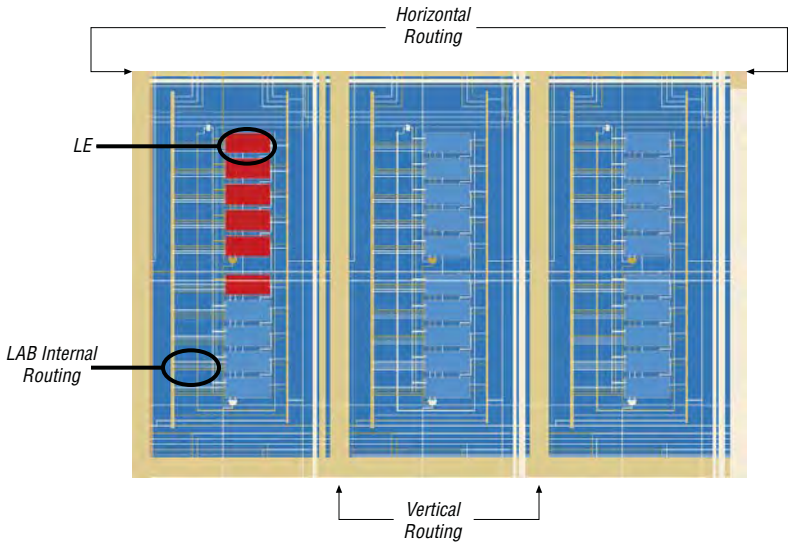
Figure 17–5. Tooltip Message: Second-Level View



Third-Level View

At the third level, which provides the greatest level of detail, you can see each routing resource that is used in a LAB in the FPGA. Figure 17–6 shows the level of detail at the third-level view for a Stratix device.

Figure 17–6. 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 17–10.

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 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.

The Bird's Eye View is particularly useful when the parts of your design in which you are interested are at opposite ends of the chip and you want to quickly navigate between resource elements without losing your frame of reference.

For more information about the Bird's Eye View, refer to *Bird's Eye View Window* and *Displaying Resources and Information* in Quartus II Help.

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 made with the Chip Planner Floorplan view:

- "Creating Atoms"
- "Deleting Atoms" on page 17–14
- "Moving Atoms" on page 17–14

To configure the properties of atoms, such as managing the connections between different LEs/ALMs, use the Resource Property Editor.

For more information about editing atom resource properties, refer to "Performing ECOs in the Resource Property Editor" on page 17–15.

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 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 right-clicking the desired resource atom 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)
- I/O elements

The creation of resource atoms is not supported in the **Assignment** editing mode.

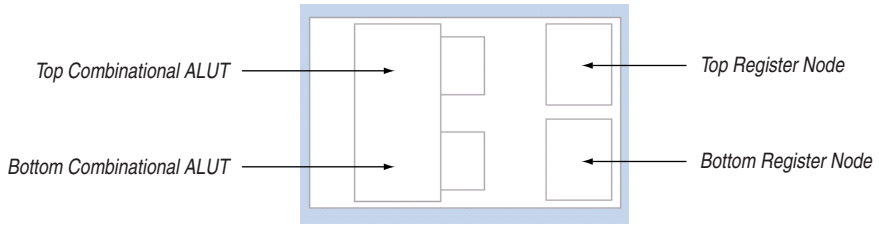
For the architectural details and resource atoms supported by your device, refer to the device data sheet.

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 17–7 shows an ALM in the Chip Planner.

Figure 17–7. ALM in the Chip Planner



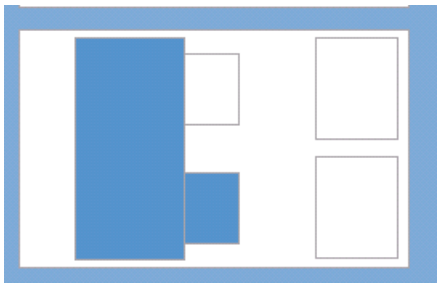
To create a combinational ALM LUT atom, perform the following steps:

1. Right-click 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 17–8 shows a combinational ALUT atom.

Figure 17–8. Combinational ALUT Atom



To create a registered ALM atom, perform the following steps:

1. Right-click 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 the Cyclone®, MAX, and Stratix device families as LEs. For all other LE-based device families, the Chip Planner shows resource atoms as the combinational output of the LE, LUT, and the registered output of the LE. Figure 17–9 shows an example of an atom resource in the Chip Planner for Cyclone, MAX II, and Stratix devices.

Figure 17–9. LE for Cyclone, MAX, and Stratix Devices in the Chip Planner

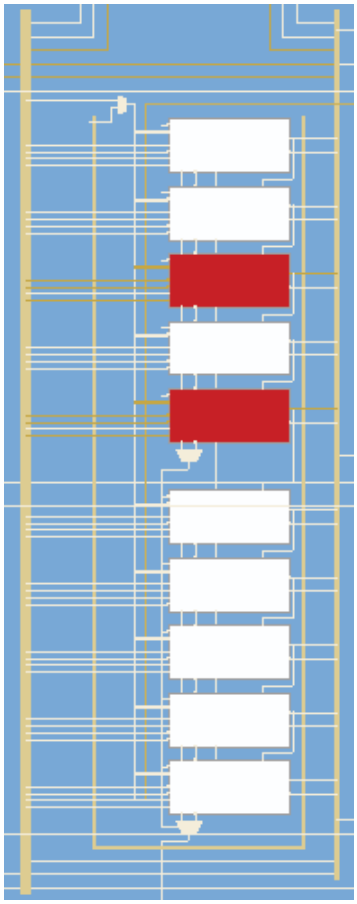
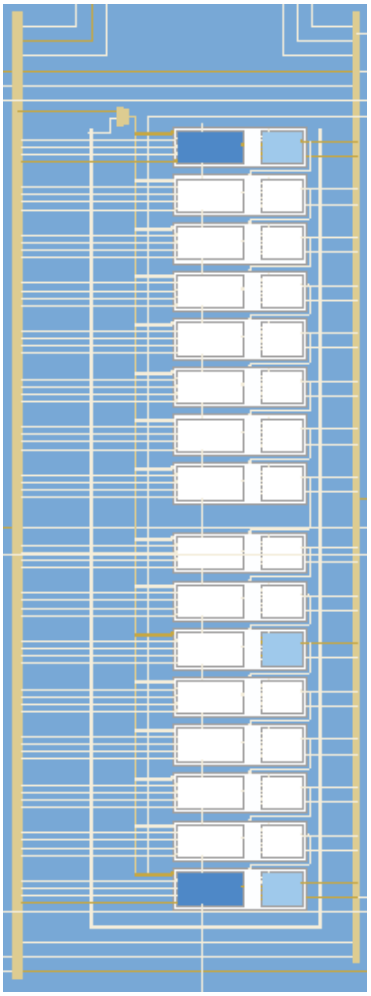


Figure 17–10 shows a Cyclone II resource atom in the Chip Planner.

Figure 17–10. LE for a Cyclone II Device in the Chip Planner



To create an LE resource for Cyclone, MAX. and Stratix device families, perform the following steps:

1. Right-click 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 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 the 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 atom, right-click 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.

For more information about removing fan-out connections, refer to "Performing ECOs in the Resource Property Editor" on page 17–15.

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 be done only 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 the 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.

Performing ECOs in the Resource Property Editor

You can view and edit the following resources with the Resource Property Editor:

- "Logic Element" on page 17–15
- "Adaptive Logic Module" on page 17–18
- "FPGA I/O Elements" on page 17–20
- "PLL Properties" on page 17–30
- "FPGA RAM Blocks" on page 17–25
- "FPGA DSP Blocks" on page 17–26

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 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. To open the Resource Property Editor for an LE, on the Project menu, point to **Locate,** and then click **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.

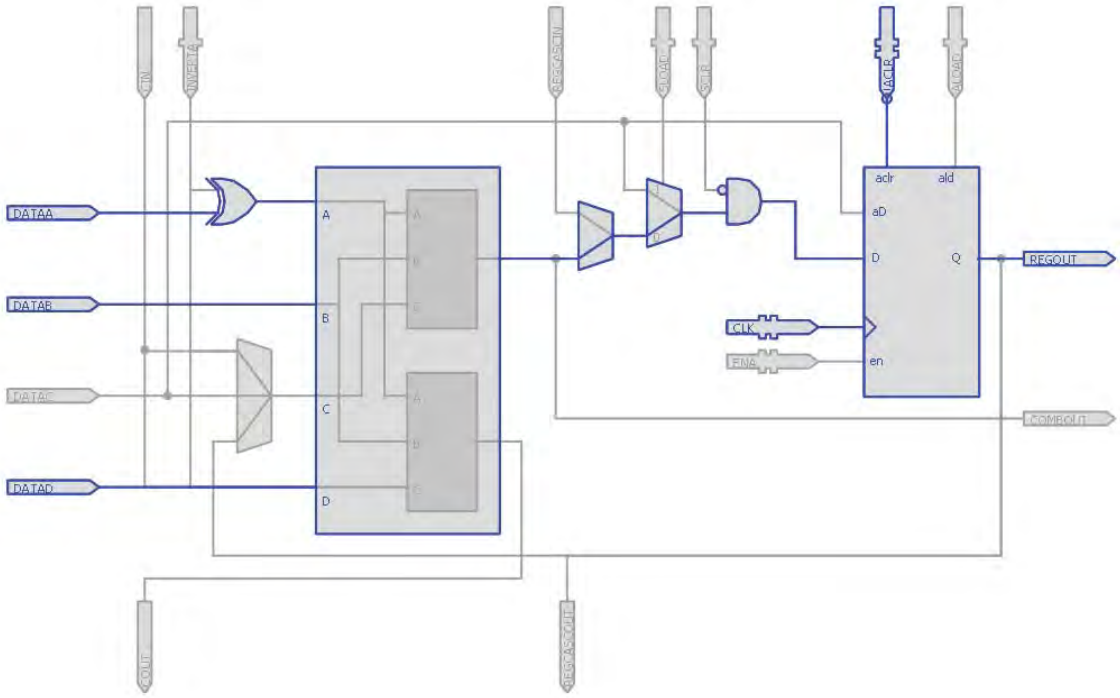
You can use the Resource Property Editor to change the following LE properties:

- Data input to the LUT
- LUT mask or LUT equation

Logic Element Schematic View

Figure 17–11 shows how the LE appears in the Resource Property Editor.

Figure 17–11. Stratix LE Architecture *(Note 1)*, *(2)*



Notes to Figure 17–11:

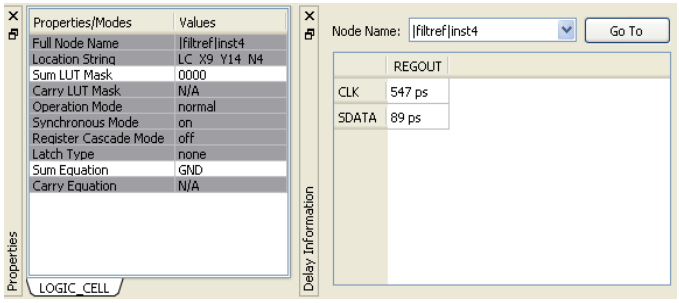
(1) By default, the Quartus II software displays the used resources in blue and the unused in gray.

(2) For more information about the Stratix device's LE architecture, refer to the *Stratix Device Handbook*.

LE Properties

Figure 17–12 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**.

Figure 17–12. 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 drive only 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 change only 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 sclr 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.

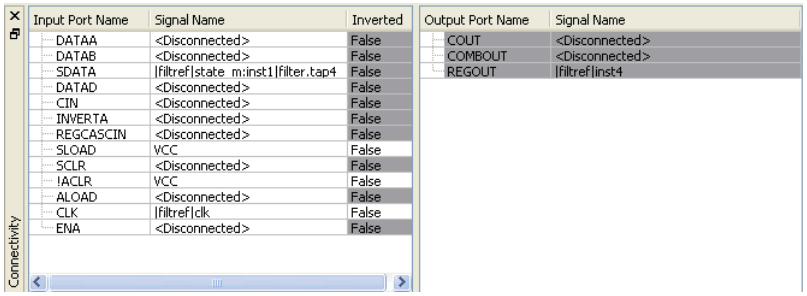
Cell Delay Table

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 17–13 shows the LE connections in the Connectivity window.

Figure 17–13. View LE 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 **Remove Fan-outs** dialog box 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**.

For more information about deleting atoms, refer to *Example of Deleting an Atom Using the Chip Planner and Resource Property Editor* in Quartus II Help.

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 ALUT-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 with 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:

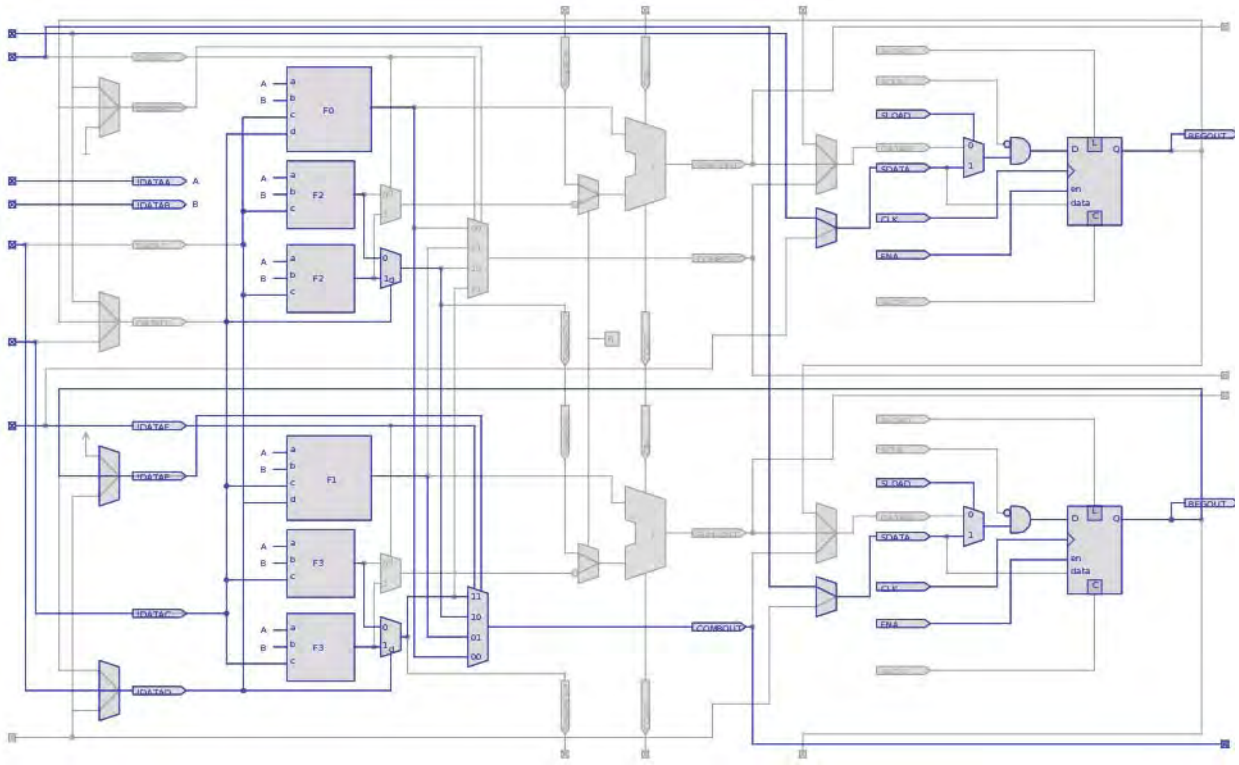
- Data input to the LUT
- LUT mask or LUT equation

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 17–14).

For a detailed description of the ALM, refer to the device handbooks of devices based on an ALM architecture.

Figure 17–14. ALM Schematic *(Note 1)*



Note to Figure 17–14:

(1) By default, the Quartus II software displays the used resources in blue and the unused in gray. For Figure 17–14, the used resources are in blue and the unused resources are in gray.

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 input and output connections for the ALM.

FPGA I/O Elements

Altera FPGAs that have high-performance I/O elements, 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 I/O elements. Use the Chip Planner (Floorplan view) to change placement, delete, and create new I/O elements.

For a detailed description of the device I/O elements, refer to the applicable device handbook.

You can change the following I/O properties:

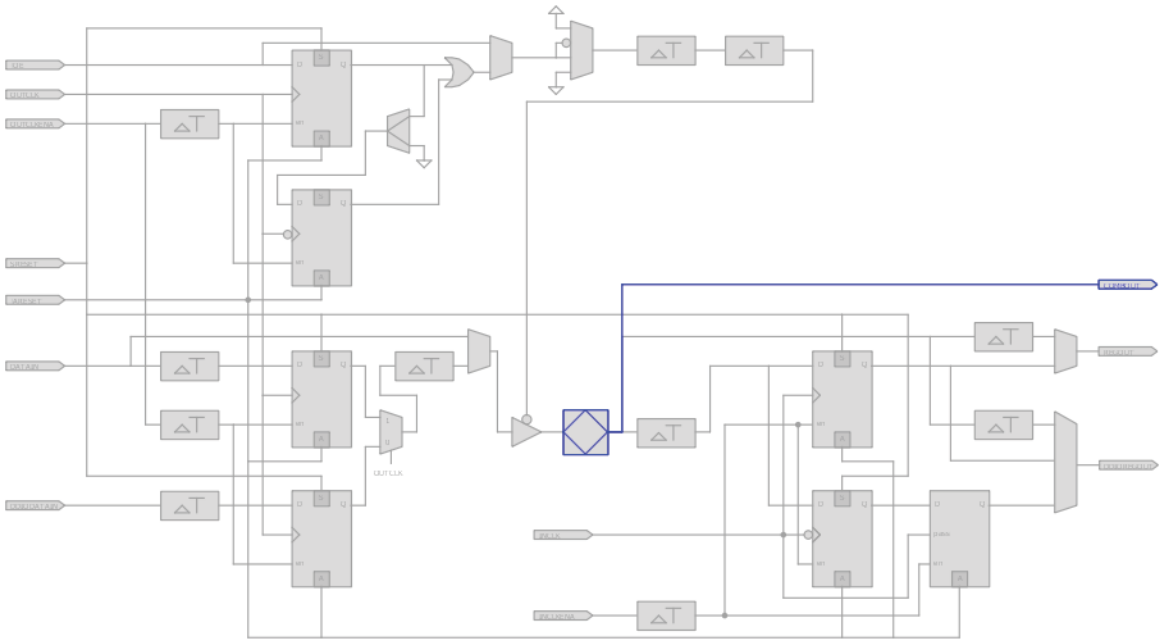
- Delay chain
- Bus hold
- Weak pull up
- Slow slew rate
- I/O standard
- Current strength
- Extend OE disable
- PCI I/O
- Register reset mode
- Register synchronous reset mode
- Register power up
- Register mode

Arria GX, Stratix, Stratix II, and Stratix GX I/O Elements

The I/O elements in Arria® GX, Stratix, Stratix II, and Stratix GX devices contain a bidirectional I/O buffer, six registers, and a latch for a complete bidirectional single data rate (SDR) or double data rate (DDR) transfer.

Figure 17–15 shows the Stratix and Stratix GX I/O element structure. The I/O element structure contains two input registers (plus a latch), two output registers, and two output enable registers.

Figure 17–15. Stratix and Stratix GX Device I/O Element and Structure *(Note 1)*, *(2)*



Notes to Figure 17–15:

(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 17–15, the used resources are in blue and the unused resources are in gray.

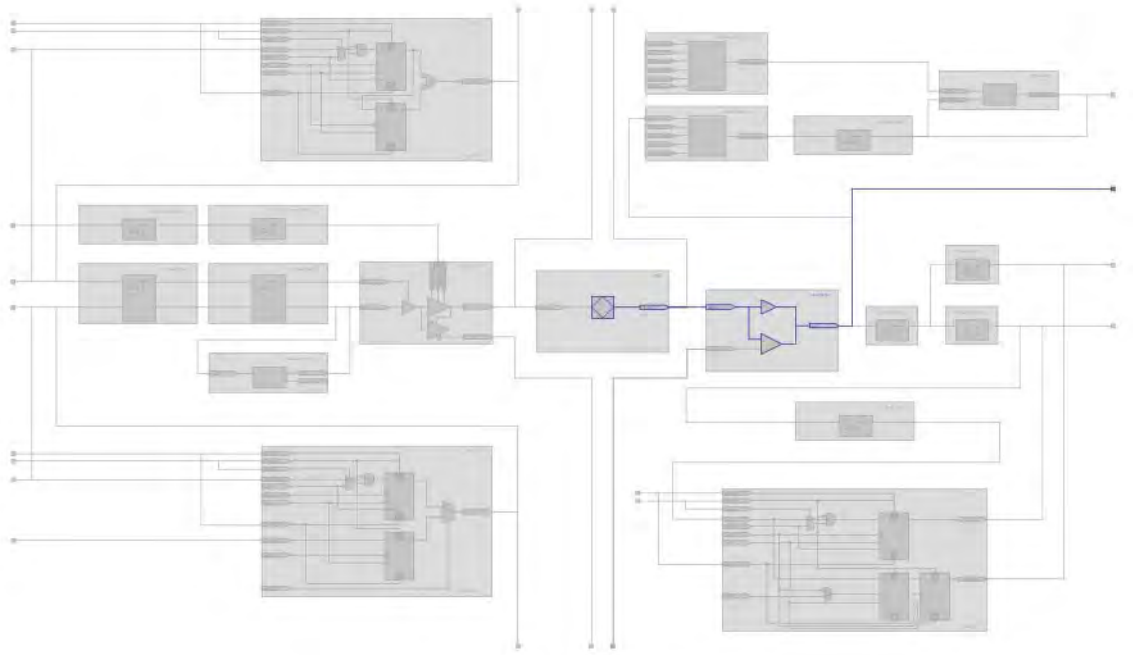
(2) For more information about I/O elements in Stratix and Stratix GX devices, refer to the *Stratix Device Handbook* and the *Stratix GX Device Handbook*.

Arria II GX, Stratix III, and Stratix IV I/O Elements

The I/O elements in Arria II GX, Stratix III, and Stratix IV 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 17–16). 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 I/O elements in Stratix III devices, refer to *AN 474: Implementing Stratix III Programmable I/O Delay Settings in the Quartus II Software*.

Figure 17–16. Stratix III Device I/O Element and Structure *(Note 1)*, *(2)*



Notes to Figure 17–16:

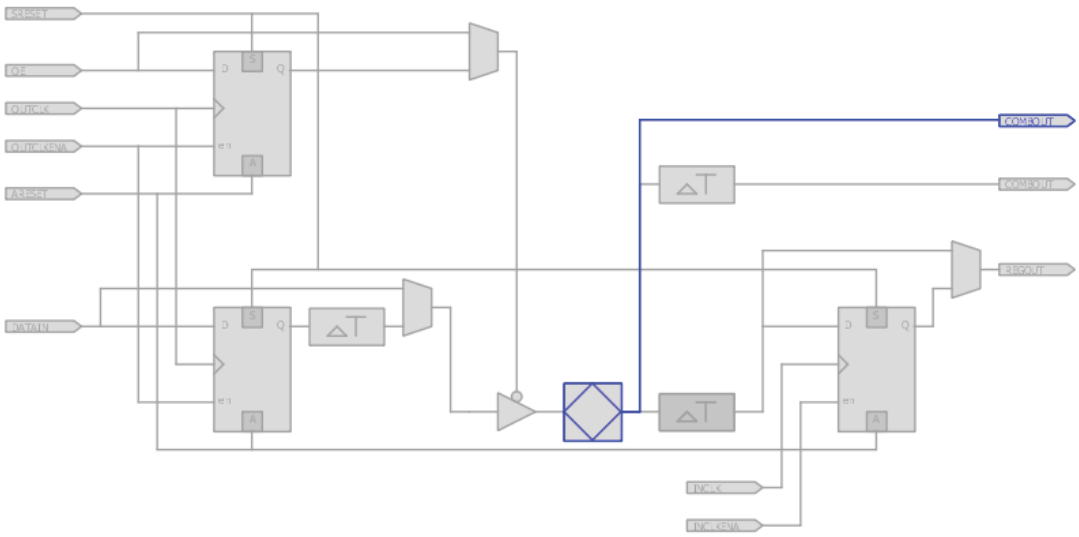
(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 17–16, the used resources are in blue and the unused resources are in gray.

(2) For more information about I/O elements in Stratix III and Stratix IV devices, refer to the Literature page of the Altera website.

Cyclone and Cyclone II I/O Elements

The I/O elements in Cyclone and Cyclone II devices contain a bidirectional I/O buffer and three registers for complete bidirectional single data-rate transfer. Figure 17–17 shows the Cyclone and Cyclone II I/O element structure. The I/O element contains one input register, one output register, and one output enable register.

Figure 17–17. Cyclone and Cyclone II Device I/O Elements and Structure *(Note 1)*, *(2)*



Notes to Figure 17–17:

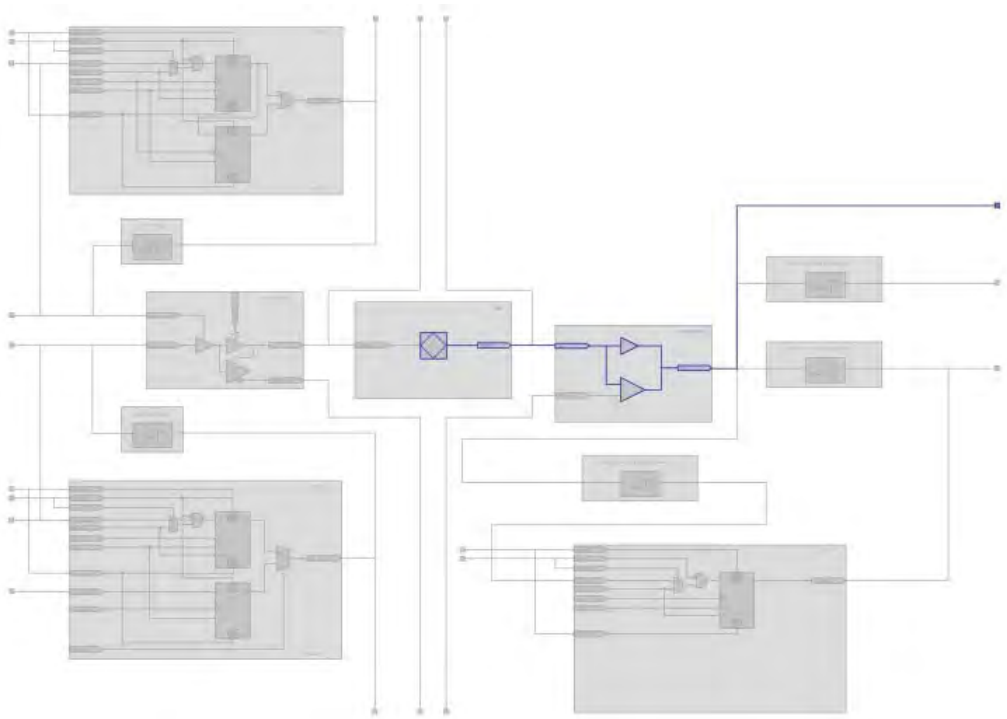
(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 17–17, the used resources are in blue and the unused resources are in gray.

(2) For more information about I/O elements in Cyclone II and Cyclone devices, refer to the *Cyclone II Device Handbook* and *Cyclone Device Handbook*, respectively.

Cyclone III I/O Elements

The I/O elements in Cyclone III devices contain a bidirectional I/O buffer and five registers for complete embedded bidirectional single data rate transfer. Figure 17–18 shows the Cyclone III I/O element structure. The I/O element 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 I/O elements for input, output, or bidirectional data paths.

Figure 17–18. Cyclone III Device I/O Elements and Structure *(Note 1)*, *(2)*



Notes to Figure 17–18:

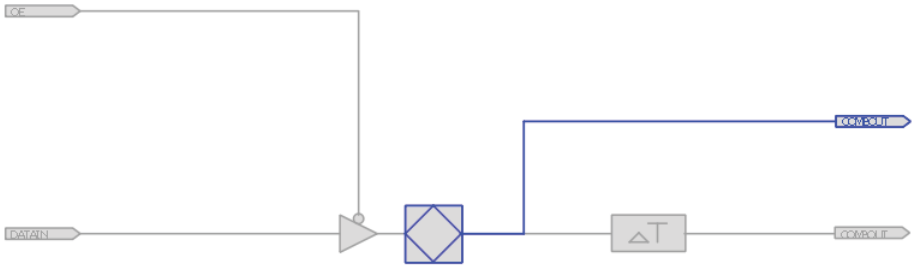
(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 17–18, the used resources are in blue and the unused resources are in gray.

(2) For more information about I/O elements in Cyclone III devices, refer to the *Cyclone III Device Handbook*.

MAX II I/O Elements

The I/O elements in MAX II devices contain a bidirectional I/O buffer. Figure 17–19 shows the MAX II I/O element structure. Registers from adjacent LABs can drive to or be driven from the I/O element's bidirectional I/O buffers.

Figure 17–19. MAX II Device I/O Elements and Structure *(Note 1)*, *(2)*



Notes to Figure 17–19:

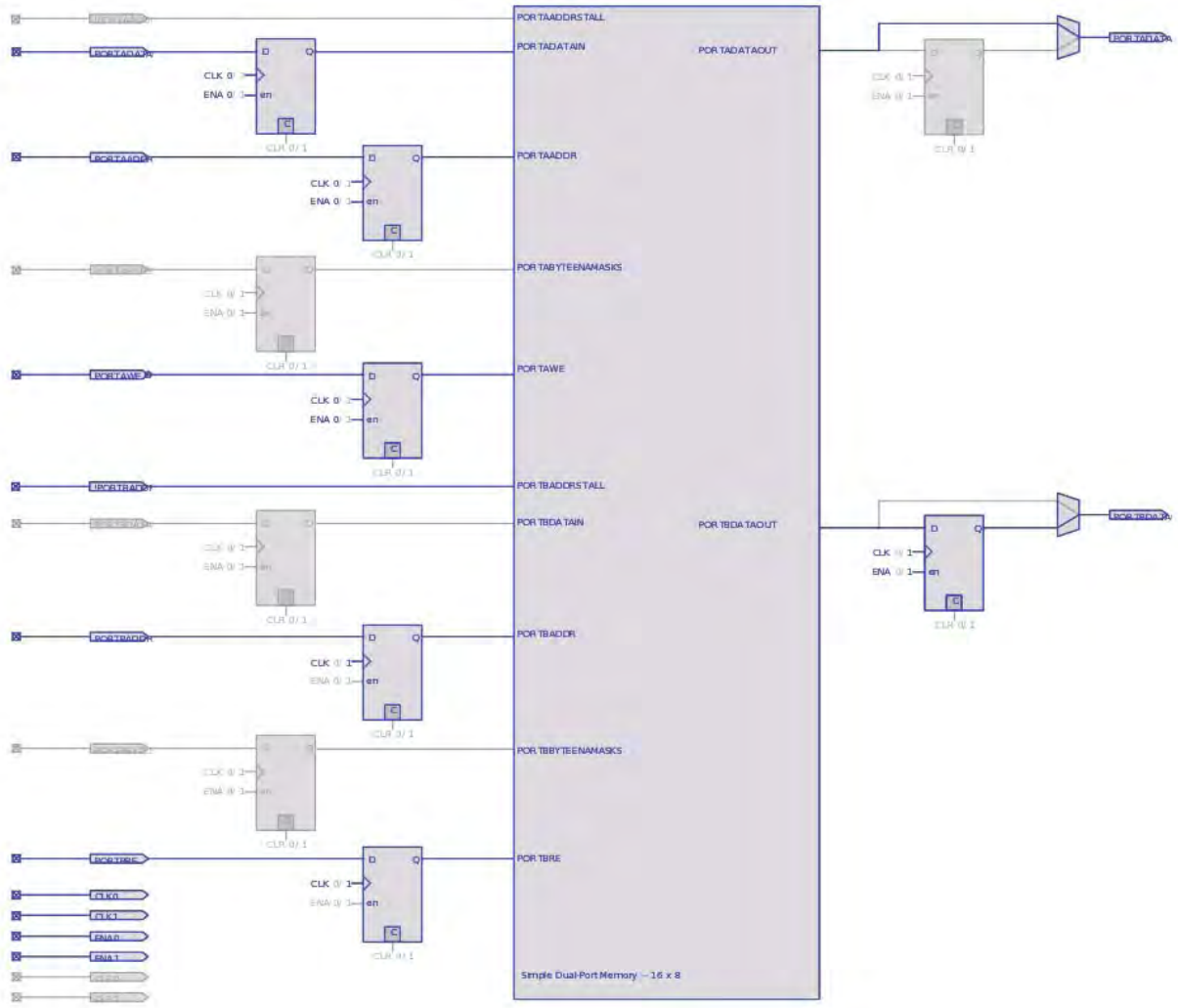
(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 17–19, the used resources are in blue and the unused resources are in gray.

(2) For more information about I/O elements 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 17–20 shows an M9K RAM view in a Stratix III device.

Figure 17–20. M9K RAM View in a Stratix III Device *(Note 1)*



Note to Figure 17–20:

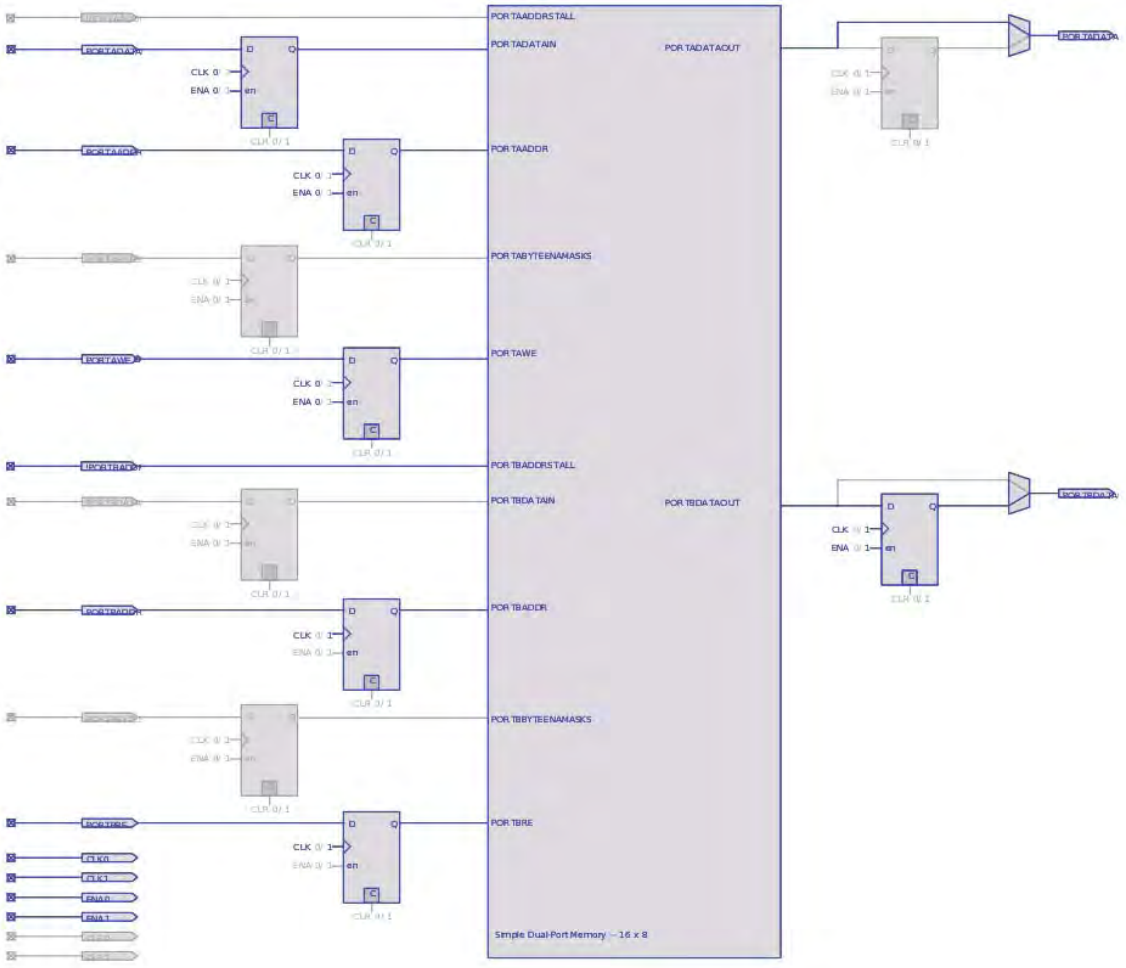
(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 17–20, the used resources are in blue and the unused resources are in gray.

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 Cyclone and Stratix 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 17–21 shows the DSP architecture in a Stratix III device.

Figure 17–21. DSP Block View in a Stratix III Device *(Note 1)*



Note to Figure 17–21:

(1) By default, the Quartus II software displays the used resources in blue and the unused resources in gray. In Figure 17–21, the used resources are in blue and the unused resources are in red.

Change Manager

The Change Manager maintains a record of every change you perform with the Chip Planner, Resource Property Editor, the SignalProbe feature, or a Tcl script. Each row of data in the Change Manager represents one ECO.

The Change Manager allows you to apply changes, roll back changes, delete changes, and export change records to a text file (**.txt**), a Comma-Separated Value File (**.csv**), or a Tcl Script File (**.tcl**). The Change Manager tracks dependencies between changes, so that when you apply, roll back, or delete a change, any prerequisite or dependent changes are also applied, rolled back, or deleted.

For more information about the Change Manager, refer to *About the Change Manager* in Quartus II Help.

Complex Changes in the Change Manager

Certain types of changes (including creating or deleting atoms and changing connectivity) can appear to be self-contained, but are actually composed of multiple actions. The Change Manager marks such complex changes with a plus icon in the **Index** column.

You can click the plus icon to expand the change record and show all the component actions preformed as part of that complex change.

For more information about complex change records and about managing changes with the Change Manager, refer to *Examples of Managing Changes With the Change Manager* in 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 made a SignalProbe assignment, you can use the Change Manager to quickly disable SignalProbe assignments by selecting **Revert to Last Saved Netlist** on the shortcut 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 changes to a **.txt**, a **.csv**, or a **.tcl**. Tcl scripts allow you to reapply changes that were deleted during compilation.

For more information about exporting changes, refer to *Managing Changes With the Change Manager* in Quartus II Help.

For more information about netlist types and the 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*.

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 *About Quartus II Scripting* in Quartus II Help.

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 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 how 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:

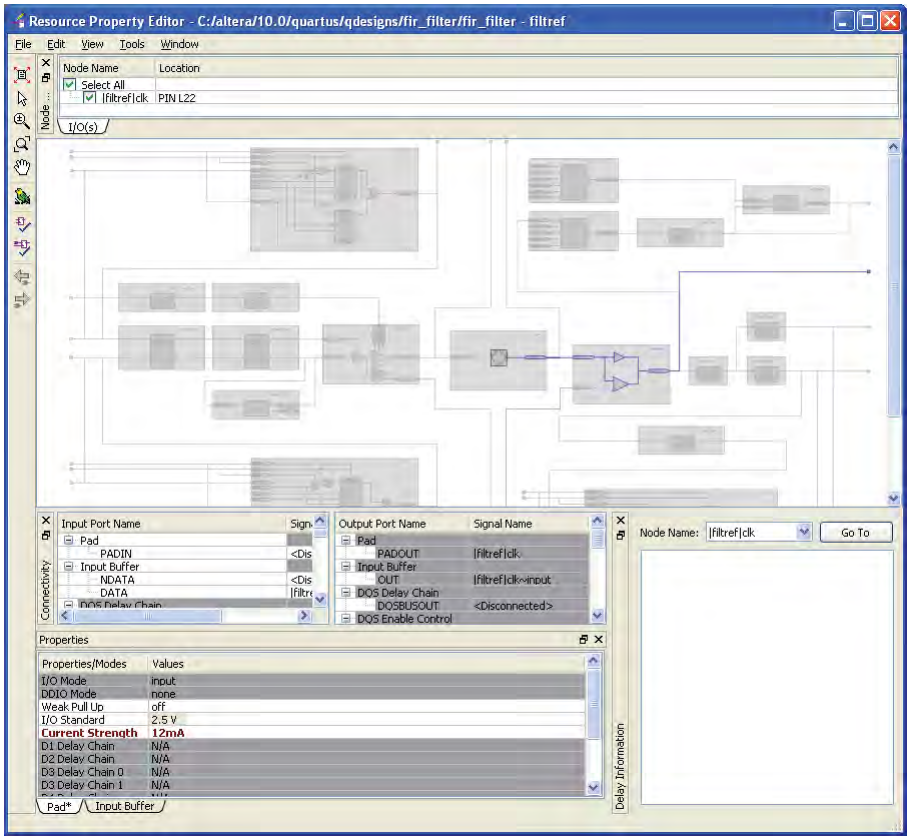
- Adjust the drive strength of an I/O with the Chip Planner
- Modify the PLL properties with the Resource Property Editor (see "Modify the PLL Properties With the Chip Planner" on page 17–29)
- Modify the connectivity between new resource atoms with the Chip Planner and Resource Property Editor

Adjust the Drive Strength of an I/O with 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 17–22.

Figure 17–22. I/O in the Resource Property Editor



3. In the Resource Property Editor, point to the **Current Strength** option in the **Properties** pane and double-click the value to enable the drop-down list.
4. Change the value for the **Current Strength** option.
5. Right-click the ECO change in the Change Manager and click **Check & Save All Netlist Changes** to apply the ECO change.

You can change the pin locations of input or output ports with the ECO flow. 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. You can then click **Check & Save All Netlist Changes** to compile the ECO.

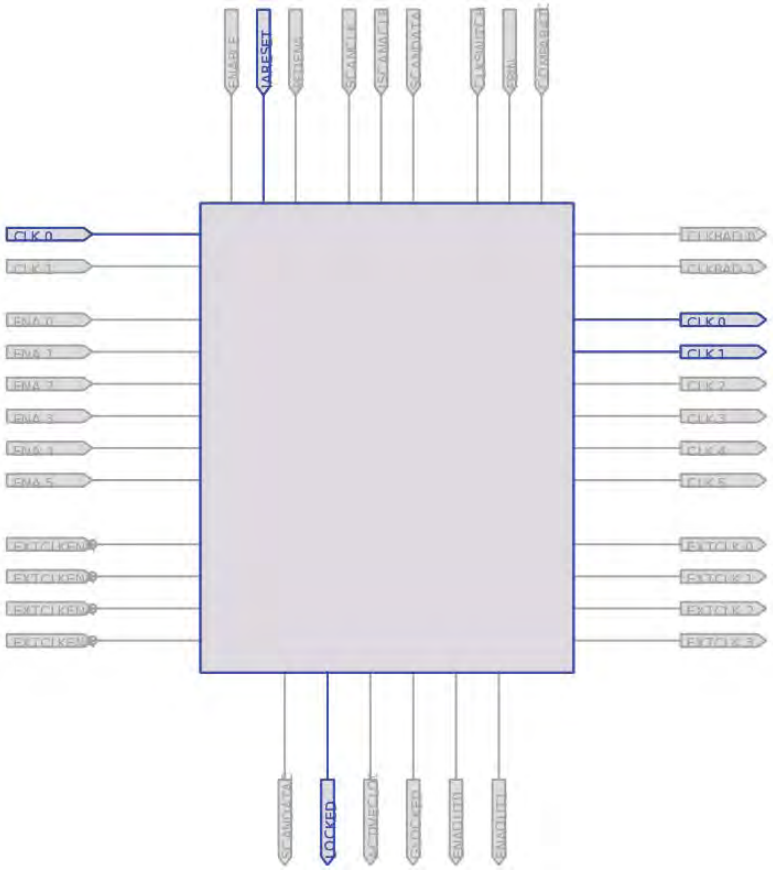
Modify the PLL Properties With the Chip Planner

You use PLLs to modify and generate clock signals to meet design requirements. Additionally, you can use PLLs to distribute 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 17–23 shows a Stratix PLL in the Resource Property Editor.

Figure 17–23. 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

You can also view post-compilation PLL properties in the Compilation Report. To do so, in the Compilation Report, select **Fitter** and then select **Resource Section**.

Adjusting the Duty Cycle

Use Equation 17–1 to adjust the duty cycle of individual output clocks.

Equation 17–1.

$$\text{High \%} = \frac{\text{Counter High}}{(\text{Counter High} + \text{Counter Low})}$$

Adjusting the Phase Shift

Use Equation 17–2 to adjust the phase shift of an output clock of a PLL.

Equation 17–2.

$$\text{Phase Shift} = (\text{Period } V_{CO} \times 0.125 \times \text{Tap } V_{CO}) + (\text{Initial } V_{CO} \times \text{Period } V_{CO})$$

For normal mode Tap V_{CO}, Initial V_{CO}, and Period V_{CO} are governed by the following settings:

$$\text{Tap } V_{CO} = \text{Counter Delay} - \text{M Tap } V_{CO}$$

$$\text{Initial } V_{CO} = \text{Counter Initial} - \text{M Initial}$$

$$\text{Period } V_{CO} = \text{In Clock Period} \times N \div M$$

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, Cyclone, Cyclone II, and Stratix II devices, refer to the appropriate device handbook.

Adjusting the Output Clock Frequency

Use Equation 17–3 to adjust the PLL output clock in normal mode.

Equation 17–3.

$$\text{Output Clock Frequency} = \text{Input Frequency} \bullet \frac{M \text{ value}}{N \text{ Value} + \text{Counter High} + \text{Counter Low}}$$

Use Equation 17–4 to adjust the PLL output clock in external feedback mode.

Equation 17–4.

$$\text{OUTCLK} = \frac{M \text{ value} + \text{External Feedback Counter High} + \text{External Feedback Counter Low}}{N \text{ value} + \text{Counter High} + \text{Counter Low}}$$

Adjusting the Spread Spectrum

Use Equation 17–5 to adjust the spread spectrum for your PLL.

Equation 17–5.

$$\%\ \text{spread} = \frac{M_2 N_1}{M_1 N_2}$$

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 when you are debugging a design, 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 how to create a new resource atom and modify 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. To 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. To 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 Edit Connection dialog box, type the output port name of the register you have created.
11. Run the ECO Fitter to apply the changes by clicking **Check and Save Netlist Changes**.

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 with the tooltips created when you position 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 the TimeQuest analyzer to ensure that your changes did not adversely affect the timing performance of your design.

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 the design still meets all timing requirements, you should perform static timing analysis.

For more information about performing a static timing analysis of your design, refer to the *The Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the Quartus II *Handbook*.

Conclusion

Altera developed the Chip Planner in the Quartus II software suite to assist you in getting your design to the marketplace quickly. 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 speeds design verification and timing closure.

Document Revision History

Table 17–1 shows the revision history for this chapter.

Table 17–1. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Added information to page 17–1. ■ Added information to "Engineering Change Orders" on page 17–2. ■ Changed heading from "Performance" to "Performance Preservation" on page 17–2. ■ Updated information in "Performance Preservation" on page 17–2. ■ Changed heading from "Documentation" to "Change Modification Record" on page 17–3. ■ Changed heading from "Resource Property Editor" to "Performing ECOs in the Resource Property Editor" on page 17–15. ■ Removed "Using Incremental Compilation in the ECO Flow" section. Preservation support for ECOs with the incremental compilation flow has been removed in the Quartus II software version 10.0. ■ Removed "Referenced Documents" section.
November 2009	9.1.0	■ Updated device support list ■ Made minor editorial updates
March 2009	9.0.0	■ Updated Figure 17–17 ■ Made minor editorial updates ■ Chapter 15 was previously Chapter 13 in the 8.1.0 release.
November 2008	8.1.0	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.
May 2008	8.0.0	■ 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

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Development Software Literature web page.

Take an online survey to provide feedback about this handbook chapter.

About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 10.0.

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 10.0 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.
	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 10.0

Volume 3: Verification

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

QII5V3-10.0.1

Copyright © 2010 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 xxiii

Section I. Simulation

Chapter 1. Simulating Designs with EDA Tools

Introduction 1-1
PLD Design Flow 1-2
RTL Functional Simulation Flow 1-3
Gate-Level Timing Simulation Flow 1-4
Converting BDF Format to HDL Format 1-4
Simulation Libraries 1-5
RTL Functional Simulation 1-5
Gate-Level Timing Simulation 1-6
Simulation Netlist Files 1-6
Generating Post-Synthesis Simulation Netlist Files 1-7
Generating Gate-Level Timing Simulation Netlist Files 1-7
Generating Timing Simulation Netlist Files with Different Timing Models 1-8
EDA Simulation Library Compiler 1-9
Running the EDA Simulation Library Compiler Through the GUI 1-10
Running the EDA Simulation Library Compiler from the Command Line 1-11
Using the NativeLink Feature 1-11
Setting Up the EDA Simulator Execution Path 1-11
Configuring NativeLink Settings 1-12
Running RTL Functional Simulation Using the NativeLink Feature 1-14
Running Gate-Level Timing Simulation Using the NativeLink Feature 1-14
Setting Up Testbench Files Using the NativeLink Feature 1-14
Conclusion 1-16
Document Revision History 1-16

Chapter 2. Mentor Graphics ModelSim/QuestaSim Support

Software Requirements 2-1
Design Flow with ModelSim-Altera or ModelSim/QuestaSim Software 2-2
Simulation Libraries 2-2
Precompiled Simulation Libraries in the ModelSim-Altera Software 2-2
Simulation Library Files in the Quartus II Software 2-3
Disabling Timing Violation on Registers 2-3
Performing Simulation Using the ModelSim-Altera Software 2-3
Performing Functional Simulation 2-3
Setting Up a Quartus II Project for the ModelSim-Altera Software 2-4
Compiling and Loading Designs with the ModelSim-Altera Software 2-4
Performing the Simulation 2-4
Performing Post-Synthesis Simulation 2-4
Performing Gate-Level Timing Simulation 2-4
Performing Simulation Using the ModelSim/QuestaSim Software 2-5
Simulating VHDL Designs Using the GUI 2-5
Performing Functional Simulation 2-5
Performing Post-Synthesis Simulation 2-6
Performing Gate-Level Simulation 2-7

Simulating Verilog HDL Designs Using the GUI 2-7
Performing Functional Simulation 2-7
Performing Post-Synthesis Simulation 2-8
Performing Gate-Level Simulation 2-9
Simulating VHDL Designs From the Command Line 2-9
Performing Functional Simulation 2-9
Performing Post-Synthesis Simulation 2-11
Performing Gate-Level Simulation 2-12
Simulating Verilog HDL Designs from the Command Line 2-13
Performing Functional Simulation 2-13
Performing Post-Synthesis Simulation 2-14
Performing Gate-Level Simulation 2-15
Passing Parameter Information from Verilog to VHDL 2-15
Speeding Up Simulation 2-16
Simulating Designs that Include Transceivers 2-16
Functional Simulation for Stratix GX Devices 2-17
Performing Functional Simulation in VHDL (ModelSim-Altera) 2-17
Performing Functional Simulation in VHDL (ModelSim/QuestaSim) 2-17
Performing Functional Simulation in Verilog HDL (ModelSim-Altera) 2-17
Performing Functional Simulation in Verilog HDL (ModelSim/QuestaSim) 2-17
Gate-Level Timing Simulation for Stratix GX Devices 2-18
Performing Gate-Level Timing Simulation in VHDL (ModelSim-Altera) 2-18
Performing Gate-Level Timing Simulation in VHDL (ModelSim/QuestaSim) 2-18
Performing Gate-Level Timing Simulation in Verilog HDL (ModelSim-Altera) 2-18
Performing Gate-Level Timing Simulation in Verilog HDL (ModelSim/QuestaSim) 2-19
Functional Simulation for Stratix II GX Devices 2-19
Performing Functional Simulation in VHDL (ModelSim-Altera) 2-19
Performing Functional Simulation in VHDL (ModelSim/QuestaSim) 2-19
Performing Functional Simulation in Verilog HDL (ModelSim-Altera) 2-20
Performing Functional Simulation in Verilog HDL (ModelSim/QuestaSim) 2-20
Gate-Level Timing Simulation for Stratix II GX Devices 2-20
Performing Gate-Level Timing Simulation in VHDL (ModelSim-Altera) 2-20
Performing Gate-Level Timing Simulation in VHDL (ModelSim/QuestaSim) 2-20
Performing Gate-Level Timing Simulation in Verilog HDL ModelSim-Altera) 2-21
Performing Gate-Level Timing Simulation in Verilog HDL ModelSim/QuestaSim) 2-21
Functional Simulation for Stratix IV GX Devices 2-21
Performing Functional Simulation in VHDL (ModelSim-Altera) 2-21
Performing Functional Simulation in VHDL (ModelSim/QuestaSim) 2-21
Performing Functional Simulation in Verilog HDL (ModelSim-Altera) 2-22
Performing Functional Simulation in Verilog HDL (ModelSim/QuestaSim) 2-22
Gate-Level Timing Simulation for Stratix IV GX Devices 2-22
Performing Gate-Level Timing Simulation in VHDL (ModelSim-Altera) 2-22
Performing Gate-Level Timing Simulation in VHDL (ModelSim/QuestaSim) 2-22
Performing Gate-Level Timing Simulation in Verilog HDL (ModelSim-Altera) 2-23
Performing Gate-Level Timing Simulation in Verilog HDL (ModelSim/QuestaSim) 2-23
Functional Simulation for Stratix V GX Devices 2-23
Performing Functional Simulation in VHDL (ModelSim/QuestaSim) 2-24
Performing Functional Simulation in Verilog HDL (ModelSim-Altera) 2-24
Performing Functional Simulation in Verilog HDL (ModelSim/QuestaSim) 2-24
Transport Delays 2-24
+transport_path_delays 2-24
+transport_int_delays 2-24
Using the NativeLink Feature with ModelSim-Altera or ModelSim/QuestaSim Software 2-24
ModelSim/QuestaSim Error Message Verification 2-25

Generating a Timing Value Change Dump (.vcd) File for the PowerPlay Power Analyzer 2-25
Viewing a Waveform from a .wlf File 2-26
Scripting Support 2-26
Generating a Post-Synthesis Simulation Netlist for ModelSim/QuestaSim 2-27
Tcl Commands 2-27
Command Prompt 2-27
Generating a Gate-Level Timing Simulation Netlist for ModelSim/QuestaSim 2-27
Tcl Commands 2-27
Command Line 2-28
Software Licensing and Licensing Setup in ModelSim-Altera Subscription Edition 2-28
LM_LICENSE_FILE Variable 2-29
Conclusion 2-29
Document Revision History 2-30

Chapter 3. Synopsys VCS and VCS MX Support

Software Requirements 3-1
Using the VCS or VCS MX Software in the Quartus II Design Flow 3-1
Compiling Libraries Using the EDA Simulation Library Compiler 3-2
Functional Simulations 3-2
Functional Simulation for Verilog HDL Designs 3-2
Functional Simulation for VHDL Designs 3-3
Post-Synthesis Simulation 3-4
Post-Synthesis Simulation for Verilog HDL Designs 3-5
Post-Synthesis Simulation for VHDL Designs 3-5
Gate-Level Timing Simulation 3-6
Gate-Level Timing Simulation for Verilog HDL Designs 3-6
Gate-Level Timing Simulation for VHDL Designs 3-6
Disabling Timing Violation on Registers 3-7
Performing Timing Simulation Using the Post-Synthesis Netlist 3-7
Common VCS and VCS MX Software Compiler Options 3-8
Using DVE 3-8
Debugging Support Command-Line Interface 3-8
Simulating Designs that Include Transceivers 3-9
Functional Simulation for Stratix GX Devices 3-9
Compiling Library Files for Functional Simulation in Verilog HDL 3-9
Gate-Level Timing Simulation for Stratix GX Devices 3-9
Compiling Library Files for Gate-Level Timing Simulation in Verilog HDL 3-10
Functional Simulation for Stratix II GX Devices 3-10
Compiling Library Files for Functional Simulation in Verilog HDL 3-10
Gate-Level Timing Simulation for Stratix II GX Devices 3-10
Compiling Library Files for Gate-Level Timing Simulation in Verilog HDL 3-11
Functional Simulation for Stratix IV GX Devices 3-11
Compiling Library Files for Functional Simulation in Verilog HDL 3-11
Gate-Level Timing Simulation for Stratix IV GX Devices 3-11
Compiling Library Files for Gate-Level Timing Simulation in Verilog HDL 3-12
Functional Simulation for Stratix V GX Devices 3-12
Compiling Library Files for Functional Simulation 3-12
Transport Delays 3-13
+transport_path_delays 3-13
+transport_int_delays 3-13
Using NativeLink with the VCS or VCS MX Software 3-13
Generating a Timing .vcd File for the PowerPlay Power Analyzer 3-13
Viewing a Waveform from a .vpd or .vcd File 3-14
Scripting Support 3-15

Generating a Post-Synthesis Simulation Netlist for VCS . . . 3-15
Tcl Commands . . . 3-15
Command Prompt . . . 3-15
Generating a Gate-Level Timing Simulation Netlist for VCS . . . 3-15
Tcl Commands . . . 3-15
Command Prompt . . . 3-15
Conclusion . . . 3-16
Document Revision History . . . 3-16

Chapter 4. Cadence NC-Sim Support

Software Requirements . . . 4-1
Simulation Flow Overview . . . 4-1
Operation Modes . . . 4-2
Quartus II Software and NC Simulation Flow Overview . . . 4-2
Compiling Libraries Using the EDA Simulation Library Compiler . . . 4-3
Functional Simulation . . . 4-3
Creating Libraries . . . 4-4
For VHDL Designs . . . 4-4
For Verilog HDL Designs . . . 4-4
Compiling Source Code . . . 4-5
Elaborating Your Design . . . 4-6
Simulating Your Design . . . 4-6
Post-Synthesis Simulation . . . 4-7
Quartus II Simulation Output Files . . . 4-7
Creating Libraries . . . 4-7
Compiling Project Files and Libraries . . . 4-7
Elaborating Your Design . . . 4-7
Simulating Your Design . . . 4-8
Gate-Level Timing Simulation . . . 4-8
Generating a Gate-Level Timing Simulation Netlist . . . 4-8
Disabling Timing Violation on Registers . . . 4-8
Creating Libraries . . . 4-9
Compiling Project Files and Libraries . . . 4-9
Elaborating Your Design . . . 4-9
Compiling the .sdo File (VHDL Only) in Command-Line Mode . . . 4-10
Compiling the .sdo File (VHDL Only) in GUI Mode . . . 4-10
Simulating Your Design . . . 4-10
Simulating Designs that Include Transceivers . . . 4-10
Functional Simulation for Stratix GX Devices . . . 4-11
Compiling Library Files for Functional Simulation in VHDL . . . 4-11
Compiling Library Files for Functional Simulation in Verilog HDL . . . 4-11
Gate-Level Timing Simulation for Stratix GX Devices . . . 4-11
Compiling Library Files for Gate-Level Timing Simulation in VHDL . . . 4-11
Compiling Library Files for Gate-Level Timing Simulation in Verilog HDL . . . 4-12
Functional Simulation for Stratix II GX Devices . . . 4-12
Compiling Library Files for Functional Simulation in VHDL . . . 4-13
Compiling Library Files for Functional Simulation in Verilog HDL . . . 4-13
Gate-Level Timing Simulation for Stratix II GX Devices . . . 4-13
Compiling Library Files for Gate-Level Timing Simulation in VHDL . . . 4-14
Compiling Library Files for Gate-Level Timing Simulation in Verilog HDL . . . 4-14
Functional Simulation for Stratix IV GX Devices . . . 4-14
Compiling Library Files for Functional Simulation in VHDL . . . 4-15
Compiling Library Files for Functional Simulation in Verilog HDL . . . 4-15
Gate-Level Timing Simulation for Stratix IV GX Devices . . . 4-15

Compiling Library Files for Gate-Level Timing Simulation in VHDL 4-16
Compiling Library Files for Gate-Level Timing Simulation in Verilog HDL 4-16
Functional Simulation for Stratix V GX Devices 4-16
Compiling Library Files for Functional Simulation 4-17
Compiling Library Files for Functional Simulation in VHDL 4-17
Compiling Library Files for Functional Simulation in Verilog HDL 4-17
Pulse Reject Delays 4-18
-PULSE_R 4-18
-PULSE_INT_R 4-18
Using the NativeLink Feature with NC-Sim 4-18
Generating a Timing VCD File for the PowerPlay Power Analyzer 4-18
Viewing a Waveform from a .trn File 4-19
Scripting Support 4-20
Generating NC-Sim Simulation Output Files 4-20
Tcl Commands 4-20
Command Prompt 4-21
Conclusion 4-21
Document Revision History 4-21

Chapter 5. Aldec Active-HDL and Riviera-PRO Support

Software Requirements 5-1
Using Active-HDL or Riviera-PRO Software in Quartus II Design Flows 5-1
Simulation Libraries 5-2
Simulation Library Files in the Quartus II Software 5-2
Disabling Timing Violation on Registers 5-2
Compiling Libraries Using the EDA Simulation Library Compiler 5-2
Performing Simulation Using the Active-HDL Software (GUI Mode) 5-3
Simulating VHDL Designs 5-3
Performing Functional Simulation 5-3
Simulating Verilog HDL Designs 5-4
Performing Functional Simulation 5-4
Performing Simulation Using the Riviera-PRO Software (GUI Mode) 5-5
Performing Simulation Using the Active-HDL or Riviera-PRO Software (Command-Line Mode) . . . 5-6
Simulating Verilog HDL Designs Using the Active-HDL or Riviera-PRO Software 5-6
Performing Functional Simulation 5-7
Performing Post-Synthesis Simulation 5-7
Performing Gate-Level Timing Simulation 5-8
Simulating VHDL Designs Using the Active-HDL or Riviera-PRO Software 5-8
Performing Functional Simulation 5-8
Performing Post-Synthesis Simulation 5-9
Performing Gate-Level Timing Simulation 5-10
Compiling System Verilog Files 5-10
Simulating Designs that Include Transceivers 5-10
Functional Simulation for Stratix II GX Devices 5-11
Performing Functional Simulation in VHDL 5-11
Performing Functional Simulation in Verilog HDL 5-12
Gate-Level Timing Simulation for Stratix II GX Devices 5-12
Performing Gate-Level Timing Simulation in VHDL 5-12
Performing Gate-Level Timing Simulation in Verilog HDL 5-12
Functional Simulation for Stratix GX Devices 5-12
Performing Functional Simulation in VHDL 5-13
Performing Functional Simulation in Verilog HDL 5-13
Gate-Level Timing Simulation for Stratix GX Devices 5-13
Performing Gate-Level Timing Simulation in VHDL 5-13

Performing Gate-Level Timing Simulation in Verilog HDL 5-14
Functional Simulation for Stratix IV GX Devices 5-14
Performing Functional Simulation in VHDL 5-14
Performing Functional Simulation in Verilog HDL 5-15
Gate-Level Timing Simulation for Stratix IV GX Devices 5-15
Performing Gate-Level Timing Simulation in VHDL 5-15
Performing Gate-Level Timing Simulation in Verilog HDL 5-15
Functional Simulation for Stratix V GX Devices 5-16
Performing Functional Simulation in VHDL 5-16
Performing Functional Simulation in Verilog HDL 5-16
Transport Delays 5-17
Using the NativeLink Feature in Active-HDL or Riviera-PRO Software 5-17
Generating .vcd Files for the PowerPlay Power Analyzer 5-17
Scripting Support 5-18
Generating a Post-Synthesis Simulation Netlist for Active-HDL or Riviera-PRO 5-18
Tcl Commands 5-18
Command Line 5-19
Generating a Gate-Level Timing Simulation Netlist for Active-HDL or Riviera-PRO 5-19
Tcl Commands 5-19
Command Line 5-19
Conclusion 5-19
Document Revision History 5-20

Chapter 6. Simulating Altera IP in Third-Party Simulation Tools
IP Functional Simulation Flow 6-2
Verilog HDL and VHDL IPFS Models 6-3
Instantiate the IP in Your Design 6-4
Perform Functional Simulation 6-4
Simulating Altera IP Using the Quartus II NativeLink Feature 6-4
Perform Analysis and Elaboration on Your Design 6-5
Run Simulation Using the Quartus II NativeLink Feature 6-5
Simulating Altera IP Without the Quartus II NativeLink Feature 6-5
Using the EDA Simulation Library Compiler 6-6
Design Language Examples 6-6
Verilog HDL Example: Simulating the IPFS Model in the ModelSim SE Software 6-6
VHDL Example: Simulating the IPFS Model in the ModelSim SE Software 6-8
Conclusion 6-9
Document Revision History 6-10

Section II. Timing Analysis

Chapter 7. The Quartus II TimeQuest Timing Analyzer
Getting Started with the TimeQuest Timing Analyzer 7-2
Running the TimeQuest Timing Analyzer 7-3
Running the TimeQuest Timing Analyzer from the Quartus II GUI 7-3
Running the TimeQuest Timing Analyzer as a Stand-Alone Application 7-4
Running the quartus_sta Executable at a System Command Prompt 7-4
Timing Analysis Overview 7-5
Open the Project 7-6
Create a Timing Netlist 7-6
Constrain the Design 7-6
Update Timing Netlist 7-7
Generate Timing Reports 7-7

The Timing Netlist . . . 7-8
Clock Analysis . . . 7-11
Clock Setup Check . . . 7-11
Clock Hold Check . . . 7-12
Recovery and Removal . . . 7-13
Multicycle Paths . . . 7-14
Metastability . . . 7-16
Common Clock Path Pessimism . . . 7-17
Clock-As-Data . . . 7-19
Collections . . . 7-20
Adding and Removing Collection Items . . . 7-21
SDC Constraint Files . . . 7-22
Fitter and Timing Analysis with SDC Files . . . 7-22
Specifying SDC Files for Place-and-Route . . . 7-22
Specifying SDC Files for Static Timing Analysis . . . 7-23
Synopsys Design Constraints File Precedence . . . 7-23
Clock Specification . . . 7-23
Generated Clocks . . . 7-24
Virtual Clocks . . . 7-26
Multi-Frequency Clocks . . . 7-27
Automatic Clock Detection . . . 7-28
Derive PLL Clocks . . . 7-28
Default Clock Constraints . . . 7-30
Clock Groups . . . 7-30
Clock Effect Characteristics . . . 7-32
Clock Latency . . . 7-32
Clock Uncertainty . . . 7-33
Intra-Clock Transfers . . . 7-34
Inter-Clock Transfers . . . 7-34
I/O Interface Clock Transfers . . . 7-34
I/O Specifications . . . 7-35
Input and Output Delay . . . 7-36
Set Input Delay . . . 7-36
Set Output Delay . . . 7-37
Delay and Skew Specifications . . . 7-38
set_net_delay . . . 7-38
set_max_skew . . . 7-38
Timing Exceptions . . . 7-39
Precedence . . . 7-39
False Path . . . 7-39
Minimum Delay . . . 7-39
Maximum Delay . . . 7-40
Multicycle Path . . . 7-41
Delay Annotation . . . 7-42
Constraint and Exception Removal . . . 7-43
Timing Reports . . . 7-43
report_timing . . . 7-44
report_exceptions . . . 7-45
report_metastability . . . 7-46
report_clock_transfers . . . 7-46
report_clocks . . . 7-47
report_min_pulse_width . . . 7-47
report_net_timing . . . 7-48
report_sdc . . . 7-48

report_ucp . . . 7-49
report_bottleneck . . . 7-49
report_datasheet . . . 7-50
report_rskm . . . 7-50
report_tccs . . . 7-51
report_partitions . . . 7-51
report_path . . . 7-51
report_net_delay . . . 7-53
report_max_skew . . . 7-53
report_skew . . . 7-54
check_timing . . . 7-55
report_clock_fmax_summary . . . 7-55
create_timing_summary . . . 7-56
Other Features . . . 7-57
Multi-Corner Analysis . . . 7-57
Advanced I/O Timing and Board Trace Model Assignments . . . 7-59
Wildcard Assignments and Collections . . . 7-59
Resetting a Design . . . 7-60
Cross-Probing . . . 7-61
locate . . . 7-61
The Quartus II Software Options and Compilation Report . . . 7-62
Conclusion . . . 7-63
Document Revision History . . . 7-63

Chapter 8. Best Practices for the Quartus II TimeQuest Timing Analyzer
Clock Requirements . . . 8-1
Base Clocks . . . 8-1
Derived Clocks . . . 8-2
Virtual Clocks . . . 8-2
I/O Requirements . . . 8-4
Input Requirements . . . 8-4
Output Requirements . . . 8-5
Exceptions . . . 8-5
False Paths . . . 8-5
Minimum and Maximum Delays . . . 8-6
Multicycles . . . 8-6
Conclusion . . . 8-7
Document Revision History . . . 8-7

Chapter 9. Switching to the Quartus II TimeQuest Timing Analyzer
Benefits of Switching to the Quartus II TimeQuest Timing Analyzer . . . 9-1
Chapter Contents . . . 9-1
Switching Your Design . . . 9-2
Compile Your Design . . . 9-2
Create an .sdc File . . . 9-2
Conversion Utility . . . 9-2
Start the Quartus II TimeQuest Timing Analyzer . . . 9-3
Set the Default Timing Analyzer . . . 9-3
Differences Between the Timing Analyzers . . . 9-3
Terminology . . . 9-4
Netlists . . . 9-4
Collections . . . 9-5
Constraints . . . 9-5

Constraint Files 9-5
Constraint Entry 9-6
Constraint File Priority 9-7
Constraint Priority 9-9
Ambiguous Constraints 9-9
Clocks 9-10
Related and Unrelated Clocks 9-10
Clock Offset 9-11
Clock Latency 9-11
Offset and Latency Example 9-11
Clock Uncertainty 9-13
Derived and Generated Clocks 9-14
Automatic Clock Detection 9-15
Hold Relationship 9-17
Clock Objects 9-18
Hold Multicycle 9-18
Fitter Behavior 9-20
Fitter Performance 9-20
Reporting 9-20
Paths and Pairs 9-21
Default Reports 9-21
Netlist Names 9-21
Non-Integer Clock Periods 9-22
Other Features 9-23
Timing Assignment Conversion 9-24
Setup Relationship 9-24
Hold Relationship 9-25
Clock Latency 9-25
Clock Uncertainty 9-25
Inverted Clock 9-25
Not a Clock 9-26
Default Required f_{MAX} Assignment 9-26
Virtual Clock Reference 9-26
Clock Settings 9-27
Multicycle 9-27
Clock Enable Multicycle 9-28
I/O Constraints 9-28
Input and Output Delay 9-29
t_{SU} Requirement 9-29
t_H Requirement 9-31
t_{CO} Requirement 9-33
Minimum t_{CO} Requirement 9-35
t_{PD} Requirement 9-37
Minimum t_{PD} Requirement 9-38
Cut Timing Path 9-38
Maximum Delay 9-38
Minimum Delay 9-39
Maximum Clock Arrival Skew 9-39
Maximum Data Arrival Skew 9-39
Constraining Skew on an Output Bus 9-39
Conversion Utility 9-41
Unsupported Global Assignments 9-41
Recommended Global Assignments 9-41
Clock Conversion 9-42

Instance Assignment Conversion 9-43
PLL Phase Shift Conversion 9-45
t_{CO} Requirement Conversion 9-46
Entity-Specific Assignments 9-46
Paths Between Unrelated Clock Domains 9-47
Unsupported Instance Assignments 9-47
Reviewing Conversion Results 9-47
Warning Messages 9-47
Clocks 9-49
Clock Transfers 9-49
Path Details 9-49
Unconstrained Paths 9-50
Bus Names 9-50
Other 9-50
Re-Running the Conversion Utility 9-50
Notes 9-50
Output Pin Load Assignments 9-50
Constraint Target Types 9-51
DDR Constraints with the DDR Timing Wizard 9-51
Unsupported SDC Features 9-51
Constraint Passing and Optimization 9-51
Clock Network Delay Reporting 9-51
Project Management 9-52
Conversion Utility 9-52
t_{PD} and Minimum t_{PD} Requirement Conversion 9-52
Document Revision History 9-52

Chapter 10. Quartus II Classic Timing Analyzer

Timing Analysis Tool Setup 10-2
Static Timing Analysis Overview 10-2
Clock Analysis 10-4
Clock Setup Check 10-4
Clock Hold Check 10-6
Multicycle Paths 10-7
Clock Settings 10-8
Individual Clock Settings 10-8
Default Clock Settings 10-8
Clock Types 10-9
Base Clocks 10-9
Derived Clocks 10-9
Undefined Clocks 10-9
PLL Clocks 10-10
Clock Uncertainty 10-10
Clock Latency 10-11
Timing Exceptions 10-13
Multicycle 10-14
Destination Multicycle Setup Exception 10-14
Destination Multicycle Hold Exception 10-14
Source Multicycle Setup Exception 10-15
Source Multicycle Hold Exception 10-16
Default Hold Multicycle 10-16
Clock Enable Multicycle 10-16
Setup Relationship and Hold Relationship 10-19
Maximum Delay and Minimum Delay 10-20

False Paths . . . 10-20
I/O Analysis . . . 10-22
External Input Delay and Output Delay Assignments . . . 10-22
Input Delay Assignment . . . 10-22
Output Delay Assignment . . . 10-23
Virtual Clocks . . . 10-24
Asynchronous Paths . . . 10-25
Recovery and Removal . . . 10-25
Recovery Report . . . 10-25
Removal Report . . . 10-27
Skew Management . . . 10-29
Maximum Clock Arrival Skew . . . 10-29
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 . . . 10-32
Fast Corner Analysis . . . 10-33
Early Timing Estimation . . . 10-33
Timing Constraint Checker . . . 10-34
Latch Analysis . . . 10-34
Timing Analysis Using the Quartus II GUI . . . 10-35
Assignment Editor . . . 10-35
Timing Settings . . . 10-36
Clock Settings Dialog Box . . . 10-36
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 . . . 10-39
Creating Clocks . . . 10-39
Base Clocks . . . 10-39
Derived Clocks . . . 10-39
Clock Latency . . . 10-39
Clock Uncertainty . . . 10-40
Cut Timing Paths . . . 10-40
Input Delay Assignment . . . 10-40
Maximum and Minimum Delay . . . 10-41
Maximum Clock Arrival Skew . . . 10-41
Maximum Data Arrival Skew . . . 10-41
Multicycle . . . 10-41
Output Delay Assignment . . . 10-42
Report Timing . . . 10-42
Setup and Hold Relationships . . . 10-42
Assignment Group . . . 10-43
Virtual Clock . . . 10-43
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}) 10-47
Conclusion 10-47
Document Revision History 10-47

Chapter 11. Synopsys PrimeTime Support

Quartus II Settings for Generating the PrimeTime Software Files 11-1
Files Generated for the PrimeTime Software Environment 11-2
The Netlist 11-2
The .sdo File 11-2
Generating Multiple Operating Conditions with the TimeQuest Analyzer 11-3
The Tcl Script 11-4
Generated File Summary 11-6
Running the PrimeTime Software 11-7
Analyzing Quartus II Projects 11-7
Other pt_shell Commands 11-7
PrimeTime Timing Reports 11-8
Sample PrimeTime Software Timing Report 11-8
Comparing Timing Reports from the Classic Timing Analyzer and the PrimeTime Software . . . 11-8
Clock Setup Relationship and Slack 11-9
Clock Hold Relationship and Slack 11-12
Input Delay and Output Delay Relationships and Slack 11-15
Static Timing Analyzer Differences 11-17
Classic Timing Analyzer and PrimeTime Software 11-17
Rise/Fall Support 11-17
Minimum and Maximum Delays 11-17
Recovery/Removal Analysis 11-17
Encrypted Intellectual Property Blocks 11-17
Registered Clock Signals 11-18
Multiple Source and Destination Register Pairs 11-18
Latches 11-18
LVDS I/O 11-19
Clock Latency 11-19
Input and Output Delay Assignments 11-19
Generated Clocks Derived from Generated Clocks 11-19
TimeQuest Timing Analyzer and PrimeTime Software 11-19
Encrypted Intellectual Property Blocks 11-19
Latches 11-20
LVDS I/O 11-20
The TimeQuest Timing Analyzer .sdc File and PrimeTime Compatibility 11-20
Clock and Data Paths 11-20
Inverting and Non-Inverting Propagation 11-20
Multiple Rise/Fall Numbers For a Timing Arc 11-20
Virtual Generated Clocks 11-21
Generated Clocks Derived from Generated Clocks 11-21
Conclusion 11-21
Document Revision History 11-21

Section III. Power Estimation and Analysis

Chapter 12. PowerPlay Power Analysis

Creating PowerPlay EPE Spreadsheets . . . 12-2
PowerPlay EPE File Generator Compilation Report . . . 12-3
Types of Power Analysis . . . 12-4
Factors Affecting Power Consumption . . . 12-4
Device Selection . . . 12-4
Environmental Conditions . . . 12-5
Airflow . . . 12-5
Heat Sink and Thermal Compound . . . 12-5
Junction Temperature . . . 12-5
Board Thermal Model . . . 12-5
Device Resource Usage . . . 12-6
Number, Type, and Loading of I/O Pins . . . 12-6
Number and Type of Logic Elements, Multiplier Elements, and RAM Blocks . . . 12-6
Number and Type of Global Signals . . . 12-6
Signal Activities . . . 12-6
PowerPlay Power Analyzer Flow . . . 12-7
Operating Conditions . . . 12-7
Signal Activities Data Sources . . . 12-9
Simulation Results . . . 12-9
Using Simulation Files in Modular Design Flows . . . 12-10
Complete Design Simulation . . . 12-11
Modular Design Simulation . . . 12-12
Multiple Simulations on the Same Entity . . . 12-12
Overlapping Simulations . . . 12-13
Partial Simulations . . . 12-13
Node Name Matching Considerations . . . 12-13
Glitch Filtering . . . 12-14
Node and Entity Assignments . . . 12-15
Timing Assignments to Clock Nodes . . . 12-16
Default Toggle Rate Assignment . . . 12-16
Vectorless Estimation . . . 12-16
Using the PowerPlay Power Analyzer . . . 12-17
Common Analysis Flows . . . 12-17
Signal Activities from Full Post-Fit Netlist (Timing) Simulation . . . 12-17
Signal Activities from Full Post-Fit Netlist (Zero Delay) Simulation . . . 12-17
Signal Activities from RTL (Functional) Simulation, Supplemented by Vectorless Estimation . . . 12-17
Signal Activities from Vectorless Estimation and User-Supplied Input Pin Activities . . . 12-18
Signal Activities from User Defaults Only . . . 12-18
Generating a .vcd . . . 12-18
Generating a .vcd from ModelSim Software . . . 12-19
Generating a .vcd from Full Post-Fit Netlist (Zero Delay) Simulation . . . 12-20
Running the PowerPlay Power Analyzer Using the Quartus II GUI . . . 12-20
PowerPlay Power Analyzer Compilation Report . . . 12-23
Summary . . . 12-23
Settings . . . 12-23
Simulation Files Read . . . 12-24
Operating Conditions Used . . . 12-24
Thermal Power Dissipated by Block . . . 12-24
Thermal Power Dissipation by Block Type (Device Resource Type) . . . 12-24

Thermal Power Dissipation by Hierarchy 12-24
Core Dynamic Thermal Power Dissipation by Clock Domain 12-24
Current Drawn from Voltage Supplies 12-24
Confidence Metric Details 12-25
Signal Activities 12-25
Messages 12-25
Specific Rules for Reporting 12-25
Scripting Support 12-26
Running the PowerPlay Power Analyzer from the Command–Line 12-26
Conclusion 12-27
Document Revision History 12-28

Section IV. System Debugging Tools

Chapter 13. System Debugging Tools Overview

System Debugging Tools 13-1
Analysis Tools for RTL Nodes 13-3
Resource Usage 13-4
Pin Usage 13-5
Usability Enhancements 13-6
Stimulus-Capable Tools 13-8
In-System Sources and Probes 13-8
In-System Memory Content Editor 13-8
Virtual JTAG Interface Megafunction 13-9
System Console 13-9
Conclusion 13-9
Document Revision History 13-10

Chapter 14. Analyzing and Debugging Designs with the System Console

Setting Up the System Console 14-1
Interactive Help 14-1
Using the System Console 14-2
SOPC Builder Communications 14-2
Console Commands 14-4
Plug-ins 14-5
Design Service Commands 14-6
Data Pattern Generator Commands 14-6
Data Pattern Checker Commands 14-8
Programmable Logic Device (PLD) Commands 14-9
Board Bring-Up Commands 14-9
JTAG Debug Commands 14-9
Clock and Reset Signal Commands 14-10
Avalon-MM Commands 14-10
Processor Commands 14-12
Bytestream Commands 14-12
Transceiver Toolkit Commands 14-12
System Console Examples 14-17
LED Light Show Example 14-18
JTAG Examples 14-20
Verify JTAG Chain 14-20
Verify Clock 14-21
Checksum Example 14-21
Nios II Processor Example 14-24

Device Support 14-26
Conclusion 14-26
Document Revision History 14-26

Chapter 15. Transceiver Link Debugging Using the Quartus II Software
Transceiver Toolkit Overview 15-1
Auto Sweep 15-1
EyeQ 15-2
Control Links 15-2
Transceiver Link Debugging Examples 15-2
Test Setup for Link Debugging 15-2
Custom_phy 15-4
Data Pattern Generator 15-4
Data Checker 15-5
Compiling Design Examples 15-6
Changing Pin Assignments 15-6
Transceiver Toolkit Link Test Setup 15-7
Loading the Project in System Console 15-7
Linking the Hardware Resource 15-7
Creating the Channels 15-8
Running the Link Tests 15-9
Viewing Results in the EyeQ Feature 15-9
Tcl Script in System Console 15-11
Running Tcl Scripts 15-11
. . . . . .Conclusion 15-12
Document Revision History 15-12

Chapter 16. Quick Design Debugging Using SignalProbe
Debugging Using the SignalProbe Feature 16-1
Reserve the SignalProbe Pins 16-1
Perform a Full Compilation 16-2
Assign a SignalProbe Source 16-2
Add Registers to the Pipeline Path to SignalProbe Pin 16-2
Perform a SignalProbe Compilation 16-3
Analyze the Results of the SignalProbe Compilation 16-3
Performing a SignalProbe Compilation 16-4
Understanding the Results of a SignalProbe Compilation 16-4
Analyzing SignalProbe Routing Failures 16-6
Scripting Support 16-6
Make a SignalProbe Pin 16-6
Delete a SignalProbe Pin 16-7
Enable a SignalProbe Pin 16-7
Disable a SignalProbe Pin 16-7
Perform a SignalProbe Compilation 16-7
Script Example 16-7
Reserving SignalProbe Pins 16-7
Common Problems When Reserving a SignalProbe Pin 16-7
Adding SignalProbe Sources 16-8
Assigning I/O Standards 16-8
Adding Registers for Pipelining 16-8
Run SignalProbe Automatically 16-9
Run SignalProbe Manually 16-9
Enable or Disable All SignalProbe Routing 16-9

Allow SignalProbe to Modify Fitting Results . . . 16-9
Conclusion . . . 16-9
Document Revision History . . . 16-10

Chapter 17. Design Debugging Using the SignalTap II Logic Analyzer
Hardware and Software Requirements . . . 17-2
Design Flow Using the SignalTap II Logic Analyzer . . . 17-4
SignalTap II Logic Analyzer Task Flow . . . 17-4
Add the SignalTap II Logic Analyzer to Your Design . . . 17-5
Configure the SignalTap II Logic Analyzer . . . 17-5
Define Trigger Conditions . . . 17-6
Compile the Design . . . 17-6
Program the Target Device or Devices . . . 17-6
Run the SignalTap II Logic Analyzer . . . 17-6
View, Analyze, and Use Captured Data . . . 17-7
Embedding Multiple Analyzers in One FPGA . . . 17-7
Monitoring FPGA Resources Used by the SignalTap II Logic Analyzer . . . 17-7
Using the MegaWizard Plug-In Manager to Create Your Logic Analyzer . . . 17-8
Configure the SignalTap II Logic Analyzer . . . 17-8
Assigning an Acquisition Clock . . . 17-8
Adding Signals to the SignalTap II File . . . 17-9
Signal Preservation . . . 17-10
Assigning Data Signals Using the Technology Map Viewer . . . 17-11
Node List Signal Use Options . . . 17-11
Untappable Signals . . . 17-11
Adding Signals with a Plug-In . . . 17-12
Adding Finite State Machine State Encoding Registers . . . 17-13
Modifying and Restoring Mnemonic Tables for State Machines . . . 17-13
Additional Considerations . . . 17-14
Specifying the Sample Depth . . . 17-14
Capturing Data to a Specific RAM Type . . . 17-14
Choosing the Buffer Acquisition Mode . . . 17-14
Non-Segmented Buffer . . . 17-15
Segmented Buffer . . . 17-16
Using the Storage Qualifier Feature . . . 17-16
Input Port Mode . . . 17-19
Transitional Mode . . . 17-20
Conditional Mode . . . 17-21
Start/Stop Mode . . . 17-22
State-Based . . . 17-23
Showing Data Discontinuities . . . 17-23
Disable Storage Qualifier . . . 17-23
Managing Multiple SignalTap II Files and Configurations . . . 17-24
Define Triggers . . . 17-25
Creating Basic Trigger Conditions . . . 17-25
Creating Advanced Trigger Conditions . . . 17-26
Examples of Advanced Triggering Expressions . . . 17-27
Trigger Condition Flow Control . . . 17-28
Sequential Triggering . . . 17-28
State-Based Triggering . . . 17-29
SignalTap II Trigger Flow Description Language . . . 17-32
State Labels . . . 17-33
Boolean_expression . . . 17-33
Action_list . . . 17-34

Resource Manipulation Action 17-34
Buffer Control Action 17-34
State Transition Action 17-35
Using the State-Based Storage Qualifier Feature 17-35
Specifying the Trigger Position 17-39
Creating a Power-Up Trigger 17-40
Enabling a Power-Up Trigger 17-40
Managing and Configuring Power-Up and Runtime Trigger Conditions 17-41
Using External Triggers 17-42
Using the Trigger Out of One Analyzer as the Trigger In of Another Analyzer 17-42
Compile the Design 17-44
Faster Compilations with Quartus II Incremental Compilation 17-44
Enabling Incremental Compilation for Your Design 17-45
Using Incremental Compilation with the SignalTap II Logic Analyzer 17-45
Preventing Changes Requiring Recompilation 17-47
Timing Preservation with the SignalTap II Logic Analyzer 17-48
Performance and Resource Considerations 17-48
Program the Target Device or Devices 17-49
Querying the SignalTap II Instance on a Programmed Device 17-50
Run the SignalTap II Logic Analyzer 17-52
Runtime Reconfigurable Options 17-54
SignalTap II Status Messages 17-56
View, Analyze, and Use Captured Data 17-57
Capturing Data Using Segmented Buffers 17-57
Differences in Pre-fill Write Behavior Between Different Acquisition Modes 17-59
Creating Mnemonics for Bit Patterns 17-60
Automatic Mnemonics with a Plug-In 17-61
Locating a Node in the Design 17-61
Saving Captured Data 17-62
Exporting Captured Data to Other File Formats 17-62
Creating a SignalTap II List File 17-62
Other Features 17-63
Using the SignalTap II MATLAB MEX Function to Capture Data 17-63
Using SignalTap II in a Lab Environment 17-64
Remote Debugging Using the SignalTap II Logic Analyzer 17-64
Equipment Setup 17-65
Using the SignalTap II Logic Analyzer in Devices with Configuration Bitstream Security 17-65
Backward Compatibility with Previous Versions of Quartus II Software 17-65
SignalTap II Scripting Support 17-66
SignalTap II Command-Line Options 17-66
SignalTap II Tcl Commands 17-67
Design Example: Using SignalTap II Logic Analyzers in SOPC Builder Systems 17-68
Custom Triggering Flow Application Examples 17-69
Design Example 1: Specifying a Custom Trigger Position 17-69
Design Example 2: Trigger When triggercond1 Occurs Ten Times between triggercond2 and triggercond3 17-70
Conclusion 17-71
Document Revision History 17-71

Chapter Revision Dates

The chapters in this book, *Quartus II Handbook Version 10.0 Volume 3: Verification*, were revised on the following dates. Where chapters or groups of chapters are available separately, part numbers are listed.

Chapter 1 Simulating Designs with EDA Tools
Revised: *July 2010*
Part Number: *QII53025-10.0.0*

Chapter 2 Mentor Graphics ModelSim / QuestaSim Support
Revised: *July 2010*
Part Number: *QII53001-10.0.0*

Chapter 3 Synopsys VCS and VCS MX Support
Revised: *July 2010*
Part Number: *QII53002-10.0.0*

Chapter 4 Cadence NC-Sim Support
Revised: *July 2010*
Part Number: *QII53003-10.0.0*

Chapter 5 Aldec Active-HDL and Riviera-PRO Support
Revised: *July 2010*
Part Number: *QII53023-10.0.0*

Chapter 6 Simulating Altera IP in Third-Party Simulation Tools
Revised: *August 2010*
Part Number: *QII53014-10.0.1*

Chapter 7 The Quartus II TimeQuest Timing Analyzer
Revised: *July 2010*
Part Number: *QII53018-10.0.0*

Chapter 8 Best Practices for the Quartus II TimeQuest Timing Analyzer
Revised: *July 2010*
Part Number: *QII53024-10.0.0*

Chapter 10 Quartus II Classic Timing Analyzer
Revised: *July 2010*
Part Number: *QII53004-10.0.0*

Chapter 11 Synopsys PrimeTime Support
Revised: *July 2010*
Part Number: *QII53005-10.0.0*

Chapter 12 PowerPlay Power Analysis
Revised: *July 2010*
Part Number: *QII53013-10.0.0*

Chapter 13 System Debugging Tools Overview
Revised: *July 2010*
Part Number: *QII53027-10.0.0*

Chapter 14 Analyzing and Debugging Designs with the System Console
Revised: *July 2010*
Part Number: *QII53028-10.0.0*

Chapter 15 Transceiver Link Debugging Using the Quartus II Software
Revised: *August 2010*
Part Number: *QII53029-10.0.1*

Chapter 16 Quick Design Debugging Using SignalProbe
Revised: *July 2010*
Part Number: *QII53008-10.0.0*

Chapter 17 Design Debugging Using the SignalTap II Logic Analyzer
Revised: *July 2010*
Part Number: *QII53009-10.0.0*

Chapter 18 In-System Debugging Using External Logic Analyzers
Revised: *August 2010*
Part Number: *QII53016-10.0.1*

Chapter 19 In-System Modification of Memory and Constants
Revised: *August 2010*
Part Number: *QII53012-10.0.1*

Chapter 20 Design Debugging Using In-System Sources and Probes
Revised: *July 2010*
Part Number: *QII53021-10.0.0*

Chapter 21 Cadence Encounter Conformal Support
Revised: *July 2010*
Part Number: *QII53011-10.0.0*

Chapter 22 Quartus II Programmer
Revised: *July 2010*
Part Number: *QII53022-10.0.0*

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. The Quartus II software provides a wide range of features for performing functional and timing simulation of designs in EDA simulation tools.

This section includes the following chapters:

- Chapter 1, Simulating Designs with EDA Tools

  This chapter provides guidelines to help you perform simulation for your Altera® designs using EDA simulators and the Quartus II NativeLink feature.

- Chapter 2, Mentor Graphics ModelSim/ QuestaSim Support

  This chapter provides detailed instructions about how to simulate your design in the ModelSim-Altera® software or the Mentor Graphics® ModelSim software.

- Chapter 3, Synopsys VCS and VCS MX Support

  This chapter describes how to use the Synopsys VCS and VCS MX software to simulate designs that target Altera FPGAs.

- Chapter 4, Cadence NC-Sim Support

  This chapter describes the basic NC-Sim, NC-Verilog, and NC-VHDL functional, post-synthesis, and gate-level timing simulations.

- Chapter 5, Aldec Active-HDL and Riviera-PRO Support

  This chapter describes how to use the Active-HDL software to simulate designs that target Altera FPGAs.

- Chapter 6, Simulating Altera IP in Third-Party Simulation Tools

  This chapter describes the process for instantiating the IP megafunctions in your design and simulating their functional simulation models in Altera-supported, third-party simulation tools.

1. Simulating Designs with EDA Tools

QII53025-10.0.0

This chapter provides guidelines to help you perform simulation for your Altera® designs using EDA simulators and the Quartus® II NativeLink feature.

Introduction

The Quartus II software assists you in FPGA and ASIC designs, from RTL level to on-chip level. Simulation is the process of verifying the design behavior and timing before configuring the chip.

In previous versions of the Quartus II software, you could use either the Quartus II simulator or an EDA simulator to perform your simulation. The Quartus II software no longer supports a built-in simulator, and you must use EDA simulators to perform simulation.

This chapter guides you through simulation of your Altera design using EDA simulators. There are two ways to simulate your Altera design using EDA simulators:

- Use the NativeLink feature to automatically create scripts for EDA simulators and launch them
- Perform manual simulation using EDA simulators

This chapter focuses on the steps required to perform your simulation using EDA simulators with the NativeLink feature.

Manual simulation using EDA simulators is described in the *Simulation* section in volume 3 of the *Quartus II Handbook*.

This chapter includes the following topics:

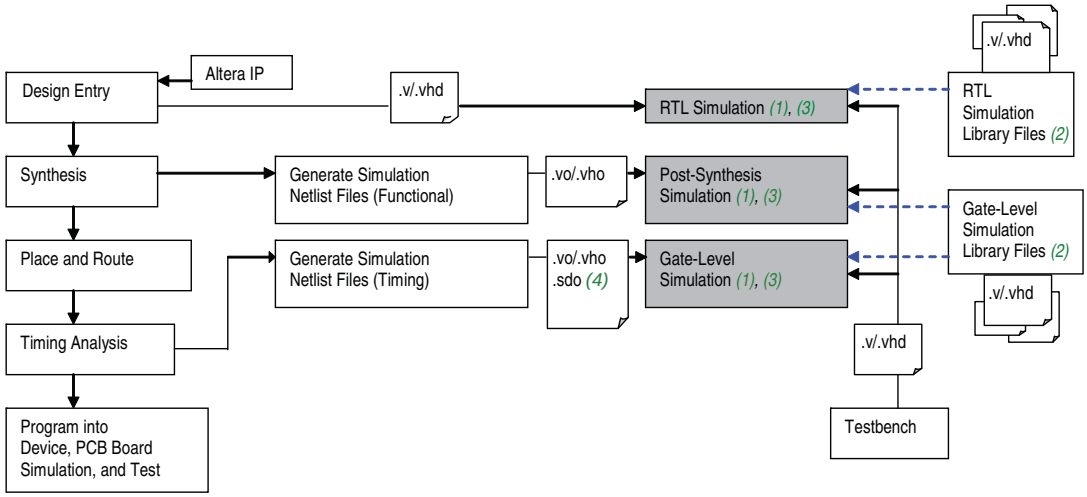
- "PLD Design Flow" on page 1–2
- "Simulation Libraries" on page 1–5
- "EDA Simulation Library Compiler" on page 1–9
- "Using the NativeLink Feature" on page 1–11

PLD Design Flow

This section describes the programmable logic device (PLD) design flow.

Figure 1–1 shows the Altera design flow using EDA simulators and the Quartus II software.

Figure 1–1. Altera Design Flow Using EDA Simulators and the Quartus II Software



Notes to Figure 1–1:

(1) Simulation can be run with the NativeLink feature or performed manually.

(2) ModelSim-Altera does not require simulation library source files for simulation.

(3) The simulation tools include ModelSim, ModelSim-Altera, QuestaSim, VCS, VCS MX, NCSim, Active-HDL, and Riviera-PRO.

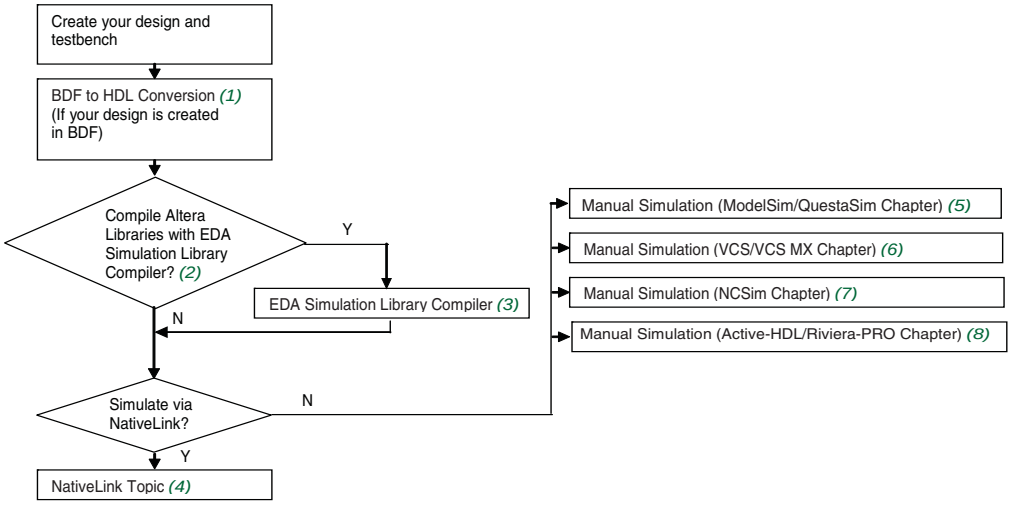
(4) **.vo**, **.vho**, and **.sdo** are generated by the Quartus II Netlist Writer.

RTL Functional Simulation Flow

Figure 1–2 shows the overall RTL functional simulation flow from the Quartus II software. For information about running EDA simulation automatically through the Quartus II software, refer to "Using the NativeLink Feature" on page 1–11.

For information about manual simulation, refer to the EDA simulator chapters in the *Simulation* section in volume 3 of the *Quartus II Handbook*.

Figure 1–2. : EDA RTL Functional Simulation Flow from the Quartus II Software



Notes to Figure 1–2:

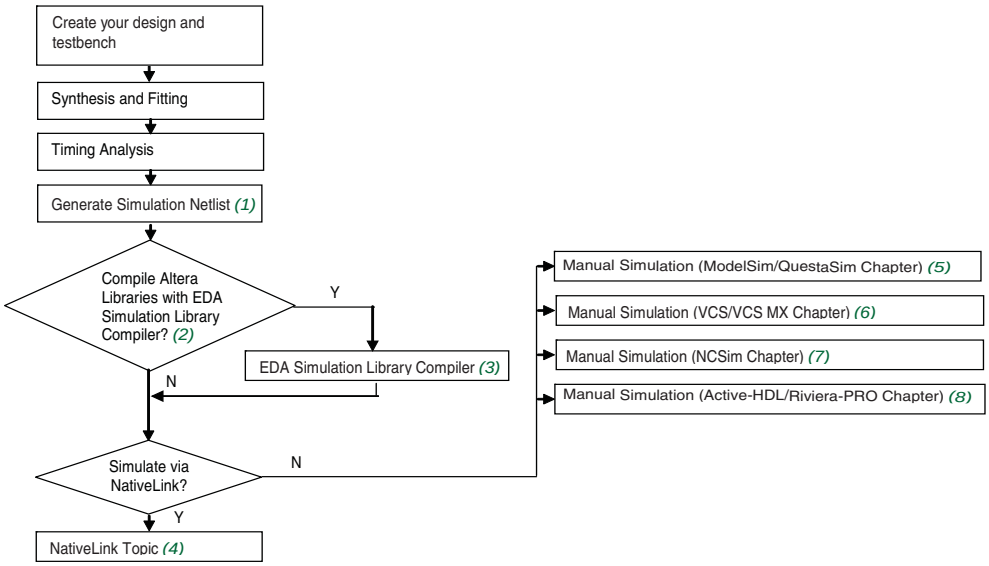
(1) For more information, refer to "Converting BDF Format to HDL Format" on page 1–4.
(2) Do not compile the Altera libraries if you are using ModelSim-Altera or ModelSim-Altera Web Edition.
(3) For more information, refer to *Compiling Simulation Libraries in the Quartus II Software* in Quartus II Help.
(4) For more information, refer to *Using the NativeLink Feature with Other EDA Tools* in Quartus II Help.
(5) For more information, refer to *Performing a Functional Simulation with the ModelSim Software* or *Performing a Functional Simulation with the QuestaSim Software* in Quartus II Help.
(6) For more information, refer to *Performing a Functional Simulation with the VCS Software* or *Performing a Functional Simulation with the VCS-MX Software* in Quartus II Help.
(7) For more information, refer to *Performing a Functional Simulation with the NCSim Software* in Quartus II Help.
(8) For more information, refer to *Performing a Simulation of a Verilog HDL Design with the Active-HDL Software*, *Performing a Simulation of a VHDL Design with the Active-HDL Software*, or *Performing an RTL Functional Simulation with the Riviera-PRO Software* in Quartus II Help.

Gate-Level Timing Simulation Flow

Figure 1–3 shows the overall gate-level timing simulation flow from the Quartus II software. For information about running EDA simulation automatically through the Quartus II software, refer to "Using the NativeLink Feature" on page 1–11.

For information about manual simulation, refer to the EDA simulator chapters in the *Simulation* section in volume 3 of the *Quartus II Handbook*.

Figure 1–3. EDA Gate-Level Timing Simulation Flow from Quartus II Software



Notes to Figure 1–3:

(1) For more information, refer to "Simulation Netlist Files" on page 1–6.
(2) Do not compile the Altera libraries if you are using ModelSim-Altera or ModelSim-Altera Web Edition.
(3) For more information, refer to *Compiling Simulation Libraries in the Quartus II Software* in Quartus II Help.
(4) For more information, refer to *Using the NativeLink Feature with Other EDA Tools* in Quartus II Help.
(5) For more information, refer to *Performing a Timing Simulation with the ModelSim Software* or *Performing a Timing Simulation with the QuestaSim Software* in Quartus II Help.
(6) For more information, refer to *Performing a Timing Simulation with the VCS Software* or *Performing a Timing Simulation with the VCS-MX (VHDL) Software* in Quartus II Help.
(7) For more information, refer to *Performing a Timing Simulation with the NCSim Software* in Quartus II Help.
(8) For more information, refer to *Performing a Simulation of a Verilog HDL Design with the Active-HDL Software*, *Performing a Simulation of a VHDL Design with the Active-HDL Software*, or *Performing a Gate-Level Simulation with the Riviera-PRO Software* in Quartus II Help.

Converting BDF Format to HDL Format

If you create your design in BDF (Block Diagram Format), you must convert it to HDL format (Verilog HDL or VHDL) to perform RTL functional simulation in EDA simulators. To convert your design from BDF format to HDL format, follow these steps:

1. Detach the BDF window.

2. On the File menu, point to **Create/Update** and click **Create HDL Design File for Current File**.
3. From the **File type** list, select **VHDL** or **Verilog HDL**.
4. Click **OK**.

After following these steps, the HDL file is generated. The HDL file and the BDF file have the same name but different extensions (for example, if your BDF file is **example.bdf**, the HDL file created is **example.v** or **example.vhd**).

Simulation Libraries

When you are using an EDA simulator, you must determine which libraries are required for your simulation if you have used Altera megafunctions or IP in your design.

RTL Functional Simulation

The following libraries are required to perform RTL functional simulation:

- If your design does not include transceivers or a PCI Express® (PCIe®) core, you need the AlteraBehaviorLibraries.
- If your design includes transceivers, you need <*family*>**_hssi_atoms**, **sgate**, and the AlteraBehaviorLibraries.
- If your design includes a hard PCIe core, you need <*family*>**_pcie_hip_atoms**, **sgate**, the above libraries, and the AlteraBehaviorLibraries.
- For Stratix GX devices, you need **stratixgx_mf**, **sgate**, and the AlteraBehaviorLibraries.

AlteraBehaviorLibraries refers to **altera_mf**, **220model**, and **altera_primitives**.

If you are targeting a Stratix V device, compile the following files in the **quartus/eda/sim_lib/<vendor>** directory:

- **stratixv_atoms_ncrypt.v**
- **stratixv_hssi_atoms_ncrypt.v**
- **stratixv_pcie_hip_atoms_ncrypt.v**

These files contain IEEE encrypted Verilog models.

Vendor refers to Synopsys, Cadence Design Systems, Inc., Mentor Graphics, or Aldec, Inc.

Compile **stratixv_pcie_hip_atoms_ncrypt.v** with the System Verilog option.

Also, compile the following files in the **quartus/eda/sim_lib** directory:

- **stratixv_atoms.v**
- **stratixv_atoms.vhd**
- **stratix_hssi_atoms.v**

- **stratix_hssi_atoms.vhd**
- **stratixv_pcie_hip_atoms.v**
- **stratixv_pcie_hip_atoms.vhd**
- **stratixv_components.vhd**
- **stratix_hssi_components.vhd**
- **stratixv_pcie_hip_components.vhd**

The PCIe files are required only if you are using the PCIe hard IP.

To simulate Stratix V devices in VHDL, you need a VHDL or Verilog co-simulator.

For a list of all RTL functional simulation library files in the Quartus II directory, refer to *Altera Functional Simulation Libraries* in Quartus II Help.

Gate-Level Timing Simulation

The following libraries are required to perform gate-level timing simulation:

- If your design does not include transceivers or a PCIe core, you need <*family*>**_atoms**, **sgate**, and the AlteraBehaviorLibraries.
- If your design includes transceivers, you need <*family*>**_hssi_atoms**, <*family*>**_atoms**, **sgate**, and the AlteraBehaviorLibraries.
- If your design includes transceivers and a PCIe core, you need <*family*>**_pcie_hip_atoms**, <*family*>**_hssi_atoms**, <*family*>**_atoms**, **sgate**, and the AlteraBehaviorLibraries.

For a list of all gate-level timing and post-fit simulation library files in the Quartus II directory, refer to *Altera Post-Fit Libraries* in Quartus II Help.

Simulation Netlist Files

Simulation netlist files are required to perform post-synthesis simulation or gate-level timing simulation. These simulation netlist files are generated using the EDA Netlist Writer.

If you are performing post-synthesis simulation, the Verilog HDL Output File (**.vo**) or the VHDL Output File (**.vho**) is required. For the steps required to generate post-synthesis simulation netlist files for **.vo** or **.vho** files, refer to "Generating Post-Synthesis Simulation Netlist Files".

If you are performing gate-level timing simulation, the **.vo** or **.vho** file and the Standard Delay Format Output File (**.sdo**) are required. The **.sdo** file is used to annotate the delay for the elements found in the **.vo** or **.vho** file.

To specify options for generating **.vo, .vho**, and **.sdo** files, refer to *Specifying HDL Output Settings* in Quartus II Help.

Generating Post-Synthesis Simulation Netlist Files

To generate post-synthesis simulation netlist files in the Quartus II software, perform the following steps:

1. On the Processing menu, point to **Start** and click **Start Analysis & Synthesis** (you can also perform this after step 2).
2. Configure the EDA Netlist Writer to generate functional simulation netlist. Refer to *Specifying HDL Output Settings* in Quartus II Help.
3. 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 EDA simulators. 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/***<EDA Simulator>* **directory** (*<EDA Simulator>* can be **modelsim**, **questasim**, **vcs**, **vcs mx**, **rivierapro**, **ncsim**, or **activehdl**).

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) was not 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> ↵
```

Generating Gate-Level Timing Simulation Netlist Files

To perform gate-level timing simulation, the EDA simulators require 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, follow these steps:

1. Configure the EDA Netlist Writer to generate functional simulation netlist. Refer to "Generating Post-Synthesis Simulation Netlist Files" on page 1–7.
2. If you have not run a full compilation prior to the fitting process, perform a full compilation. On the Processing menu, click **Start Compilation**.

3. 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**, a **.vho**, and a **.sdo** file used for gate-level timing simulations in the EDA simulators. 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/**<*EDA Simulator*> **directory** (<*EDA Simulator*> can be **modelsim**, **questasim**, **vcs**, **vcs mx**, **rivierapro**, **ncsim**, or **activehdl**).

Generating Timing Simulation Netlist Files with Different Timing Models

In Stratix III and later devices, you can specify different temperature and voltage parameters to generate the timing simulation netlist files. If you enable the Quartus II TimeQuest Timing Analyzer or the Quartus II Classic Timing Analyzer when generating the timing simulation netlist files (**.vo, .vho,** and **.sdo** files), different timing models for different operating conditions are used by default. Multi-corner timing analysis is run by default during the full compilation.

To manually generate the simulation netlist files (**.vo** or **.vho** and **.sdo**) for the three different operating conditions, follow these steps:

1. Generate all available corner models at all operating conditions by typing the following command at a command prompt:

```
quartus_sta <project name> --multicorner ↵
```

2. Generate the timing simulation netlist files for all three corners by performing steps 1 through 3 in "Generating Gate-Level Timing Simulation Netlist Files" on page 1–7. The output files are generated in the simulation output directory.

The following examples show the timing simulation netlist files are generated for the operating conditions of the preceding steps when Verilog HDL 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 TimeQuest Timing Analyzer or the Quartus II Classic Timing Analyzer with a different timing model before you start the EDA Netlist Writer.

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 model:

```
create_timing_netlist --fast_model ↵
```

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 perform classic timing analysis with the fast timing model by typing the following command at a command prompt:

```
quartus_tan <project_name> --fast_model=on ↵
```

After running the Quartus II TimeQuest Timing Analyzer or the Quartus II Classic Timing Analyzer, perform steps 1 through 3 in "Generating Gate-Level Timing Simulation Netlist Files" on page 1–7 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 performing multi-corner timing analysis, refer to the *Quartus II TimeQuest Timing Analyzer* chapter or the *Quartus II Classic Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

EDA Simulation Library Compiler

The EDA Simulation Library Compiler compiles 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 for RTL functional and gate-level timing simulation.

When the compilation targets third-party simulation tools such as ModelSim, QuestaSim, Active-HDL, Riviera-PRO, VCS, VCS MX, and NCSim, the compiled libraries are kept in the directory you specified. When you perform the simulation using these simulators, you can use or reuse the compiled libraries and avoid the overhead associated with redundant library compilations.

If the compilation targets third-party simulation tool VCS, then the VCS options file **simlib_comp.vcs** is 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 the EDA Simulation Library Compiler, ensure that the appropriate simulation tools are already installed and their execution paths are specified. To specify the path, refer to "Setting Up the EDA Simulator Execution Path" on page 1–11.

For more information about compiling Verilog HDL and VHDL simulation libraries for all Altera devices and supported third-party simulators, refer to *Compiling Simulation Libraries in the Quartus II Software* in Quartus II Help.

Running the EDA Simulation Library Compiler Through the GUI

The EDA Simulation Library Compiler does not support ModelSim-Altera because ModelSim-Altera already contains precompiled libraries.

To compile libraries with the EDA Simulation Library Compiler GUI, follow these steps:

1. On the Tools menu, click **Launch EDA Simulation Library Compiler**. The **EDA Simulation Library Compiler** dialog box appears.
2. In the **Tool name** list under **EDA simulation tool**, select your preferred EDA simulator. 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 the **Selected families** box.
4. Under **Library language**, select **VHDL**, **Verilog HDL**, or both.
5. Specify the output netlist destination by editing or browsing to a directory in the **Output directory** box.

   You can then link to the directory so that the NativeLink feature will reuse the compiled libraries rather than compile the Altera Libraries. Refer to step 9 under "Configuring NativeLink Settings" on page 1–12.
6. Click **Start Compilation**.

When the EDA Simulation Library Compiler finishes, for ModelSim, QuestaSim, Active-HDL, Riviera-PRO, VCS MX, and NCSim, all required libraries are compiled and stored in <*output location you specified*>/<*verilog_libs or vhdl_libs*>. The library files (for example, **modelsim.ini**, **cds.lib**, and **synopsys_sim.setup**) are also generated and stored in <*output location you specified*>.

When the EDA Simulation Library Compiler finishes, for VCS, t**he simlib_comp.vcs** file is generated. Example 1–1 shows the contents of the **simlib_comp.vcs** file.

Example 1–1.

```
+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 testbench file in the option file, **.synopys_sim.setup**, as shown in Example 1–2.

Example 1–2.

```
+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 testbench files, type the following command:

```
vcs -file simlib_comp.vcs ↵
```

Running the EDA Simulation Library Compiler from the Command Line

To set the EDA tool executable location, type the following command:

```
export QUARTUS_INIT_PATH=$PATH
```

To run the EDA Simulation Library Compiler from the command line, type the following command:

```
quartus_sh --simlib_comp -family <device> -tool <simulation tool name>
-tool_path <simulator executable location>
-language <language> -directory <directory> ↵
```

For more information about the command line options and how to use them, type the following command:

```
quartus_sh --help=simlib_comp ↵
```

Using the NativeLink Feature

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 an EDA simulator within the Quartus II software.

Setting Up the EDA Simulator Execution Path

To run an EDA simulator automatically from the Quartus II software using the NativeLink feature, 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 the path or browse to the directory containing the executables of your EDA tool.

Table 1–1 lists the execution paths for each EDA simulator.

Table 1–1. Execution Paths for EDA Simulators

Simulator	Path
ModelSim-Altera	**c:***<ModelSim-Altera installation path>***\win32aloem** (Windows) **/***<ModelSim-Altera installation path>***/linuxaloem** (Linux)
ModelSim	**c:***<ModelSim installation path>***\win32** (Windows) **/***<ModelSim installation path>***/bin** (Linux)
QuestaSim	**c**:*<QuestaSim installation path>***\win32** (Windows) /<QuestaSim installation path>**/bin** (Linux)
VCS/VCS MX	**/***<VCS MX installation path>***/bin** (Linux)
NCSim	**/***<NCSim installation path>***/tools/bin** (Linux)
Active-HDL	**c:***<Active-HDL installation path>***\bin** (Windows)
Riviera-PRO	**c:***<Riviera-PRO installation path>***\bin** (Windows)

5. Click **OK**.

You can also specify the path to the simulator's executables by typing the **set_user_option** Tcl command, as follows:

```
set_user_option -name EDA_TOOL_PATH_MODELSIM <path to executables> ↵
set_user_option -name EDA_TOOL_PATH_MODELSIM_ALTERA <path to \
executables> ↵
set_user_option -name EDA_TOOL_PATH_QUESTASIM <path to executables> ↵
set_user_option -name EDA_TOOL_PATH_VCS <path to executables> ↵
set_user_option -name EDA_TOOL_PATH_VCS_MX <path to executables> ↵
set_user_option -name EDA_TOOL_PATH_NCSIM <path to executables> ↵
set_user_option -name EDA_TOOL_PATH_ACTIVEHDL <path to executables> ↵
set_user_option -name EDA_TOOL_PATH_RIVIERAPRO <path to executables> ↵
```

Configuring NativeLink Settings

To configure NativeLink settings, follow these 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. In the **Tool name** list, select your preferred EDA simulator.
4. For gate-level simulation, if you want to run simulation in your EDA simulator automatically after Quartus II full compilation, turn on **Run gate-level simulation automatically after compilation**.
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 Testbench Files Using the NativeLink Feature" on page 1–14.

1. If you want to run the EDA simulator in command-line mode, follow these 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**.
2. If you want to generate only the simulation script without launching the EDA simulator during NativeLink simulation, follow these 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**.
   d. Click **OK**.

   If you turn this option on and run NativeLink, only the simulation command script is generated. The file names of simulation command scripts for various simulators are as follows:

   <*project_name*>**_run_msim_**<*rtl/gate*>**_level_**<*verilog/vhdl*>**.do** (ModelSim)

   <*project_name*>**_run_questasim_**<*rtl/gate*>**_level_**<*verilog/vhdl*>**.do** (QuestaSim)

   <*project_name*>**_sim_**<*rtl/gate*>_<*verilog/vhdl*>**.do** (Riviera-PRO and Active-HDL)

   script_file.sh and <*project_name*>**_rtl.vcs** (VCS)

   <*project_name*>**_vcsmx_**<*rtl/gate*>_<*vhdl/verilog*>**.tcl** (VCS MX)

   <*project_name*>**_ncsim_**<*rtl/gate*>_<*verilog/vhdl*>**.tcl** (NCSim)

8. Depending on the simulator, perform the simulation by typing one of the following commands:

   `do <script>.do` ↵ (ModelSim Macro File)

   `quartus_sh -t <script>.tcl` ↵ (Tcl Script File)

   `sh <script>.sh` ↵ (Shell script)

9. If you have compiled libraries using the EDA Simulation Library Compiler, follow these 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. The path should be the same as the path you have set in the Output Directory in the EDA Simulation Library Compiler.

Step 9 is not applicable for Active-HDL and Riviera-PRO.

For more information about the EDA Simulation Library Compiler, refer to "EDA Simulation Library Compiler" on page 1–9.

For more information about using the Quartus II software with other EDA tools, refer to *About Using the Quartus II Software with Other EDA Tools* in Quartus II Help.

Running RTL Functional Simulation Using the NativeLink Feature

To run RTL functional simulation using the NativeLink feature, follow these steps:

1. Configure the NativeLink settings. Refer to "Configuring NativeLink Settings" on page 1–12.
2. 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.
3. On the Tools menu, point to **Run EDA Simulation Tool** and click **EDA RTL Simulation** to automatically run the EDA simulator, compile all necessary design files, and complete a simulation.

Running Gate-Level Timing Simulation Using the NativeLink Feature

To run a gate-level timing simulation using the NativeLink feature, follow these steps:

1. Configure the EDA Netlist Writer settings. Refer to "Generating Post-Synthesis Simulation Netlist Files" on page 1–7.
2. Configure the NativeLink settings. Refer to "Configuring NativeLink Settings" on page 1–12.
3. On the Processing menu, click **Start Compilation** to perform Quartus II full compilation, including generation of an EDA netlist file.
4. On the Tools menu, point to **Run EDA Simulation Tool** and click **EDA Gate Level Simulation** to automatically run the EDA simulator, compile all necessary design files, and complete a simulation.

If you have turned on **Run gate-level simulation automatically after compilation** while configuring NativeLink settings, you can skip step 4.

Setting Up Testbench Files Using the NativeLink Feature

You can use the NativeLink feature to compile your design files and testbench files, and run an EDA simulation tool to automatically perform a simulation.

To set up the NativeLink feature for simulation, follow these steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, under **EDA Tool Settings**, click **Simulation**. The **Simulation** page appears.
3. In the **Tool name** list, select your preferred EDA simulator.
4. Under **NativeLink settings**, select **None**, **Compile test bench**, or **Script to compile test bench** (Table 1–2).

Table 1–2. NativeLink Testbench Settings

Setting	Description
None	NativeLink compiles 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 are compiled. Use this option when you want to create your own script to compile your testbench file and perform simulation.

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 following these steps:

1. Click **Test Benches**. The **Test Benches** dialog box appears.
2. Click **New**. The **New Test Bench Settings** dialog box appears.
3. In the **Top level module in test bench** box, type the top-level testbench entity or module name. For example, for a Quartus II-generated VHDL testbench, type <*Vector Waveform File name*>**_vhd_vec_tst**. The testbench name in the **Test bench name** box automatically follows the top-level testbench entity or module name.
4. For gate-level simulation, turn on **Use test bench to perform VHDL timing simulation**. In the **Design instance name in test bench** box, type the full instance path to the top level of your FPGA design. For example, for a Quartus II-generated VHDL testbench, type **i1**.
5. Under **Simulation period**, select **Run simulation until all vector stimuli are used** or specify the end time of the simulation.
6. Under **Test bench files**, browse and add all of your testbench files in the **File name** box. Use the Up and Down buttons to reorder your files. The script used by the NativeLink feature compiles the files in order from top to bottom.

You can also specify the library name and HDL version to compile the testbench file. The NativeLink feature compiles the testbench file to a library name using the specified HDL version.

7. Click **OK**.
8. In the **Test benches** dialog box, click **OK**.
9. 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.

If you select **Script to compile test bench**, browse to your script and click **OK.**

You can also use the NativeLink feature in your script to compile your design files and testbench files with customized settings.

Conclusion

The Quartus II NativeLink feature eases the tasks of setting up and running a simulation, enables you to launch third-party simulators to perform simulation from within the Quartus II software, and automates the compilation and simulation of testbenches.

Document Revision History

Table 1–3 shows the revision history for this chapter.

Table 1–3. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Linked to Quartus II Help where appropriate ■ Removed Referenced Documents section ■ Removed Creating Testbench Files ■ Added VCS and QuestaSim as third-party simulation tools ■ Updated "Running the EDA Simulation Library Compiler Through the GUI" on page 1–10 ■ Updated "Setting Up the EDA Simulator Execution Path" on page 1–11 ■ Updated "Configuring NativeLink Settings" on page 1–12 ■ Updated "Setting Up Testbench Files Using the NativeLink Feature" on page 1–14
November 2009	9.1.0	Initial release

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

Take an online survey to provide feedback about this handbook chapter.

2. Mentor Graphics ModelSim/ QuestaSim Support

QII53001-10.0.0

This chapter provides detailed instructions about how to simulate your design in the ModelSim-Altera® software, Mentor Graphics® ModelSim software, and Mentor Graphics QuestaSim software.

An Altera Quartus® II software subscription includes the ModelSim-Altera Starter Edition, which is a a no-cost entry-level version of the ModelSim-Altera Subscription Edition software. The ModelSim-Altera Subscription Edition software offers support for all Altera devices. Both versions are available on PC and Linux platforms. You can use the ModelSim-Altera software to perform functional, post-synthesis, and gate-level timing simulations for either Verilog HDL or VHDL designs that target an Altera FPGA.

In this chapter, ModelSim refers to ModelSim SE, PE, DE, and QuestaSim. ModelSim-Altera refers to ModelSim-Altera Starter Edition and ModelSim-Altera Subscription Edition software.

This chapter includes the following topics:

- "Software Requirements"
- "Design Flow with ModelSim-Altera or ModelSim/QuestaSim Software"
- "Simulation Libraries" on page 2–2
- "Performing Simulation Using the ModelSim-Altera Software" on page 2–3
- "Performing Simulation Using the ModelSim/QuestaSim Software" on page 2–5
- "Simulating Designs that Include Transceivers" on page 2–16
- "Using the NativeLink Feature with ModelSim-Altera or ModelSim/QuestaSim Software" on page 2–24
- "Generating a Timing Value Change Dump (.vcd) File for the PowerPlay Power Analyzer" on page 2–25
- "Viewing a Waveform from a .wlf File" on page 2–26
- "Scripting Support" on page 2–26
- "Software Licensing and Licensing Setup in ModelSim-Altera Subscription Edition" on page 2–28

Software Requirements

To simulate your design using the ModelSim-Altera and ModelSim/QuestaSim software, you must first set up the Altera libraries. These libraries are installed with the Quartus II software.

For more information about installing the software and directories created during the Quartus II software installation, refer to the *Altera Software Installation and Licensing* manual.

Design Flow with ModelSim-Altera or ModelSim/QuestaSim Software

You can perform the following types of simulations using the ModelSim-Altera and ModelSim/QuestaSim software:

- Functional simulation
- Post-synthesis simulation
- Gate-level timing simulation

Some versions of ModelSim and QuestaSim support SystemVerilog, PSL assertions, SystemC, and more. Refer to Mentor Graphics literature or your salesperson to learn more about the features supported in the different versions of ModelSim and QuestaSim.

The VHDL version of ModelSim-Altera and other single-language VHDL versions of ModelSim cannot simulate designs that target the Stratix V device family if you are using transceivers.

You need a version of ModelSim that supports VHDL/Verilog co-simulation to simulate designs that use Stratix V transceivers.

For more information about the Quartus II software design flow, refer to the "PLD Design Flow" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

For additional documentation about ModelSim-Altera, refer to the ModelSim-Altera Help that ships with the product. Click the **Help** button on the ModelSim-Altera toolbar.

Simulation Libraries

Simulation model libraries are required to run a simulation whether you are running a functional simulation, post-synthesis simulation, or gate-level timing simulation. In general, running a functional simulation requires the functional simulation model libraries, while running a post-synthesis or gate-level timing simulation requires the gate-level timing simulation model libraries. Unless you are using ModelSim-Altera, you must compile the necessary library files before you can run the simulation. The ModelSim-Altera software has the Altera libraries pre-compiled and built in. Do not compile these libraries again.

There are a few exceptions where you must compile gate-level timing simulation library files to perform functional simulation. For example, some Altera megafunctions require gate-level libraries to perform a functional simulation using third-party simulators.

Precompiled Simulation Libraries in the ModelSim-Altera Software

Precompiled libraries for both functional and gate-level simulations are provided for the ModelSim-Altera software. You should not compile these library files before running a simulation.

The precompiled 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 precompiled 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 precompiled libraries.

For a list of precompiled library names for all functional and gate-level simulation models, refer to *ModelSim Precompiled Libraries* in Quartus II Help.

Simulation Library Files in the Quartus II Software

In ModelSim/QuestaSim, no precompiled libraries are available. You must compile the necessary libraries to perform functional or gate-level simulation.

For a list of all functional simulation library files in the Quartus II directory, refer to *Altera Functional Simulation Libraries* in Quartus II Help. For a list of all post-synthesis and post-fit (gate-level) library files in the Quartus II directory, refer to *Altera Post-Fit Libraries* in Quartus II Help.

Disabling Timing Violation on Registers

In certain situations, a timing violation can be ignored and you can 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 Quartus II Tcl command disables 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>
```

Performing Simulation Using the ModelSim-Altera Software

You can perform simulation of Verilog HDL or VHDL designs with the ModelSim-Altera software at three levels: functional, post-synthesis, and gate-level.

For high-speed simulation, you must select **ps** in the **Resolution** list for your simulator resolutions (**Design** tab of the **Start Simulation** dialog box). If you choose slower than **ps**, the high-speed simulation may fail.

Performing Functional Simulation

Functional simulation verifies code syntax and design functionality. The following sections describe how to perform functional simulation in the ModelSim-Altera software for a Verilog HDL or VHDL design.

The ModelSim-Altera software includes precompiled simulation libraries for Altera-provided models. You should not create simulation libraries and compile simulation models for the pre-compiled Altera libraries.

Setting Up a Quartus II Project for the ModelSim-Altera Software

The first steps in performing a simulation are starting the ModelSim-Altera software, changing to your project/simulation directory, and creating libraries for your design.

For more information, refer to *Setting Up a Project with the ModelSim-Altera Software* in Quartus II Help.

Compiling and Loading Designs with the ModelSim-Altera Software

For information about compiling and loading your design files and testbench files, refer to *Mapping to Libraries and Compiling Design Files with the ModelSim-Altera Software* in Quartus II Help.

Performing the Simulation

For information about performing a functional simulation with the ModelSim-Altera software, refer to *Performing a Functional Simulation with the ModelSim-Altera Software* in Quartus II Help.

Performing Post-Synthesis Simulation

You can perform post-synthesis simulation to verify that design functionality is preserved 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. For more information, refer to the "Generating Post-Synthesis Simulation Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

The ModelSim-Altera software includes precompiled simulation libraries. You should not create simulation libraries and compile simulation models.

For information about performing a post-synthesis simulation with the ModelSim-Altera software, refer to *Performing a Timing Simulation with the ModelSim-Altera Software* in Quartus II Help.

Performing Gate-Level Timing Simulation

Gate-level timing simulation is an important step in ensuring that the device functionality is correct and meets all timing requirements following place and route. You can create the gate-level netlist in the Quartus II software and use the netlist to perform gate-level timing simulation with the ModelSim-Altera software.

Before running gate-level timing simulation, generate gate-level timing simulation netlist files. For more information, refer to the "Generating Gate-Level Timing Simulation Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

The ModelSim-Altera software includes precompiled simulation libraries. It is not necessary to create simulation libraries and compile simulation models.

For information about performing a gate-level simulation with the ModelSim-Altera software, refer to *Performing a Timing Simulation with the ModelSim-Altera Software* in Quartus II Help.

For additional documentation about ModelSim-Altera, refer to the ModelSim-Altera Help that ships with the product. Click the **Help** button on the ModelSim-Altera toolbar.

Performing Simulation Using the ModelSim/QuestaSim Software

You can perform simulation of Verilog HDL or VHDL designs with the ModelSim/QuestaSim software at three levels: functional, post-synthesis, and gate-level.

You can perform the simulation through the GUI or from the command line. The following sections provide instructions to perform the simulation through the GUI and from 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 (**Design** tab of the **Start Simulation** dialog box). If you choose slower than **ps**, the high-speed simulation may fail.

Simulating VHDL Designs Using the GUI

This section provides information about performing functional, post-synthesis, and gate-level simulations of VHDL designs using the GUI.

Performing Functional Simulation

This section provides information about compiling simulation models and performing a functional simulation.

Compiling Simulation Models into Simulation Libraries

If you are not using the EDA Simulation Library Compiler, as described in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*, perform the following steps to compile simulation models into simulation libraries:

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**.

If you are targeting a Stratix V device, compile the following files in the **mentor** subdirectory:

stratixv_atoms_ncrypt.v
stratixv_hssi_atoms_ncrypt.v
stratixv_pcie_hip_atoms_ncrypt.v

These files contain IEEE encrypted Verilog models suitable for VHDL/Verilog co-simulation. You need a co-simulation license from Mentor Graphics to use these models.

☞ Compile these encrypted Verilog files before you compile any VHDL files.

☞ Compile **stratixv_pcie_hip_atoms_ncrypt.v** with the SystemVerilog option.

Also, compile the following files in the **quartus/eda/sim_lib** directory:

stratixv_atoms.vhd
stratixv_components.vhd
stratixv_hssi_components.vhd
stratixv_pcie_hip_components.vhd
stratixv_hssi_atoms.vhd
stratixv_pcie_hip_atoms.vhd

☞ The PCIe® files are required only if you are using the PCIe HIP.

☞ 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**.

Performing the Simulation

For information about simulating VHDL designs using the GUI, refer to *Performing a Functional Simulation with the ModelSim Software* and *Performing a Functional Simulation with the QuestaSim Software* in Quartus II Help.

To see all of the functional simulation library files, refer to *Altera Functional Simulation Libraries* in Quartus II Help.

Performing Post-Synthesis Simulation

You can perform post-synthesis simulation to verify that design functionality is preserved 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/QuestaSim software.

Before running post-synthesis simulation, generate post-synthesis simulation netlist files. For more information, refer to the "Generating Post-Synthesis Simulation Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

You cannot perform post-synthesis or post-fit (gate-level) simulation if you are targeting the Stratix V device family.

For information about performing a post-synthesis simulation using the GUI, refer to *Performing a Timing Simulation with the ModelSim Software* and *Performing a Timing Simulation with the QuestaSim Software* in Quartus II Help.

Performing 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/QuestaSim software.

You cannot perform post-synthesis or post-fit (gate-level) simulation if you are targeting the Stratix V device family.

Before running gate-level simulation, generate gate-level timing simulation netlist files. For more information, refer to the "Generating Gate-Level Timing Simulation Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

For information about performing a gate-level simulation using the GUI, refer to *Performing a Timing Simulation with the ModelSim Software* and *Performing a Timing Simulation with the QuestaSim Software* in Quartus II Help.

Simulating Verilog HDL Designs Using the GUI

This section provides information about performing functional, post-synthesis, and gate-level simulations of Verilog HDL designs using the GUI.

Performing Functional Simulation

This section provides information about compiling simulation models and performing a functional simulation.

Compiling Simulation Models into Simulation Libraries

If you are not using the EDA Simulation Library Compiler, as described in the section "EDA Simulation Library Compiler" in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*, perform the following steps to compile simulation models into simulation libraries:

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**.

If you are targeting a Stratix V device, compile the following files in the **mentor** subdirectory:

stratixv_atoms_ncrypt.v
stratixv_hssi_atoms_ncrypt.v
stratixv_pcie_hip_atoms_ncrypt.v

These files contain IEEE encrypted Verilog models.

☞ Compile **stratixv_pcie_hip_atoms_ncrypt.v** with the SystemVerilog option.

Also, compile the following files in the **quartus/eda/sim_lib** directory:

stratixv_atoms.v
stratixv_hssi_atoms.v
stratixv_pcie_hip_atoms.v

☞ The PCIe file is required only if you are using the PCIe HIP.

☞ 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**.

Performing the Simulation

For information about performing a functional simulation using the GUI, refer to *Performing a Functional Simulation with the ModelSim Software* and *Performing a Functional Simulation with the QuestaSim Software* in Quartus II Help.

To see all of the functional simulation library files, refer to *Altera Functional Simulation Libraries* in Quartus II Help.

Performing Post-Synthesis Simulation

Perform post-synthesis simulation to verify that design functionality is preserved after synthesis. Create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the ModelSim/QuestaSim software.

Before running post-synthesis simulation, generate post-synthesis simulation netlist files. For more information, refer to the "Generating Post-Synthesis Simulation Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

☞ You cannot perform post-synthesis or post-fit (gate-level) simulation if you are targeting the Stratix V device family.

For information about performing a post-synthesis simulation using the GUI, refer to *Performing a Timing Simulation with the ModelSim Software* and *Performing a Timing Simulation with the QuestaSim Software* in Quartus II Help.

Performing 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/QuestaSim software.

You cannot perform post-synthesis or post-fit (gate-level) simulation if you are targeting the Stratix V device family.

Before running gate-level simulation, generate gate-level timing simulation netlist files. For more information, refer to the "Generating Gate-Level Timing Simulation Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

For information about performing a gate-level simulation using the GUI, refer to *Performing a Timing Simulation with the ModelSim Software* and *Performing a Timing Simulation with the QuestaSim Software* in Quartus II Help.

Simulating VHDL Designs From the Command Line

This section provides information about performing functional, post-synthesis, and gate-level simulations of VHDL designs from the command line.

Simulating VHDL designs from the ModelSim/QuestaSim command line gives you more flexibility and control in compiling the libraries and loading and simulating the VHDL design files. All simulation commands are Tcl commands that can be included in the ***.do** file. Using the ***.do** file allows you to run simulation in batch mode. You have to execute only the ***.do** file, and the ModelSim/QuestaSim tool automatically executes all commands in the ***.do** script macro file.

Performing Functional Simulation

Function simulation verifies code syntax and design functionality.

Type the following commands to perform a functional simulation for VHDL designs with one of the libraries (lib1) listed in the *Altera Functional Simulation Libraries* in Quartus II Help.

To create and compile Altera libraries, type the following commands:

```
vlib <lib1> ↵
vmap <lib1> <lib1> ↵
vcom -work <lib1> <lib1>.vhd ↵
vlib <lib2> ↵
vmap <lib2> <lib2> ↵
vcom -work <lib2> <lib2>.vhd ↵
```

To create the work library and compile the design and testbench files, type the following commands:

```
vlib work ↵
vmap work work ↵
vcom -work work <design_file1>.vhd <design file2>.vhd <testbench \
file>.vhd ↵
```

To load the design, type the following command:

```
vsim -L work -L <lib1> -L <lib2> work.<testbench module name> ↵
```

To add signals to the waveform viewer and run the simulation, type the following commands:

```
add wave * ↵
run ↵
```

Example

```
# Create and compile Altera libraries

vlib altera_mf
vmap altera_mf altera_mf
vcom -work altera_mf altera_mf_components.vhd altera_mf.vhd
vlib lpm
vmap lpm lpm
vcom -work lpm 220pack.vhd 220model.vhd

# Create work library and compile design files and testbench file

vlib work
vmap work work
vcom -work work top_level.vhd adder.vhd testbench.vhd

# Load design

vsim -L work -L altera_mf -L lpm work.testbench

# add signals to the waveform viewer and run simulation

add wave *
run
```

If you are targeting a Stratix V device, compile the following files in the **quartus/eda/sim_lib/mentor** directory:

stratixv_atoms_ncrypt.v
stratixv_hssi_atoms_ncrypt.v
stratixv_pcie_hip_atoms_ncrypt.v

These files contain IEEE encrypted Verilog models suitable for VHDL/Verilog co-simulation. You need a co-simulation license from Mentor Graphics to use these models.

Compile these encrypted Verilog files before you compile any VHDL files.

Compile **stratixv_pcie_hip_atoms_ncrypt.v** with the SystemVerilog option.

Also, compile the following files in the **quartus/eda/sim_lib** directory:

stratixv_atoms.vhd
stratixv_components.vhd
stratixv_hssi_components.vhd
stratixv_pcie_hip_components.vhd
stratixv_hssi_atoms.vhd
stratixv_pcie_hip_atoms.vhd

The PCIe® files are required only if you are using the PCIe HIP.

Performing Post-Synthesis Simulation

Perform post-synthesis simulation to verify that design functionality is preserved after synthesis. Create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the ModelSim/QuestaSim software. Before running post-synthesis simulation, generate post-synthesis simulation netlist files.

For more information, refer to the "Generating Post-Synthesis Simulation Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the Quartus II Handbook.

You cannot perform post-synthesis or post-fit (gate-level) simulation if you are targeting the Stratix V device family.

Type the following commands to perform a post-synthesis simulation for VHDL designs with one of the libraries (lib1) listed in *Altera Post-Fit Libraries* in Quartus II Help.

To create and compile Altera libraries, type the following commands:

```
vlib <lib1> ↵
vmap <lib1> <lib1> ↵
vcom -work <lib1> <lib1>.vhd ↵
vlib <lib2> ↵
vmap <lib2> <lib2>↵
vcom -work <lib2> <lib2>.vhd ↵
```

To create the work library and compile design and testbench files, type the following commands:

```
vlib work ↵
vmap work work ↵
vcom -work work <output_netlist>.vho <testbench file>.vhd ↵
```

To load the design, type the following command:

```
vsim +transport_int_delays +transport_path_delays  -L work -L \ <lib1> -L
<lib2> work.<testbench module name> ↵
```

To add signals to the waveform viewer and run simulation, type the following commands:

```
add wave * ↵
run ↵
```

Example

```
# Create and compile Altera libraries

vlib altera
vmap altera altera
vcom -work altera altera_primitives_components.vhd \
altera_primitives.vhd
vlib stratixiii
vmap stratixiii stratixiii
vcom -work stratixiii stratixiii.atoms.vhd stratixiii_components.vhd

# Create work library and compile design files and testbench file

vlib work
vmap work work
vcom -work work top_level.vho testbench.vhd

# Load design

vsim +transport_int_delays +transport_path_delays  -L work -L \ altera
-L stratixiii work.testbench

# add signals to the waveform viewer and run simulation

add wave *
run
```

Performing Gate-Level Simulation

The steps for gate-level timing simulation are almost same as the steps for post-synthesis simulation.

The only difference is that the **.sdo** file must be back-annotated for gate level-timing simulation.

For VHDL designs, add the **-sdftyp** option for back-annotating.

Example

```
vsim +transport_int_delays +transport_path_delays -sdftyp \ <instance
path to design>= <path to SDO file> -L work -L stratixiii -L \ altera
work.testbench
```

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`).

You cannot perform post-synthesis or post-fit (gate-level) simulation if you are targeting the Stratix V device family.

Simulating Verilog HDL Designs from the Command Line

This section provides information about performing functional, post-synthesis, and gate-level simulations of Verilog HDL designs from the command line.

Simulating Verilog HDL designs from the ModelSim/QuestaSim command line gives you more flexibility and control in compiling the libraries and loading and simulating the Verilog HDL design files. All simulation commands are Tcl commands that can be included in the ***.do** file. Using the ***.do** file allows you to run simulation in batch mode. You have to execute only the ***.do** file, and the ModelSim/QuestaSim tool automatically executes all commands in the ***.do** script macro file.

Performing Functional Simulation

Functional simulation verifies code syntax and design functionality.

Type the following commands to perform a functional simulation for Verilog HDL designs with one of the libraries (lib1) listed in *Altera Functional Simulation Libraries* in Quartus II Help.

To create and compile Altera libraries, type the following commands:

```
vlib <lib1> ↵
vmap <lib1> <lib1> ↵
vlog -work <lib1> <lib1>.v ↵
vlib <lib2> ↵
vmap <lib2> <lib2> ↵
vlog -work <lib2> <lib2>.v ↵
```

To create the work library and compile design and testbench files, type the following commands:

```
vlib work ↵
vmap work work ↵
vlog -work work <design_file1>.v <design file2>.v <testbench file>.v ↵
```

To load the design, type the following command:

```
vsim -L work -L <lib1> -L <lib2> work.<testbench module name> ↵
```

To add signals to the waveform viewer and run simulation, type the following commands:

```
add wave * ↵
run ↵
```

Example

```
# Create and compile Altera libraries

vlib altera_mf_ver
vmap altera_mf_ver altera_mf_ver
vlog -work altera_mf_ver altera_mf.v
vlib lpm_ver
vmap lpm_ver lpm_ver
vlog -work lpm_ver 220model.v

# Create work library and compile design files and testbench file

vlib work
vmap work work
vlog -work work top_level.v adder.v testbench.v

# Load design
```

```
vsim -L work -L altera_mf_ver -L lpm_ver work.testbench
# add signals to the waveform viewer and run simulation
add wave *
run
```

If you are targeting a Stratix V device, compile the following files in the **quartus/eda/sim_lib/mentor** directory:

stratixv_atoms_ncrypt.v
stratixv_hssi_atoms_ncrypt.v
stratixv_pcie_hip_atoms_ncrypt.v

These files contain IEEE encrypted Verilog models.

Compile **stratixv_pcie_hip_atoms_ncrypt.v** with the SystemVerilog option.

Also, compile the following files in the **quartus/eda/sim_lib** directory:

stratixv_atoms.v
stratixv_hssi_atoms.v
stratixv_pcie_hip_atoms.v

The PCIe file is required only if you are using the PCIe HIP.

Performing Post-Synthesis Simulation

Perform post-synthesis simulation to verify that design functionality is preserved after synthesis. Create the post-synthesis netlist in the Quartus II software and use the netlist to perform post-synthesis simulation with the ModelSim/QuestaSim software. Before running post-synthesis simulation, generate post-synthesis simulation netlist files.

For more information, refer to the "Generating Post-Synthesis Simulation Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the Quartus II Handbook.

You cannot perform post-synthesis or post-fit (gate-level) simulation if you are targeting the Stratix V device family.

Type the following commands to perform a post-synthesis simulation for Verilog HDL designs with one of the libraries (lib1) listed in *Altera Post-Fit Libraries* in Quartus II Help.

To create and compile Altera libraries, type the following commands:

```
vlib <lib1> ↵
vmap <lib1> <lib1> ↵
vlog -work <lib1> <lib1>.v ↵
vlib <lib2> ↵
vmap <lib2> <lib2> ↵
vlog -work <lib2> <lib2>.v ↵
```

To create the work library and compile design and testbench files, type the following commands:

```
vlib work ↵
vmap work work ↵
vlog -work work <output_netlist>.vo <testbench file>.v ↵
```

To load the design, type the following command:

```
vsim -t ps +transport_int_delays +transport_path_delays  -L work -L \
<lib1> -L <lib2> work.<testbench module name> ↵
```

To add signals to the waveform viewer and run the simulation, type the following commands:

```
add wave * ↵
run ↵
```

Example

```
# Create and compile Altera libraries

vlib altera_ver
vmap altera_ver altera_ver
vlog -work altera_ver altera_primitives.v
vlib stratixiii_ver

vmap stratixiii_ver stratixiii_ver
vlog -work stratixiii_ver stratixiii_atoms.v

# Create work library and compile design files and testbench file

vlib work
vmap work work
vlog -work work top_level.vo testbench.v

# Load design

vsim +transport_int_delays +transport_path_delays  -L work -L
altera_ver -L stratixiii_ver work.testbench

#add signals to the waveform viwer and run simulation

add wave *
run
```

Performing Gate-Level Simulation

The steps for gate-level timing simulation are almost same as the steps for post-synthesis simulation.

The only difference is that the **.sdo** file must be back-annotated for gate level-timing simulation.

For Verilog HDL designs, the back-annotating process is done within the **output_netlist.vo** script. Therefore, you are not required to back-annotate the SDO file again.

You cannot perform post-synthesis or post-fit (gate-level) simulation if you are targeting the Stratix V device family.

Passing Parameter Information from Verilog to VHDL

You must use in-line parameters to pass values from Verilog HDL to VHDL. Using the **defparam** construct will cause an error in simulation. In the example below:

```
lpm_add_sub_component (
.dataa (dataa),
.datab (datab),
.result (sub_wire0)
);
defparam
```

```
lpm_add_sub_component.lpm_direction = "ADD",
lpm_add_sub_component.lpm_hint =
"ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO",
lpm_add_sub_component.lpm_type = "LPM_ADD_SUB",
lpm_add_sub_component.lpm_width = 12;
```

You will see the following error message:

```
# ** Error: (vsim-3043)
/apps2/home/users/bhlee/SPR_ADOQS/ADOQS10000935_IN_LINE_PARAMETER/lpm_
add_sub1.v(67): Unresolved reference to 'lpm_add_sub_component' in
lpm_add_sub_component.lpm_direction.

# Region: /IN_LINE_PARAMETER_vlg_vec_tst/i1/b2v_inst
```

This megafunction instantiation has been modified to use in-line parameters:

```
lpm_add_sub#(12,"SIGNED","ADD",0,"LPM_ADD_SUB","ONE_INPUT_IS_CONSTANT=
NO,CIN_USED=NO")
lpm_add_sub_component (
    .dataa (dataa),
    .datab (datab),
    .result (sub_wire0)
);
```

The sequence of the parameters depends on the sequence of the GENERIC in the VHDL component declaration.

Speeding Up Simulation

By default, the ModelSim/QuestaSim software runs in a debug-optimized mode. To run the ModelSim/QuestaSim software in speed-optimized mode, add the following two **vlog** command line switches:

```
vlog -fast -05
```

In this mode, module boundaries are flattened and loops are optimized. This eliminates levels of debugging hierarchy, which may result in faster simulation. This switch is not supported in the ModelSim-Altera simulator.

Simulating Designs that Include Transceivers

If your design includes an Arria GX, Arria II GX, Cyclone IV, HardCopy IV, Stratix GX, Stratix II GX, Stratix IV, or Stratix V transceiver, you must compile additional library files to perform functional or gate-level timing simulations.

You cannot perform post-synthesis or post-fit (gate-level) simulation if you are targeting the Stratix V device family.

For Stratix V, you must compile the libraries listed in *Compiling Stratix V Libraries* in Quartus II Help.

Performing simulation with transceivers in Arria II, Cyclone IV, HardCopy IV, and Stratix IV device families are very similar. You have to replace only **stratixiigx_atoms** and **stratixiigx_hssi_atoms** model files with **arriaii_atoms** and **arriaii_hssi_atoms** model files for Arria II devices, **cycloneiv_atoms** and **cycloneiv_hssi_atoms** model files for Cyclone IV devices, and **stratixiv_atoms** and **stratixiv_hssi_atoms** model files for Stratix IV devices.

For high-speed simulation, you must select **ps** in the **Resolution** list for your simulator resolutions (**Design** tab of the **Start Simulation** dialog box). If you choose slower than **ps**, the high-speed simulation may fail.

If your design contains PCI Express® hard IP, refer to the "Simulate the Design" section in the *PCI Express Compiler User Guide.*

The VHDL version of ModelSim-Altera and other single language VHDL versions of ModelSim cannot simulate designs that target the Stratix V device family.

You need a version of ModelSim that supports VHDL/Verilog co-simulation to simulate designs that use Stratix V transceivers.

Functional Simulation for Stratix GX Devices

To perform a 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. If you are using ModelSim/QuestaSim, you must create these libraries to perform a simulation.

Performing Functional Simulation in VHDL (ModelSim-Altera)

To perform functional simulation for Stratix GX devices in VHDL, type the following commands:

```
vcom -work <my_design>.vhd <my testbench>.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L altgxb work.<my_testbench> ↵
```

Performing Functional Simulation in VHDL (ModelSim/QuestaSim)

To perform functional simulation for Stratix GX devices in VHDL, type the following commands:

```
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 in Verilog HDL (ModelSim-Altera)

To perform functional simulation for Stratix GX devices in Verilog HDL, type the following commands:

```
vlog -work <my design>.v <my testbench>.v ↵
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L altgxb work.<my testbench> ↵
```

Performing Functional Simulation in Verilog HDL (ModelSim/QuestaSim)

To perform functional simulation for Stratix GX devices in Verilog HDL, type the following commands:

```
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> ↵
```

Gate-Level Timing Simulation for Stratix GX Devices

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/QuestaSim, you must create these libraries to perform a simulation.

Performing Gate-Level Timing Simulation in VHDL (ModelSim-Altera)

To perform gate-level timing simulation for Stratix GX devices in VHDL, type the following commands:

```
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↵
```

Performing Gate-Level Timing Simulation in VHDL (ModelSim/QuestaSim)

To perform gate-level timing simulation for Stratix GX devices in VHDL, type the following commands:

```
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 ↵
```

Performing Gate-Level Timing Simulation in Verilog HDL (ModelSim-Altera)

To perform gate-level timing simulation for Stratix GX devices in Verilog HDL, type the following commands:

```
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 ↵
```

Performing Gate-Level Timing Simulation in Verilog HDL (ModelSim/QuestaSim)

To perform gate-level timing simulation for Stratix GX devices in Verilog HDL, type the following commands:

```
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 ↵
```

Functional Simulation for Stratix II GX Devices

To perform 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/QuestaSim, 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. The <*alt2gxb entity name*>**.vho** or <*alt2gxb module name*>**.vo** is generated in the current project directory.

The ALT2GXB functional simulation library file generated by the Quartus II software references **stratixiigx_hssi** WYSIWYG atoms.

Performing Functional Simulation in VHDL (ModelSim-Altera)

To perform functional simulation for Stratix II GX devices in VHDL, type the following commands:

```
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> ↵
```

Performing Functional Simulation in VHDL (ModelSim/QuestaSim)

To perform functional simulation for Stratix II GX devices in VHDL, type the following commands:

```
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 Functional Simulation in Verilog HDL (ModelSim-Altera)

To perform functional simulation for Stratix II GX devices in Verilog HDL, type the following commands:

```
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> ↵
```

Performing Functional Simulation in Verilog HDL (ModelSim/QuestaSim)

To perform functional simulation for Stratix II GX devices in Verilog HDL, type the following commands:

```
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> ↵
```

Gate-Level Timing Simulation for Stratix II GX Devices

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/QuestaSim, you must create these libraries to perform a simulation.

Performing Gate-Level Timing Simulation in VHDL (ModelSim-Altera)

To perform gate-level timing simulation for Stratix II GX devices in VHDL, type the following commands:

```
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 ↵
```

Performing Gate-Level Timing Simulation in VHDL (ModelSim/QuestaSim)

To perform gate-level timing simulation for Stratix II GX devices in VHDL, type the following commands:

```
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 ↵
```

Performing Gate-Level Timing Simulation in Verilog HDL ModelSim-Altera)

To perform gate-level timing simulation for Stratix II GX devices in Verilog HDL, type the following commands:

```
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 ↵
```

Performing Gate-Level Timing Simulation in Verilog HDL ModelSim/QuestaSim)

To perform gate-level timing simulation for Stratix II GX devices in Verilog HDL, type the following commands:

```
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 ↵
```

Functional Simulation for Stratix IV GX Devices

To perform a functional simulation of your design that instantiates the ALTGX megafunction, which enables the gigabit transceiver block on Stratix IV devices, compile the **stratixiv_hssi** model file into the **altgx** library.

The **stratixiv_hssi** model file references the **lpm** and **sgate** libraries. If you are using ModelSim/QuestaSim, you must create these libraries to perform a simulation.

Performing Functional Simulation in VHDL (ModelSim-Altera)

To perform functional simulation for Stratix IV devices in VHDL, type the following commands:

```
vcom -work <my design>.vhd <my testbench>.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L stratixiv_hssi work.<my testbench>↵
```

Performing Functional Simulation in VHDL (ModelSim/QuestaSim)

To perform functional simulation for Stratix IV devices in VHDL, type the following commands:

```
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 stratixiv_hssi \
stratixiv_hssi_atoms.vhd stratixiv_hssi_components.vhd ↵
vcom -work <my design>.vhd <my testbench>.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L stratixiv_hssi work.<my testbench> ↵
```

Performing Functional Simulation in Verilog HDL (ModelSim-Altera)

To perform functional simulation for Stratix IV devices in Verilog HDL, type the following commands:

```
vlog -work <my design>.v <my testbench>.v ↵
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L \
stratixiv_hssi_ver work.<my testbench> ↵
```

Performing Functional Simulation in Verilog HDL (ModelSim/QuestaSim)

To perform functional simulation for Stratix IV devices in Verilog HDL, type the following commands:

```
vlog -work lpm_ver 220model.v ↵
vlog -work altera_mf_ver altera_mf.v ↵
vlog -work sgate_ver sgate.v ↵
vlog -work straixiv_hssi_ver stratixiv_hssi_atoms.v ↵
vlog -work <my design>.v <my testbench>.v ↵
vsim -L lpm_ver -L sgate_ver-L stratixiv_hssi_ver work.<my testbench> ↵
```

Gate-Level Timing Simulation for Stratix IV GX Devices

Perform a gate-level timing simulation of your design that includes a Stratix IV transceiver by compiling the **stratixiv_atoms** and **stratixiv_hssi_atoms** model files into the **stratixiv** and **stratixiv_hssi** libraries, respectively.

The **stratixgx_hssi_atoms** model file references the **lpm** and **sgate** libraries. If you are using ModelSim/QuestaSim, you must create these libraries to perform a simulation.

Performing Gate-Level Timing Simulation in VHDL (ModelSim-Altera)

To perform gate-level timing simulation for Stratix IV devices in VHDL, type the following commands:

```
vcom -work <my design>.vho <my testbench>.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L stratixiv -L stratixiv_hssi \
-sdftyp <design instance>=<path to .sdo file>.sdo work.<my testbench> \
-t ps - +transport_int_delays +transport_path_delays ↵
```

Performing Gate-Level Timing Simulation in VHDL (ModelSim/QuestaSim)

To perform gate-level timing simulation for Stratix IV devices in VHDL, type the following commands:

```
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 stratixiv stratixiv_atoms.vhd stratixiv_components.vhd ↵
vcom -work stratixiv_hssi stratixiv_hssi_atoms.vhd \
stratixiv_hssi_components.vhd ↵
vcom -work <my design>.vho <my testbench>.vhd ↵
vsim -L lpm -L altera_mf -L sgate -L stratixiv -L stratixiv_hssi \
-sdftyp <design instance>=<path to .sdo file>.sdo work.<my testbench> \
-t ps +transport_int_delays +transport_path_delays ↵
```

Performing Gate-Level Timing Simulation in Verilog HDL (ModelSim-Altera)

To perform gate-level timing simulation for Stratix IV devices in Verilog HDL, type the following commands:

```
vlog -work <my design>.vo <my testbench>.v ↵
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixiv_ver -L \
stratixiv_hssi_ver work.<my testbench> -t ps +transport_int_delays \
+transport_path_delays ↵
```

Performing Gate-Level Timing Simulation in Verilog HDL (ModelSim/QuestaSim)

To perform gate-level timing simulation for Stratix IV devices in Verilog HDL, type the following commands:

```
vlog -work lpm_ver 220model.v ↵
vlog -work altera_mf_ver altera_mf.v ↵
vlog -work sgate_ver sgate.v ↵
vlog -work stratixiv_ver stratixiv_atoms.v ↵
vlog -work stratixiv_hssi_ver stratixiv_hssi_atoms.v ↵
vlog -work <my design>.vo <my testbench>.v ↵
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixiv_ver \
-L stratixiv_hssi_ver work.<my testbench> -t ps +transport_int_delays \
+transport_path_delays ↵
```

Functional Simulation for Stratix V GX Devices

To perform a functional simulation of your design that instantiates the Custom PHY megafunction, which enables the gigabit transceiver block on Stratix V devices, compile the **stratixv_hssi** model file.

The **stratixiv_hssi** model file references the **lpm** and **sgate** libraries. You must create these libraries to perform a simulation using ModelSim/QuestaSim.

The transceiver module from the MegaWizard Plug-In Manager is created in **Interfaces/Transceiver PHY**. Select **Custom PHY**.

Performing Functional Simulation in VHDL (ModelSim/QuestaSim)

For information about how to perform functional simulation for Stratix V devices in VHDL, refer to "Performing Functional Simulation" on page 2–9.

Performing Functional Simulation in Verilog HDL (ModelSim-Altera)

For information about how to perform functional simulation for Stratix V devices in Verilog HDL, refer to *Performing a Functional Simulation with the ModelSim-Altera Software* and *Compiling Stratix V Libraries* in Quartus II Help.

Performing Functional Simulation in Verilog HDL (ModelSim/QuestaSim)

For information about how to perform functional simulation for Stratix V devices in Verilog HDL, refer to "Performing Functional Simulation" on page 2–13.

Transport Delays

By default, the ModelSim/QuestaSim software filters out all pulses that are shorter than the propagation delay between primitives. Turning on the transport delay options in the ModelSim/QuestaSim 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/QuestaSim software.

The following ModelSim/QuestaSim 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/QuestaSim 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/QuestaSim within the Quartus II software.

For more information, refer to the "Using the NativeLink Feature" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

ModelSim/QuestaSim Error Message Verification

ModelSim/QuestaSim error and warning messages are tagged with a `vsim` or `vcom` code. To determine the cause and resolution for a `vsim` or `vcom` error or warning, use the `verror` command.

For example, ModelSim/QuestaSim 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 Value Change Dump (.vcd) File for the PowerPlay Power Analyzer

To generate a timing Value Change Dump (***.vcd**) file for the PowerPlay Power Analyzer, you must first generate a ***.vcd** script file in the Quartus II software and run the ***.vcd** script file from the ModelSim/QuestaSim or ModelSim-Altera software to generate a timing ***.vcd** file. This timing ***.vcd** file can then be used by PowerPlay for power analysis. 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, click the "+" icon to expand **EDA Tool Settings** and select **Simulation**. The **Simulation** page appears.
3. Choose the appropriate third-party simulation tool (ModelSim/QuestaSim 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/QuestaSim software, perform the following steps:

1. In the ModelSim/QuestaSim 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/QuestaSim or ModelSim-Altera software. If you do not exit the software, the ModelSim/QuestaSim 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 used for generating the waveform view through ModelSim-Altera or ModelSim/QuestaSim.

To view a waveform from a **.wlf** file through ModelSim-Altera or ModelSim/QuestaSim, perform the following steps:

1. Type `vsim` on the command line. The **ModelSim/QuestaSim** or **ModelSim-Altera** dialog box appears.
2. On the File menu, click **Datasets**. The **Datasets Browser** dialog box appears.
3. Click **Open** and browse to the directory that contains your **.wlf** file.
4. Select the **.wlf** file and 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-Altera or ModelSim/QuestaSim directly. The **.vcd** file must first be converted to a **.wlf** file.

8. Use the `vcd2wlf` command to convert the file. For example, type the following on a command-line:

```
vcd2wlf <example>.vcd <example>.wlf ↵
```

9. After you convert the **.vcd** file to a **.wlf** file, follow the procedures for viewing a waveform from a **.wlf** file through ModelSim/QuestaSim.

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 Quartus II Help command line and Tcl API help browser. To access this information, type the following command to start a help browser:

```
quartus_sh --qhelp ↵
```

Generating a Post-Synthesis Simulation Netlist for ModelSim/QuestaSim

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/QuestaSim (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/QuestaSim 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 ↵
```

or

```
set_global_assignment-name EDA_SIMULATION_TOOL "QuestaSim (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/QuestaSim simulator. Specify VHDL or Verilog HDL for the format:

```
quartus_eda <project name> --simulation=on --format=<format> \
--tool=ModelSim --functional ↵
```

or

```
quartus_eda <project name> --simulation=on --format=<format> \
--tool=QuestaSim --functional ↵
```

Generating a Gate-Level Timing Simulation Netlist for ModelSim/QuestaSim

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)" ↵`

  or

  `set_global_assignment -name EDA_SIMULATION_TOOL \`
  `"QuestaSim-Altera (Verilog)" ↵`
- `set_global_assignment -name EDA_SIMULATION_TOOL \`
  `"ModelSim-Altera (VHDL)" ↵`

  or

```
set_global_assignment -name EDA_SIMULATION_TOOL \
"QuestaSim-Altera (VHDL)" ↵
```

- ```
  set_global_assignment -name EDA_SIMULATION_TOOL \
  "ModelSim (Verilog)" ↵
  ```

  or

  ```
  set_global_assignment -name EDA_SIMULATION_TOOL \
  "QuestaSim (Verilog)" ↵
  ```

- ```
  set_global_assignment -name EDA_SIMULATION_TOOL \
  "ModelSim (VHDL)" ↵
  ```

  or

  ```
  set_global_assignment -name EDA_SIMULATION_TOOL \
  "QuestaSim (VHDL)" ↵
  ```

Command Line

Generate a simulation output file for the ModelSim/QuestaSim 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 ↵
```

or

```
quartus_eda <project name> --simulation=on --format=<format> \
--tool=QuestaSim ↵
```

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.

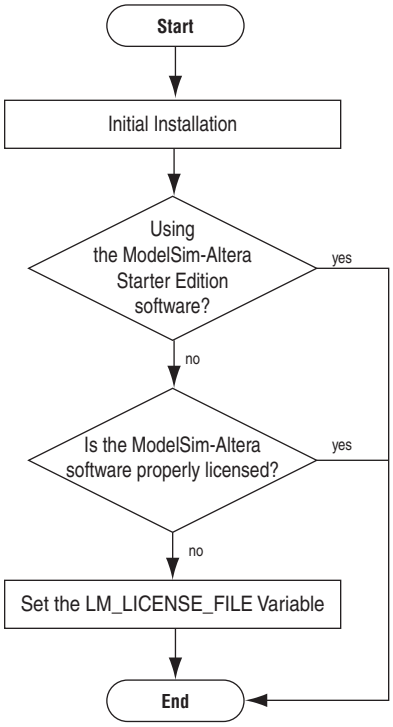
The USB software guard is not supported by versions earlier than Mentor Graphics ModelSim software 5.8d.

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 2–1 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 2–1. 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 the *Altera Software Installation and Licensing* manual.

Conclusion

Using the ModelSim/QuestaSim and ModelSim-Altera simulation software within the Altera FPGA design flow enables Altera software users to easily and accurately perform 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/QuestaSim and ModelSim-Altera software helps ensure design functionality and success and, ultimately, a quick time-to-market.

Document Revision History

Table 2–1 shows the revision history for this chapter.

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

Date	Version	Changes Made
July 2010	10.0.0	■ Removed simulation library tables and linked to Quartus II Help ■ Added other links to Quartus II Help and ModelSim-Altera Help where appropriate and removed redundant information ■ Added QuestaSim support ■ Added Stratix V simulation information ■ Minor editorial changes throughout ■ Removed Referenced Documents section
November 2009	9.1.0	■ Removed NativeLink information and referenced new *Simulating Designs with EDA Tools* chapter ■ Added Stratix IV transceiver simulation section ■ Reformatted transceiver simulation sections ■ Text edits throughout chapter
March 2009	9.0.0	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"
November 2008	8.1.0	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

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

Date	Version	Changes Made
May 2008	8.0.0	Updated the following: ■ "Altera Design Flow with ModelSim-Altera or ModelSim Software" on page 2–3 ■ "Simulation Libraries" on page 2–4 ■ "Simulation Netlist Files" on page 2–11 ■ "Perform Simulation Using ModelSim-Altera Software" on page 2–15 ■ "Perform Simulation Using ModelSim Software" on page 2–33 ■ "Simulating Designs that Include Transceivers" on page 2–57 ■ "Using the NativeLink Feature with ModelSim-Altera or ModelSim Software" on page 2–63 ■ "Generating a Timing VCD File for PowerPlay" on page 2–68

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

Take an online survey to provide feedback about this handbook chapter.

3. Synopsys VCS and VCS MX Support

QII53002-10.0.0

This chapter describes how to use the Synopsys VCS and VCS MX software to simulate designs that target Altera® FPGAs. This chapter provides step-by-step instructions about how to perform functional simulations, post-synthesis simulations, and gate-level timing simulations.

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 includes the following topics:

- "Software Requirements"
- "Using the VCS or VCS MX Software in the Quartus II Design Flow" on page 3–1
- "Common VCS and VCS MX Software Compiler Options" on page 3–8
- "Using DVE" on page 3–8
- "Debugging Support Command-Line Interface" on page 3–8
- "Simulating Designs that Include Transceivers" on page 3–9
- "Transport Delays" on page 3–13
- "Using NativeLink with the VCS or VCS MX Software" on page 3–13
- "Generating a Timing .vcd File for the PowerPlay Power Analyzer" on page 3–13
- "Viewing a Waveform from a .vpd or .vcd File" on page 3–14
- "Scripting Support" on page 3–15

Software Requirements

To simulate your design using the Synopsys VCS or VCS MX software, you must first set up the Altera libraries. These libraries are installed with the Quartus® II software.

For more information about installing the software and the directories created during the Quartus II software installation, refer to the *Altera Software Installation and Licensing* manual.

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:

- Functional simulation
- Post-synthesis simulation
- Gate-level timing simulation

Refer to the "PLD Design Flow" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook* for the Quartus II software design flow.

Compiling 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 compile all libraries required by functional and gate-level timing simulations.

For more information, refer to the "EDA Simulation Library Compiler" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

Functional Simulations

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.

For a list of the functional simulation library files in the Quartus II directory, refer to *Altera Functional Simulation Libraries* in Quartus II Help.

Functional Simulation for Verilog HDL Designs

The following VCS command performs a functional simulation for Verilog HDL designs with one of the libraries listed in *Altera Functional Simulation Libraries* in Quartus II Help:

```
vcs -R <testbench>.v <design name>.v –v <altera_library1>.v -v <altera_library2>.v ↵
```

If you have already generated the option file (**simlib_comp.vcs**) from "Compiling Libraries Using the EDA Simulation Library Compiler" on page 3–2, 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 functional simulation for Verilog HDL designs:

```
# Create Libraries Directories
mkdir <Directory_to_store_compiled_altera_library1>
mkdir <Directory_to_store_compiled_altera_library2>

# Create Work Directory
mkdir <Directory_to_store_compiled_design_and_testbench_files>

# Compilation
# (Before this step, make sure the mapped file "synopsys_sim.setup" was created.)
# Libraries Compilation
vlogan -work <altera_library1_name> <altera_library1>.v
vlogan -work <altera_library2_name> <altera_library2>.v

# Design and Test bench files Compilation
vlogan -work <work_library_name> <design>.v <testbench>.v

# Eleboration
```

```
vcs -debug_all <work_library_name>.<testbench_top_level_module>

# Run Simulation
simv -gui
```

The **synopsys_sim.setup** file contains the following mapping commands to map the libraries:

```
<altera_library1_name> : <Directory_to_store_compiled_altera_library1>
<altera_library2_name> : <Directory_to_store_compiled_altera_library2>
<work_library_name> : <Directory_to_store_compiled_design_and_testbench_files>
```

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.

If you are targeting a Stratix V device, compile the following files in the **quartus/eda/sim_lib/synopsys** directory:

stratixv_atoms_ncrypt.v
stratixv_hssi_atoms_ncrypt.v
stratixv_pcie_hip_atoms_ncrypt.v

These files contain IEEE encrypted Verilog models.

Compile **stratixv_pcie_hip_atoms_ncrypt.v** with the SystemVerilog option.

Also, compile the following files in the **quartus/eda/sim_lib** directory:

stratixv_atoms.v
stratixv_hssi_atoms.v
stratixv_pcie_hip_atoms.v

The PCIe® files are required only if you are using the PCIe HIP.

Functional Simulation for VHDL Designs

For VHDL designs, you need to use VCS MX software to perform all three types of simulations. The following commands are for performing a functional simulation for VHDL designs with one of the libraries listed in *Altera Functional Simulation Libraries* in Quartus II Help:

```
# Create Libraries Directories
mkdir <Directory_to_store_compiled_altera_library1>
mkdir <Directory_to_store_compiled_altera_library2>

# Create Work Directory
mkdir <Directory_to_store_compiled_design_and_testbench_files>

# Compilation
# (Before this step, make sure the mapped file "synopsys_sim.setup" is created.)
# Libraries Compilation
vhdlan -work  <altera_library1_name> <altera_library1>.vhd
vhdlan -work  <altera_library2_name> <altera_library2>.vhd

# Design and Test bench files Compilation
vhdlan -work <work_library_name> <design>.vhd <testbench>.vhd

# Eleboration
vcs -debug_all <work_library_name>.<testbench_top_level_module>
```

```
# Run Simulation

simv -gui
```

The **synopsys_sim.setup** file contains the following mapping commands to map the libraries:

```
<altera_library1_name> : <Directory_to_store_compiled_altera_library1>
<altera_library2_name> : <Directory_to_store_compiled_altera_library2>
<work_library_name> : <Directory_to_store_compiled_design_and_testbench_files>
```

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.

If you have generated the Altera libraries with the EDA Simulation Library Compiler, ignore the steps **# Create Libraries Directories** and **# Libraries Compilation**.

If you are targeting a Stratix V device, compile the following files in the **quartus/eda/sim_lib/synopsys** directory:

stratixv_atoms_ncrypt.v
stratixv_hssi_atoms_ncrypt.v
stratixv_pcie_hip_atoms_ncrypt.v

These files contain IEEE encrypted Verilog models suitable for VHDL/Verilog co-simulation. You need a co-simulation license from Synopsys to use these models.

Compile these encrypted Verilog files before you compile any VHDL files.

- Compile **stratixv_pcie_hip_atoms_ncrypt.v** with the SystemVerilog option.

Also, compile the following files in the **quartus/eda/sim_lib** directory:

stratixv_atoms.vhd
stratixv_components.vhd
stratixv_hssi_components.vhd
stratixv_hssi_atoms.vhd
stratixv_pcie_components.vhd
stratixv_pcie_hip_atoms.vhd

The PCIe files are required only if you are using the PCIe HIP.

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.

For more information, refer to the "Generating Post-Synthesis Simulation Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

You cannot perform post-synthesis or post-fit simulation if you are targeting the Stratix V device family.

Post-Synthesis Simulation for Verilog HDL Designs

The following VCS command shows the command-line syntax used to perform a post-synthesis simulation with the appropriate device family library listed in *Altera Functional Simulation Libraries* in Quartus II Help.

```
vcs -R <testbench> <post synthesis netlist> -v <altera_library1> ↵
```

If you have already generated the option file (**simlib_comp.vcs**) as described in "Compiling Libraries Using the EDA Simulation Library Compiler" on page 3–2, modify the **simlib_comp.vcs** file to add the testbench and post synthesis netlist, and then 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_altera_library1>
mkdir <Directory_to_store_compiled_altera_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_sim.setup" is created.)
# Libraries Compilation
vlogan -work  <altera_library1_name> <altera_library1>.v
vlogan -work  <altera_library2_name> <altera_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_sim.setup** file contains the following commands to map the libraries:

```
<altera_library1_name> : <Directory_to_store_compiled_altera_library1>
<altera_library2_name> : <Directory_to_store_compiled_altera_library2>
<work_library_name> :
   <Directory_to_store_compiled_post_synthesis_netlist_and_testbench_files>
```

Post-Synthesis Simulation for VHDL Designs

The following VCS MX commands show the command-line syntax used to perform a post-synthesis simulation with the appropriate device family library listed in *Altera Post-Fit Libraries* in Quartus II Help.

```
# Create Libraries Directories
mkdir <Directory_to_store_compiled_altera_library1>
mkdir <Directory_to_store_compiled_altera_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_sim.setup" is created.)
# Libraries Compilation
vhdlan -work  <altera_library1_name> <altera_library1>.vhd
vhdlan -work  <altera_library2_name> <altera_library2>.vhd
```

```
# Design and Test bench files Compilation
vhdlan -work <work_library_name> <post_synthesis_netlist>.vho <test_bench>.vhd
# Eleboration
vcs -debug_all <work_library_name>.<testbench_top_level_module>
# Run Simulation
simv -gui
```

The **synopsys_sim.setup** file contains the following commands to map the libraries:

```
<altera_library1_name> : <Directory_to_store_compiled_altera_library1>
<altera_library2_name> : <Directory_to_store_compiled_altera_library2>
<work_library_name> :
    <Directory_to_store_compiled_post_synthesis_netlist_and_testbench_files>
```

If you have generated the Altera libraries as described in "Compiling Libraries Using the EDA Simulation Library Compiler" on page 3–2, ignore the steps `# Create Libraries Directories` and `# Libraries Compilation`.

Gate-Level Timing Simulation

A gate-level timing simulation verifies the functionality and timing 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.

For information about how to generate a gate-level timing simulation netlist, refer to the "Generating Gate-Level Timing Simulation Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

For a list of the gate-level timing simulation library files in the Quartus II directory, refer to *Altera Post-Fit Libraries* in Quartus II Help.

You cannot perform post-synthesis or post-fit simulation if you are targeting the Stratix V device family.

Gate-Level Timing Simulation for Verilog HDL Designs

For gate-level timing simulation, follow the steps in "Post-Synthesis Simulation for Verilog HDL Designs" on page 3–5.

You do not have to specify the Standard Delay Output File (***.sdo**) file because it is already specified in the Verilog Output File (***.vo**) file. However, the ***.sdo** file must be in the same directory as the simulator executable file **simv** generated by VCS.

Gate-Level Timing Simulation for VHDL Designs

For gate-level timing simulation, follow the steps in "Post-Synthesis Simulation for VHDL Designs" on page 3–5.

For VHDL, the ***.sdo** file must be specified in the `simv` command as follows:

```
simv -xlrm -gui -sdf typ:<testbench module name>/<design instnace
name>.sdo ↵
```

Adding the `-xlrm` switch avoids the errors that occur when the timing arcs in SDO do not match Altera VHDL simulation models as per the IEEE VITAL ASIC standard. However, adding this switch reduces timing accuracy, as it may cause some SDO delays to be ignored. Therefore, Altera recommends generating the Verilog HDL simulation output netlist (**.vo**) if you want to perform gate-level timing simulation.

Disabling Timing Violation on Registers

In certain situations, the timing violations can be ignored and you can disable the "**X**" propagation that happens when there are 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 that applies to all registers in the design is **On**, which means a register outputs "**X**" whenever a timing violation occurs. To disable "**X**" propagation due to a 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> ↵
```

Performing 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 3–1 lists some of the available options.

Table 3–1. VCS Software Compiler Options

Library	Description
`-R`	Runs the executable file immediately.
`-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.
`-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.
`+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 DVE

Design Viewpoint Editor (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 command shows the command-line syntax for simulating in DVE:

`simv -gui` (for Verilog HDL simulations)

However, to open the GUI with these commands, you must enable the use of Unified Command Line Interface (UCLI) and DVE when performing elaboration. To enable UCLI and DVE, enter the following command:

`vcs -debug_all` (for Verilog HDL 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 UCLI 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 `+ucli` run-time option. Use the VCS software UCLI 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 +ucli ↵
```

The `+ucli` command takes an optional number argument that specifies the level of debugging capability. As the optional debugging capability is increased, there is an increase in simulation time.

For more information about the `+ucli` options, refer to the *VCS User Guide* included in the VCS software installation.

For the design examples to run gate-level timing simulations in VHDL or Verilog HDL language, refer to Synopsys VCS Simulation Design Example.

Simulating Designs that Include Transceivers

If your design includes Arria®, Arria II, Cyclone® IV, HardCopy® IV, Stratix®, Stratix II, Stratix IV, or Stratix V transceivers, you must compile additional library files to perform functional or gate-level timing simulations.

For high-speed simulation, you must select **ps** in the **Resolution** list for your simulator resolutions (**Design** tab of the **Start Simulation** dialog box). If you choose slower than **ps**, the high-speed simulation might fail.

If your design contains PCI Express hard IP, refer to the "Simulate the Design" section in the *PCI Express Compiler User Guide*.

Functional Simulation for Stratix GX Devices

To perform a functional simulation of your design that instantiates the ALTGXB megafunction, enabling the gigabit transceiver block 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. You must create these libraries to perform a simulation.

Compiling Library Files for Functional 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 ↵
```

Gate-Level Timing Simulation for Stratix GX Devices

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 Gate-Level Timing 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 ↵
```

Functional Simulation for Stratix II GX Devices

Functional simulation for Stratix II GX devices is similar to functional simulation for Arria GX devices. To simulate the transceiver 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, enabling 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. 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. The <*alt2gxb entity name*>**.vho** file or <*alt2gxb module name*>**.vo** file is generated in the current project directory.

The ALT2GXB functional simulation library file generated by the Quartus II software references **stratixiigx_hssi wysiwyg atoms**.

Compiling Library Files for 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 ↵
```

Gate-Level Timing Simulation for Stratix II GX Devices

Gate-level timing simulation for Stratix II GX devices is similar to gate-level timing simulation for Arria GX devices. To simulate the transceiver 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. You must create these libraries to perform a simulation.

Compiling Library Files for 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 ↵
```

Functional Simulation for Stratix IV GX Devices

Functional simulation for Stratix IV devices is similar to functional simulation for Arria II, Cyclone IV, and HardCopy IV devices. To simulate transceivers in Arria II, Cyclone IV, and HardCopy IV devices, you only have to replace the **stratixiv_hssi** model file with the **arriaii_hssi**, **cycloneiv_hssi**, and **hardcopyiv_hssi** model files, respectively.

To perform a functional simulation of your design that instantiates the ALTGX megafunction, enabling the gigabit transceiver block on Stratix IV devices, compile the **stratixiv_hssi** model file into the **stratixiv_hssi** library,

The **stratixiv_hssi_atoms** model file references the **lpm** and **sgate** libraries. You must create these libraries to perform a simulation.

Compiling Library Files for Functional Simulation in Verilog HDL

To compile the libraries necessary for functional simulation of a Verilog HDL design targeting a Stratix IV device, type the following command at the VCS command prompt:

```
vcs -R <testbench>.v <altgx>.v -v \ stratixiv_hssi_atoms.v -v sgate.v \
-v 220model.v -v altera_mf.v ↵
```

Gate-Level Timing Simulation for Stratix IV GX Devices

Gate-level timing simulation for Stratix IV devices is similar to gate-level timing simulation for Arria II, Cyclone IV, and HardCopy IV devices. To simulate transceivers in Arria II, Cyclone IV, and HardCopy IV devices, you only have to replace the **stratixiv_hssi** model file with the **arriaii_hssi**, **cycloneiv_hssi**, and **hardcopyiv_hssi** model files, respectively.

To perform a gate-level timing simulation of your design that includes a Stratix IV transceiver, compile **stratixiv_atoms** and **stratixiv_hssi_atoms** into the **stratixiv** and **stratixiv_hssi** libraries, respectively.

To perform a gate-level timing simulation of your design that includes a Stratix IV transceiver, compile **stratixiv_atoms** and **stratixiv_hssi_atoms** into the **stratixiv** and **stratixiv_hssi** libraries, respectively.

The **stratixiv_hssi_atoms** model file references the **lpm** and **sgate** libraries. You must create these libraries to perform a simulation.

Compiling Library Files for Gate-Level Timing Simulation in Verilog HDL

To compile the libraries necessary for timing simulation of a Verilog HDL design targeting a Stratix IV device, type the following command at the VCS command prompt:

```
vcs -R <testbench>.v <gate-level netlist>.vo -v stratixiv_atoms.v \
-v stratixiv_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
```

Functional Simulation for Stratix V GX Devices

Functional simulation for Stratix V devices is similar to functional simulation for Arria II, Cyclone IV, HardCopy IV, and Stratix IV devices. To simulate transceivers in Arria II, Cyclone IV, HardCopy IV, and Stratix IV devices, you only have to replace the **stratixv_hssi** model file with the **arriaii_hssi**, **cycloneiv_hssi**, **hardcopyiv_hssi**, and **stratixiv_hssi** model files, respectively.

The **stratixv_hssi_atoms** model file references the **lpm** and **sgate** libraries. You must compile these libraries to perform a simulation.

The transceiver module from the MegaWizard Plug-In Manager is created in **Interfaces/Transceiver PHY**. Select **Custom PHY**.

Compiling Library Files for Functional Simulation

To compile the libraries necessary for functional simulation of a Verilog HDL or VHDL design targeting a Stratix V device, type the following commands at the VCS command prompt:

```
mkdir -p ./stratixv ↵
mkdir -p ./stratixv_pcie_hip ↵
mkdir -p ./stratixv_hssi ↵
mkdir -p ./work ↵

vlogan +v2k -work stratixv \
$QUARTUS_ROOTDIR/eda/sim_lib/synopsys/stratixv_atoms_ncrypt.v ↵

vlogan +v2k -work stratixv_hssi \
$QUARTUS_ROOTDIR/eda/sim_lib/synopsys/stratixv_hssi_atoms_ncrypt.v ↵

vlogan -sverilog -work stratixv_pcie_hip \
$QUARTUS_ROOTDIR/eda/sim_lib/synopsys/stratixv_pcie_hip_atoms_ncrypt.v ↵

vhdlan -work stratixv_hssi \
$QUARTUS_ROOTDIR/eda/sim_lib/stratixv_hssi_components.vhd ↵

vhdlan -work stratixv_hssi \
$QUARTUS_ROOTDIR/eda/sim_lib/stratixv_hssi_atoms.vhd ↵

vcs test ↵
./simv ↵
```

The PCIe files are required only if you are using the PCIe HIP.

For VHDL, you must compile the Verilog HDL files first.

In addition to the top-level variant wrapper, <*variant*>**.v**, you also get a simulation files subdirectory, <*variant*>**_sim/**. All Verilog (**.v**) and SystemVerilog (**.sv**) files in the simulation directory must also be compiled into the simulation project.

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.

For more information, refer to the "Using the NativeLink Feature" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

Generating a Timing .vcd File for the PowerPlay Power Analyzer

To generate a timing Verilog Value Change Dump File (***.vcd**) 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 analysis.

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**, click **Simulation.** On the **Simulation** page, in the **Tool name** list, select **VCS** and turn on the **Generate Value Change Dump (VCD) file script** option.
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. 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 finished and the *<revision_name>***.vcd** file is generated in the simulation directory.

The **.vcd** file is not supported in the VCS MX software.

For more detailed information about using the timing **.vcd** file for power analysis, 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 Virtual Panoramic Display (**.vpd**) file is automatically generated when your simulation is finished. The **.vpd** file is not readable. It is used for generating the waveform view through DVE. You can view your waveform result in DVE if you have created a **.vpd** or **.vcd** file.

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 DVE directly. The **.vcd** file must first be converted to a **.vpd** file. To convert the 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 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 the following 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 when you compile your design or as part of a Tcl script that compiles your design:

```
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 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.

Document Revision History

Table 3–2 shows the revision history for this chapter.

Table 3–2. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Linked to Quartus II Help where appropriate ■ Added Stratix V simulation information ■ Minor text edits ■ Removed VirSim references ■ Removed Referenced Documents section
November 2009	9.1.0	■ Removed NativeLink information and referenced new *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook* ■ Added "RTL Functional Simulation for Stratix IV Devices" and "Gate-Level Timing Simulation for Stratix IV Devices" sections ■ Minor text edits
March 2009	9.0.0	■ Added support for Synopsys VCS MX software ■ Changed chapter title to "Synopsys VCS and VCS MX Support" ■ Major revision to "Compiling Libraries Using the EDA Simulation Library Compiler" on page 4–2 ■ Major revision to "RTL Functional Simulations" on page 4–2 ■ Added Table 3–4 on page 3–10 and Table 3–5 on page 3–11 ■ Added new section "Using DVE" on page 4–7 ■ Added new section "Generating a Simulation Script from the EDA Netlist Writer" on page 3–16 ■ Added new section "Viewing a Waveform from a .vpd or .vcd File" on page 4–13
November 2008	8.1.0	■ 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

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

Take an online survey to provide feedback about this handbook chapter.

4. Cadence NC-Sim Support

QII53003-10.0.0

This chapter describes the basic NC-Sim, NC-Verilog, and NC-VHDL functional, post-synthesis, and gate-level timing simulations.

The Cadence Incisive verification platform includes NC-Sim, NC-Verilog, NC-VHDL, Verilog HDL, and VHDL desktop simulators.

This chapter is a "getting started" guide to using the Cadence Incisive verification platform in Altera® FPGA design flows. This chapter also describes the location of the simulation libraries and how to automate simulations.

This chapter includes the following topics:

- "Software Requirements"
- "Simulation Flow Overview"
- "Functional Simulation" on page 4–3
- "Post-Synthesis Simulation" on page 4–7
- "Gate-Level Timing Simulation" on page 4–8
- "Simulating Designs that Include Transceivers" on page 4–10
- "Using the NativeLink Feature with NC-Sim" on page 4–18
- "Generating a Timing VCD File for the PowerPlay Power Analyzer" on page 4–18
- "Viewing a Waveform from a .trn File" on page 4–19
- "Scripting Support" on page 4–20

Software Requirements

To simulate your design using the NC-Sim software, you must first set up the Altera libraries. These libraries are installed with the Quartus II software.

For more information about installing the software and directories created during the Quartus II software installation, refer to the *Altera Software Installation and Licensing* manual.

Simulation Flow Overview

The Incisive verification platform supports the following simulation flows:

- Functional Simulation
- Post-Synthesis Simulation
- Gate-Level Timing Simulation
- Using the NativeLink Feature with NC-Sim

Functional simulation verifies the functionality of your design. When you perform a 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 "Functional Simulation" on page 4–3 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 4–7 for more information about how to perform this simulation.

After performing place-and-route, the Quartus II software generates a Verilog HDL 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 4–8 for more information about how to perform this simulation.

Operation Modes

You can use either the GUI mode or the command-line mode to simulate your design in the NC simulators.

To start the Incisive simulators in GUI mode, type the following command at a command prompt:

```
nclaunch ↵
```

To simulate in command-line mode, use the programs shown in Table 4–1.

Table 4–1. 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 "Functional Simulation" on page 4–3, "Post-Synthesis Simulation" on page 4–7, and "Gate-Level Timing Simulation" on page 4–8.

For high-speed simulation, you must select **ps** in the **Resolution** list for your simulator resolutions (**Design** tab of the **Start Simulation** dialog box). If you choose slower than **ps**, the high-speed simulation might fail.

Complete the following tasks:

1. 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.

2. 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.

3. 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**.

4. Add signals to your waveform.

   Specify which signals to view in your waveform using a simulation history manager (SHM) database.

5. Simulate your design.

   Run the simulator with the **ncsim** program (command-line mode) or by clicking **Run** in the SimVision Console window.

Compiling Libraries Using the EDA Simulation Library Compiler

The EDA Simulation Library Compiler compiles Verilog HDL and VHDL simulation libraries for all Altera devices and supported third-party simulators. You can compile all libraries required by functional and gate-level simulation with this tool.

For more information about this tool, refer to the "EDA Simulation Library Compiler" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

Functional Simulation

The following sections provide detailed instructions for performing functional simulation using the Quartus II software and the Cadence Incisive verification platform tools.

For the Altera Behavioral Simulation Models, refer to *Altera Functional Simulation Libraries* in Quartus II Help.

Creating Libraries

To create libraries, perform the following steps:

1. Create a directory for the work library and any other altera libraries you require by typing the following command at a command prompt:

   ```
   mkdir <library name> ↵
   ```

   Examples:

   ```
   mkdir worklib ↵
   mkdir altera_mf ↵
   ```

2. Using a text editor, create a **cds.lib** file and add the following line to it:

   ```
   DEFINE <library name> <physical directory path>
   ```

   Examples:

   ```
   DEFINE worklib ./worklib
   DEFINE altera_mf ./altera_mf
   ```

For information about creating a **cds.lib** file in GUI mode, refer to *Performing a Functional Simulation with the NCSim Software* in Quartus II Help.

For VHDL Designs

If you are targeting a Stratix V device, compile the following files in the **quartus/eda/sim_lib/cadence** directory:

stratixv_atoms_ncrypt.v
stratixv_hssi_atoms_ncrypt.v
stratixv_pcie_hip_atoms_ncrypt.v

These files contain IEEE encrypted Verilog models suitable for VHDL/Verilog co-simulation. You need a co-simulation license from Cadence to use these models in VHDL.

Compile these encrypted Verilog files before you compile any VHDL files.

Compile **stratixv_pcie_hip_atoms_ncrypt.v** with the SystemVerilog option.

Also, compile the following files in the **quartus/eda/sim_lib** directory:

stratixv_atoms.vhd
stratixv_components.vhd
stratixv_hssi_components.vhd
stratixv_pcie_components.vhd
stratixv_hssi_atoms.vhd
stratixv_pcie_hip_atoms.vhd

The PCIe® files are required only if you are using the PCIe HIP.

For Verilog HDL Designs

If you are targeting a Stratix V device, compile the following files in the <**quartus/eda/sim_lib/cadence** directory:

stratixv_atoms_ncrypt.v
stratixv_hssi_atoms_ncrypt.v
stratixv_pcie_hip_atoms_ncrypt.v

These files contain IEEE encrypted Verilog models.

Compile **stratixv_pcie_hip_atoms.v** with the SystemVerilog option.

Also, compile the following files in the **quartus/eda/sim_lib** directory:

stratixv_atoms.v
stratixv_hssi_atoms.v
stratixv_pcie_hip_atoms.v

The PCIe files are required only if you are using the PCIe HIP.

Compiling Source Code

To compile from your source code from the command line, type 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 your design and testbench. If your design uses LPM, Altera megafunctions, or Altera primitives, you must also compile the Altera-provided functional models. The commands in Example 4–1 and Example 4–2 show an example of each.

Example 4–1. Compile in Verilog HDL

```
ncvlog -WORK lpm 220model.v
ncvlog -WORK altera_mf altera_mf.v
ncvlog -WORK altera altera_primitives.v
ncvlog -WORK altera altera_primitives.v
ncvlog -WORK work toplevel.v testbench.v
```

Example 4–2. Compile 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
ncvhdl -V93 -WORK work toplevel.vhd testbench.vhd
```

For information about compiling in GUI mode, refer to *Performing a Functional Simulation with the NCSim Software* in Quartus II Help.

Elaborating Your Design

Before you can simulate your design, you must define the design hierarchy in a process called elaboration. The Incisive simulator elaborates your design with the language-independent ncelab program. 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.

To elaborate your Verilog HDL or VHDL design from the command line, type the following command:

```
ncelab [options][<library>.<testbench module name]
```

Example:

```
ncelab -access +rwc work.testbench_module
```

Adding the option `-access +rwc` allows signals to be viewed in the Waveform window.

If your design includes high-speed signals, you might have to add the following pulse reject options with the ncelab command:

```
ncelab -access +rwc work.testbench_module -PULSE_R 0 -PULSE_INT_R 0
```

For more information about the pulse reject options, refer to the *SDF Annotate Guide* from Cadence.

For information about elaborating your design in GUI mode, refer to *Performing a Functional Simulation with the NCSim Software* in Quartus II Help.

Simulating Your 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 and then loads other intermediate objects referenced by the snapshot. If you enable interactive debugging, **ncsim** can 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, the SHM database, or the **.vcd** file.

To perform functional simulation of your Verilog HDL or VHDL design at the command line, type the following command:

```
ncsim [options][<library>.<testbench module name>]
```

Example:

```
ncsim -gui work.testbench_module
```

Adding the option `-gui` opens the SimVision window for running your simulation.

For information about performing a functional simulation in GUI mode, refer to *Performing a Functional Simulation with the NCSim Software* in Quartus II Help.

Post-Synthesis Simulation

The following sections provide detailed instructions for performing post-synthesis simulation with the Incisive platform software and output files and simulation files from the Quartus II software.

You cannot perform post-synthesis or post-fit simulation if you are targeting the Stratix V device family.

For a list of the gate-level simulation models, refer to *Altera Post-Fit Libraries* in Quartus II Help.

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, refer to the "Generating Post-Synthesis Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

Creating 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**

Refer to "Creating Libraries" on page 4–4 for instructions about creating libraries.

Compiling Project Files and Libraries

Compile the project files and libraries into your work directory using the **ncvlog**, **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 "Compiling Source Code" on page 4–5 for instructions about compiling.

Elaborating Your Design

Elaborate your design using the **ncelab** program as described in "Elaborating Your Design" on page 4–6.

Simulating Your Design

Simulate your design using the **ncsim** program as described in "Simulating Your Design" on page 4–6.

Gate-Level Timing Simulation

The following sections provide detailed instructions for performing a timing simulation using the Quartus II output files, simulation libraries, and Cadence NC tools.

You cannot perform post-synthesis or post-fit simulations if you are targeting the Stratix V device family.

For a list of the gate-level simulation models, refer to *Altera Post-Fit Libraries* in Quartus II Help.

For details about how to perform gate-level timing simulation using the Quartus II software and the Cadence NC-Sim software, refer to *Performing a Timing Simulation with the NCSim Software* in Quartus II Help.

Generating a Gate-Level Timing Simulation Netlist

To perform a gate-level timing simulation, your design should provide the NC-Sim software with 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**, refer to the "Generating Gate-Level Timing Simulation Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

Disabling 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 can be ignored and disabled.

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.

```
set_instance_assignment -name X_ON_VIOLATION_OPTION OFF –to \
<register_name>
```

This Tcl command is also stored in the **.qsf** file.

Creating 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 about creating libraries, refer to "Creating Libraries" on page 4–4.

Compiling Project Files and Libraries

Compile the project files and libraries into your work directory with 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 "Compiling Source Code" on page 4–5.

Elaborating Your Design

When performing elaboration with the Quartus II-generated Verilog HDL netlist file, the **.sdo** file is read automatically. The **ncelab** executable recognizes the embedded system task `$sdf_annotate` and automatically compiles and annotates the **.sdo** file (runs **ncsdfc** automatically).

The **.sdo** file should be located in the same directory where you perform an elaboration or simulation, because the `$sdf_annotate` task references the **.sdo** file without using a full path. If you are starting an elaboration or simulation from a different directory, you can either comment out the `$sdf_annotate` and annotate the **.sdo** file with the GUI, or add the full path of the **.sdo** file.

Refer to "Elaborating Your Design" on page 4–6 for elaboration instructions.

VHDL netlist files do not contain system task calls to locate your **.sdf** file; therefore, you must compile the Standard **.sdo** file manually. For information about compiling the **.sdo** file, refer to "Compiling the .sdo File (VHDL Only) in Command-Line Mode" and "Compiling the .sdo File (VHDL Only) in GUI Mode".

Compiling the .sdo File (VHDL Only) in Command-Line Mode

To annotate the **.sdo** file timing data from the command line, perform the following steps:

1. Compile the **.sdo** 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 4–3 shows an example of an SDF command file.

Example 4–3. 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 .sdo File (VHDL Only) in GUI Mode

To compile the .sdo file in GUI mode, refer to *Performing a Timing Simulation with the NCSim Software* in Quartus II Help.

Simulating Your Design

Simulate your design using the **ncsim** program as described in "Simulating Your Design" on page 4–6.

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 Arria®, Arria II, Cyclone IV, HardCopy IV, Stratix, Stratix II, or Stratix IV transceivers, you must compile additional library files to perform functional or gate-level timing simulations.

For high-speed simulation, you must select **ps** in the **Resolution** list for your simulator resolutions (**Design** tab of the **Start Simulation** dialog box). If you choose slower than **ps**, the high-speed simulation might fail.

If your design contains PCI Express® hard IP, refer to the "Simulate the Design" section in the *PCI Express Compiler User Guide.*

Functional Simulation for Stratix GX Devices

To perform a 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. You must create these libraries to perform a simulation.

Compiling Library Files for Functional 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 4–4 at the NC-Sim command prompt.

Example 4–4. 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 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 4–5 at the NC-Sim command prompt.

Example 4–5. 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>
```

Gate-Level Timing Simulation for Stratix GX Devices

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 Gate-Level Timing Simulation in VHDL

To compile the libraries necessary for a timing 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 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 Gate-Level Timing Simulation in Verilog HDL

To compile the libraries necessary for a timing 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 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
```

Functional Simulation for Stratix II GX Devices

Functional simulation of Stratix II GX devices is similar to functional simulation of Arria GX devices. Example 4–9 on page 4–13 and "Compile Libraries Commands for Functional Simulation in Verilog HDL" on page 4–13 show only the functional simulation for designs that include transceivers in Stratix II GX devices. To simulate transceivers in Arria GX devices, 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, 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 4–8 shows the **cds.lib** file.

Example 4–8. 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. 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. The *<alt2gxb entity name>*.**vho** or *<alt2gxb module name>*.**vo** is generated in the current project directory.

☞ The ALT2GXB functional simulation library file generated by the Quartus II software references **stratixiigx_hssi** WYSIWYG atoms.

Compiling Library Files for Functional 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 4–9 at the NC-Sim command prompt.

Example 4–9. 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 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 4–10 at the NC-Sim command prompt.

Example 4–10. 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>
```

Gate-Level Timing Simulation for Stratix II GX Devices

Stratix II GX functional simulation is similar to Arria GX functional simulation. Example 4–11 on page 4–14 and Example 4–12 on page 4–14 show 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 "Functional Simulation for Stratix II GX Devices" on page 4–12.

The **stratixiigx_hssi_atoms** model file references the **lpm** and **sgate** libraries. You must create these libraries to perform a simulation.

Compiling Library Files for Gate-Level Timing 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 4–11 at the NC-Sim command prompt.

Example 4–11. 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 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 commands shown in Example 4–12 at the NC-Sim command prompt.

Example 4–12. 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
```

Functional Simulation for Stratix IV GX Devices

Functional simulation for Stratix IV devices is similar to functional simulation for Arria II, Cyclone IV, and HardCopy IV devices. Example 4–14 shows only the functional simulation for designs that include transceivers in Stratix IV devices. To simulate transceivers in Arria II, Cyclone IV, and HardCopy IV devices, replace the **stratixiv_hssi** model file with the **arriaii_hssi**, **cycloneiv_hssi**, and **hardcopyiv_hssi** model files, respectively.

To perform a functional simulation of your design that instantiates the ALTGX megafunction, edit your **cds.lib** file so that all of the libraries point to the work library, and compile the **stratixiv_hssi** model file into the **stratixiv_hssi** library.

When compiling the library files, you can safely ignore the following warning message:

```
"Multiple logical libraries mapped to a single location"
```

Example 4–13 shows the **cds.lib** file.

Example 4–13. 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 stratixiv_hssi ./ncsim_work
DEFINE stratixiv ./ncsim_work
DEFINE lpm ./ncsim_work
DEFINE sgate ./ncsim_work
```

The **stratixiv_hssi_atoms** model file references the **lpm** and **sgate** libraries. You must create these libraries to perform a simulation.

Compiling Library Files for Functional Simulation in VHDL

To compile the libraries necessary for functional simulation of a VHDL design targeting a Stratix IV device, type the commands shown in Example 4–14 at the NC-Sim command prompt.

Example 4–14. 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 stratixiv_hssi stratixiv_hssi_components.vhd \
stratixiv_hssi_atoms.vhd
ncvhdl -work work <altgx entity name>.vhd
ncelab work.<my design>
```

Compiling Library Files for Functional Simulation in Verilog HDL

To compile the libraries necessary for functional simulation of a Verilog HDL design targeting a Stratix IV device, type the commands shown in Example 4–15 at the NC-Sim command prompt.

Example 4–15. 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 stratixiv_hssi stratixiv_hssi_atoms.v
ncvlog -work work <altgx module name>.v
ncelab work.<my design>
```

Gate-Level Timing Simulation for Stratix IV GX Devices

Stratix IV gate-level timing simulation is similar to Arria II gate-level timing simulation.

Example 4–16 and Example 4–17 show only the gate-level timing simulation for designs that include transceivers in Stratix IV devices. To simulate transceivers in Arria II, Cyclone IV, and HardCopy IV devices, replace the **stratixiv_hssi** model file with the **arriaii_hssi**, **cycloneiv_hssi**, and **hardcopyiv_hssi** model files, respectively.

To perform a post-fit timing simulation of your design that includes the ALTGX megafunction, edit your **cds.lib** file so that all of the libraries point to the work library and compile **stratixiv_atoms** and **stratixiv_hssi_atoms** into the **stratixiv** and **stratixiv_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 Example 4–13 on page 4–15.

The **stratixiv_hssi_atoms** model file references the **lpm** and **sgate** libraries. You must create these libraries to perform a simulation.

Compiling Library Files for Gate-Level Timing Simulation in VHDL

To compile the libraries necessary for timing simulation of a VHDL design targeting a Stratix IV device, type the commands shown in Example 4–16 at the NC-Sim command prompt.

Example 4–16. 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 stratixiv stratixiv_atoms.vhd \
stratixiv_components.vhd
ncvhdl -work stratixiv_hssi stratixiv_hssi_components.vhd \
stratixiv_hssi_atoms.vhd
ncvhdl -work work <altgx>.vho
ncsdfc <project name>_vhd.sdo
ncelab work.<my design> -TIMESCALE 1ps/1ps \
-SDF_CMD_FILE <SDF Command File> -PULSE_R 0 -PULSE_INT_R 0
```

Compiling Library Files for Gate-Level Timing Simulation in Verilog HDL

To compile the libraries necessary for timing simulation of a Verilog HDL design targeting a Stratix IV device, type the commands shown in Example 4–17 at the NC-Sim command prompt.

Example 4–17. 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 stratixiv stratixiv_atoms.v
ncvlog -work stratixiv_hssi stratixiv_hssi_atoms.v
ncvlog -work work <altgx>.vo
ncelab work.<my design> -TIMESCALE 1ps/1ps -PULSE_R 0 -PULSE_INT_R 0
```

Functional Simulation for Stratix V GX Devices

Functional simulation for Stratix V devices is similar to functional simulation for Arria II, Cyclone IV, HardCopy IV, and Stratix IV devices. To simulate transceivers in Arria II, Cyclone IV, HardCopy IV, and Stratix IV devices, you only have to replace the **stratixv_hssi** model file with the **arriaii_hssi**, **cycloneiv_hssi**, **hardcopyiv_hssi**, and **stratiiv_hssi** model files, respectively.

The **stratixv_hssi_atoms** model file references the **lpm** and **sgate** libraries. You must compile these libraries to perform a simulation.

The transceiver module from the MegaWizard Plug-In Manager is created in **Interfaces/Transceiver PHY**. Select **Custom PHY**.

Compiling Library Files for Functional Simulation

To compile the libraries necessary for functional simulation of a Verilog HDL or VHDL design targeting a Stratix V device, type the following command at the VCS command prompt:

Example 4–18 shows the **cds.lib** file.

Example 4–18. 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 stratixv_hssi ./ncsim_work
DEFINE stratixv ./ncsim_work
DEFINE lpm ./ncsim_work
DEFINE sgate ./ncsim_work
```

The **stratixv_hssi_atoms** model file references the **lpm** and **sgate** libraries. You must create these libraries to perform a simulation.

Compiling Library Files for Functional Simulation in VHDL

To compile the libraries necessary for functional simulation of a VHDL design targeting a Stratix V device, type the commands shown in Example 4–19 at the NC-Sim command prompt.

Example 4–19. 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 stratixv_hssi stratixv_hssi_components.vhd \
ncvlog +v2k –work stratixv_hssi \
quartus/eda/sim_lib/cadence/stratixv_hssi_atoms_ncrypt.v
stratixv_hssi_atoms.vhd
ncvhdl -work work <my_design>.vhd
ncelab work.<my design>
```

Compiling Library Files for Functional Simulation in Verilog HDL

To compile the libraries necessary for functional simulation of a Verilog HDL design targeting a Stratix V device, type the commands shown in Example 4–20 at the NC-Sim command prompt.

Example 4–20. 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 +v2k –work stratixv_hssi \
quartus/eda/sim_lib/cadence/stratixv_hssi_atoms_ncrypt.v
ncvlog -work stratixv_hssi stratixv_hssi_atoms.v
ncvlog -work work <my design>.v
ncelab work.<my design>
```

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 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 shows 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.

For more information, refer to the "Using the NativeLink Feature" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook.*

Generating a Timing VCD File for the PowerPlay Power Analyzer

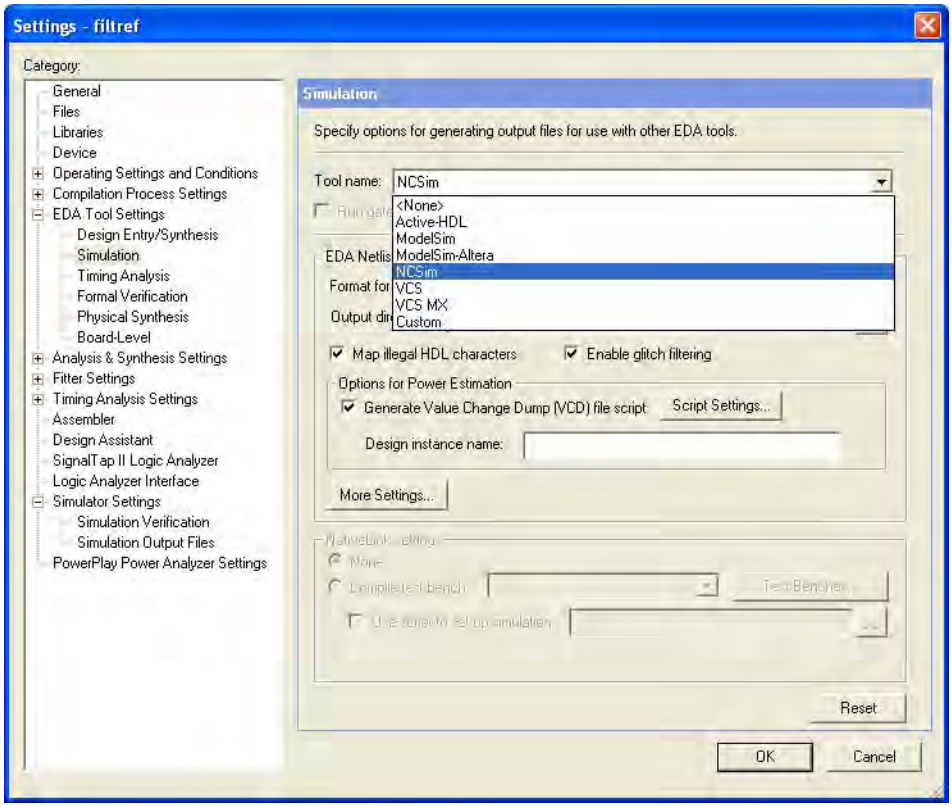
To generate a timing **.vcd** file for PowerPlay, you must first generate a VCD script in the Quartus II software and run the VCD script from the NC-Sim software to generate a timing **.vcd** file. This timing **.vcd** file can then be used by the PowerPlay Power Analyzer for power analysis. The following 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 4–1).
2. In the **Category** list, click the "+" icon to expand **EDA Tool Settings**.
3. Click **Simulation**.
4. In the **Tool name** list, click **NC-Sim**.

5. Turn on the **Generate Value Change Dump (VCD) File Script** option.

Figure 4–1. 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, 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 it finishes. Exit **ncsim** and the <*revision_name*>**.vcd** is generated in the simulation directory.

For more detailed information about using the timing **.vcd** file for power analysis, 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 ↵
```

For more information, refer to *About Quartus II Scripting* in Quartus II Help.

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*.

Generating 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 4–7.

Tcl Commands

The following three assignments cause a Verilog HDL netlist to be written out when you run the Quartus II netlist writer.

```
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)"
```

The netlist has a 1 ps timing resolution for the NC-Sim simulation software.

Type the following Tcl command to run the Quartus II Netlist Writer:

```
execute_module -tool eda
```

Command Prompt

Type 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 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.

Document Revision History

Table 4–2 shows the revision history for this chapter.

Table 4–2. Document Revision History (Part 1 of 2)

Date	Version	Changes Made
July 2010	10.0.0	■ Linked to Help where appropriate ■ Minor text edits ■ Removed Referenced Documents section
November 2009	9.1.0	■ Removed NativeLink information and referenced new *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook* ■ Added "RTL Functional Simulation for Stratix IV Devices" and "Gate-Level Timing Simulation for Stratix IV Devices" sections ■ Minor text edits
March 2009	9.0.0	■ 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

Table 4–2. Document Revision History (Part 2 of 2)

Date	Version	Changes Made
November 2008	8.1.0	■ 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.
May 2008	8.0.0.0	■ 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.

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

Take an online survey to provide feedback about this handbook chapter.

5. Aldec Active-HDL and Riviera-PRO Support

QII53023-10.0.0

This chapter describes how to use the Active-HDL and Riviera-PRO software to simulate designs that target Altera® FPGAs. This chapter provides step-by-step instructions about how to perform functional simulations, post-synthesis simulations, and gate-level timing simulations for Verilog HDL or VHDL designs.

This chapter includes the following topics:

- "Software Requirements"
- "Using Active-HDL or Riviera-PRO Software in Quartus II Design Flows"
- "Simulation Libraries" on page 5–2
- "Performing Simulation Using the Active-HDL Software (GUI Mode)" on page 5–3
- "Performing Simulation Using the Riviera-PRO Software (GUI Mode)" on page 5–5
- "Performing Simulation Using the Active-HDL or Riviera-PRO Software (Command-Line Mode)" on page 5–6
- "Compiling System Verilog Files" on page 5–10
- "Simulating Designs that Include Transceivers" on page 5–10
- "Using the NativeLink Feature in Active-HDL or Riviera-PRO Software" on page 5–17
- "Generating .vcd Files for the PowerPlay Power Analyzer" on page 5–17
- "Scripting Support" on page 5–18

Software Requirements

To simulate your design using the Active-HDL or Riviera-PRO software, you must first set up the Altera libraries. These libraries are installed with the Quartus II software.

For more information about installing the software and directories created during the Quartus II software installation, refer to the *Altera Software Installation and Licensing* manual.

Using Active-HDL or Riviera-PRO Software in Quartus II Design Flows

You can perform the following types of simulations using the Active-HDL or Riviera-PRO software:

- Functional simulation
- Post-synthesis simulation
- Gate-level timing simulation

Simulation Libraries

Simulation model libraries are required to run a simulation whether you are running a functional simulation, post-synthesis simulation, or gate-level timing simulation. In general, running a functional simulation requires the 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 functional simulation. For example, some of Altera megafunctions require gate-level libraries to perform a functional simulation in third-party simulators.

For each megafunction that you are using, refer to the last page in the Altera MegaWizard™ Plug-In Manager, which lists the simulation library files required to perform a functional simulation for that megafunction.

The transceiver megafunction (for example, ALTGX) also requires the gate-level libraries to perform functional simulation.

For detailed, step-by-step instructions about how to simulate the transceiver megafunction, refer to "Simulating Designs that Include Transceivers" on page 5–10.

Simulation Library Files in the Quartus II Software

In Active-HDL or Riviera-PRO software, you must compile the necessary libraries to perform functional, post-synthesis functional, or gate-level timing simulation. You can refer to these library files for any particular simulation model that you are looking for.

For a list of all functional simulation library files in the Quartus II directory, refer to *Altera Functional Simulation Libraries* in Quartus II Help. For a list of all gate-level timing simulation and post-fit library files in the Quartus II directory, refer to *Altera Post-Fit Libraries* in Quartus II Help.

Disabling 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>
```

Compiling 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 and gate-level timing simulation.

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.

For more information about this tool, refer to the "EDA Simulation Library Compiler" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook.*

Performing 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:

- Functional simulation
- Post-synthesis simulation
- Gate-level timing simulation

Simulation helps you verify your designs and debug them against any 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

When you simulate VHDL designs using the Active-HDL GUI, 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 functional simulation, post-synthesis simulation, and gate-level timing simulation.

Performing Functional Simulation

Functional simulation is typically performed to verify the syntax of the code and to check the functionality of the design.

For detailed information about how to perform functional simulation in the Active-HDL software for VHDL designs, refer to *Performing a Simulation of a VHDL Design with the Active-HDL Software* in Quartus II Help.

If you are targeting a Stratix V device, compile the following files in the **quartus/eda/sim_lib/aldec** directory:

stratixv_atoms_ncrypt.v
stratixv_hssi_atoms_ncrypt.v
stratixv_pcie_hip_atoms_ncrypt.v

These files contain IEEE encrypted Verilog models suitable for VHDL/Verilog co-simulation. You need a co-simulation license from Aldec to use these models in VHDL.

☞ Compile these encrypted Verilog files before you compile any VHDL files.

☞ Compile **stratixv_pcie_hip_atoms_ncrypt.v** with the SystemVerilog option.

Also, compile the following files in the **quartus/eda/sim_lib** directory:

stratixv_atoms.vhd
stratixv_components.vhd
stratixv_hssi_components.vhd
stratixv_pcie_hip_components.vhd
stratixv_hssi_atoms.vhd
stratixv_pcie_hip_atoms.vhd

☞ The PCIe® files are required only if you are using the PCIe HIP.

Simulating Verilog HDL Designs

When you simulate Verilog HDL designs using the Active-HDL GUI, 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 Active-HDL GUI to perform functional simulation, post-synthesis simulation, and gate-level timing simulation.

Performing Functional Simulation

Functional simulation is performed to verify the syntax of the code and to check the functionality of the design.

For detailed information about how to perform functional simulation in the Active-HDL software for Verilog HDL designs, refer to *Performing a Simulation of a Verilog HDL Design with the Active-HDL Software* in Quartus II Help.

If you are targeting a Stratix V device, compile the following files in the **quartus/eda/sim_lib/aldec** directory:

stratixv_atoms_ncrypt.v
stratixv_hssi_atoms_ncrypt.v
stratixv_pcie_hip_atoms_ncrypt.v

These files contain IEEE encrypted Verilog models suitable for VHDL/Verilog co-simulation.

☞ Compile these encrypted Verilog files before you compile any VHDL files.

☞ Compile **stratixv_pcie_hip_atoms_ncrypt.v** with the SystemVerilog option.

Also, compile the following files in the **quartus/eda/sim_lib** directory:

stratixv_atoms.v
stratixv_hssi_atoms.v
stratixv_pcie_hip_atoms.v

Compile **stratixv_pcie_hip_atoms.v** with the SystemVerilog option.

The PCIe® files are required only if you are using the PCIe HIP.

Performing Simulation Using the Riviera-PRO Software (GUI Mode)

You can use the Riviera-PRO software to perform the following types of simulation:

- Functional simulation
- Post-synthesis simulation
- Gate-level timing simulation

For detailed information about how to perform functional simulation with the Riviera-PRO software, refer to *Performing an RTL Functional Simulation with the Riviera-PRO Software* in Quartus II Help. For detailed information about how to perform post-synthesis simulation with the Riviera-PRO software, refer to *Performing a Post-Synthesis Simulation with the Riviera-PRO Software* in Quartus II Help. For detailed information about how to perform gate-level timing simulation with the Riviera-PRO software, refer to *Performing a Gate-Level Simulation with the Riviera-PRO Software* in Quartus II Help.

If you are performing a functional simulation in Verilog HDL and targeting a Stratix V device, compile the following files in the **quartus/eda/sim_lib/aldec** directory:

stratixv_atoms_ncrypt.v
stratixv_hssi_atoms_ncrypt.v
stratixv_pcie_hip_atoms_ncrypt.v

These files contain IEEE encrypted Verilog models suitable for VHDL/Verilog co-simulation.

Compile these encrypted Verilog files before you compile any VHDL files.

Compile **stratixv_pcie_hip_atoms_ncrypt.v** with the SystemVerilog option.

Also, compile the following files in the **quartus/eda/sim_lib** directory:

stratixv_atoms.v
stratixv_hssi_atoms.v
stratixv_pcie_hip_atoms.v

Compile **stratixv_pcie_hip_atoms.v** with the SystemVerilog option.

The PCIe® files are required only if you are using the PCIe HIP.

If you are performing a functional simulation in VHDL and targeting a Stratix V device, compile the following files in the **quartus/eda/sim_lib/aldec** directory:

stratixv_atoms_ncrypt.v
stratixv_hssi_atoms_ncrypt.v
stratixv_pcie_hip_atoms_ncrypt.v

These files contain IEEE encrypted Verilog models suitable for VHDL/Verilog co-simulation. You need a co-simulation license from Aldec to use these models in VHDL.

☞ Compile these encrypted Verilog files before you compile any VHDL files.

☞ Compile **stratixv_pcie_hip_atoms_ncrypt.v** with the SystemVerilog option.

Also, compile the following files in the **quartus/eda/sim_lib** directory:

stratixv_atoms.vhd
stratixv_components.vhd
stratixv_hssi_components.vhd
stratixv_pcie_hip_components.vhd
stratixv_hssi_atoms.vhd
stratixv_pcie_hip_atoms.vhd

☞ The PCIe® files are required only if you are using the PCIe HIP.

Performing Simulation Using the Active-HDL or Riviera-PRO Software (Command-Line 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:

- 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.

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 or Riviera-PRO from the command line, ensure that the **Active-HDL/bin** or **Riviera-PRO/bin** directory is located in PATH environment variables.

Simulating Verilog HDL Designs Using the Active-HDL or Riviera-PRO Software

If you are running the simulation commands from the Active-HDL GUI console, you must first create the workspace. Type the following commands to create and open the workspace:

```
createdesign DELAY_TEST C:/DELAY_TEST/simulation/activehdl ↵
opendesign -a DELAY_TEST.adf ↵
```

If you are running Riviera-PRO, you can skip the step above.

Performing Functional Simulation

Use the following commands to perform a functional simulation for Verilog HDL designs with one of the libraries (lib1) listed in *Altera Functional Simulation Libraries* in Quartus II Help:

```
# Create and compile Altera libraries
vlib <lib1> ↵
vlog -v2k -dbg -work <lib1> <lib1.v> ↵

vlib <lib2> ↵
vlog -v2k -dbg -work <lib2> <lib2.v> ↵

# Create work library and compile design files and testbench file
vlib work ↵
vlog -v2k -dbg -work work <design_file1.v> <design file2.v> <testbench \
file.v> ↵

# Load Design
vsim +access +r -t 1ps -L work -L <lib1> -L <lib2> work.<testbench module \
name> ↵

#add signals at waveform and run
add wave * ↵
run ↵
```

Example

```
vlib verilog_libs/lpm_ver
vlog -v2k -dbg -work lpm_ver c:/altera/91/quartus/eda/sim_lib/220model.v

vlib verilog_libs/altera_mf_ver
vlog -v2k -dbg -work altera_mf_ver
c:/altera/91/quartus/eda/sim_lib/altera_mf.v

vlib work
vlog -v2k -dbg -work work C:/project/adder.v C:/project/adder.vt

vsim +access +r -t 1ps -L work -L lpm_ver -L altera_mf_ver
work.adder_vlg_vec_tst

add wave *
```

Performing Post-Synthesis Simulation

Before you run post-synthesis simulation, generate post-synthesis simulation netlist files. Refer to the "Generating Post-Synthesis Simulation Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

You cannot perform post-synthesis or post-fit simulation if you are targeting the Stratix V device family.

Use the following commands to perform a post-synthesis simulation for Verilog HDL designs with one of the libraries (lib1) listed in *Altera Post-Fit Libraries* in Quartus II Help:

```
# Create and compile Altera libraries
vlib <lib1> ↵
vlog -v2k -dbg -work <lib1> <lib1.v> ↵

vlib <lib2> ↵
vlog -v2k -dbg -work <lib2> <lib2.v> ↵

# Create work library and compile EDA output netlist files and testbench file
vlib work ↵
vlog -v2k -dbg -work work <EDA_output_netlist.vo> <testbench file.v> ↵
```

```
# Load Design
vsim +access +r -t 1ps +transport_int_delays +transport_path_delays -L work \
-L <lib1> -L <lib2> work.<testbench module name> ↵

#add signals at waveform and run
add wave * ↵
run ↵
```

Example

```
vlib verilog_libs/lpm_ver
vlog -v2k -dbg -work lpm_ver c:/altera/91/quartus/eda/sim_lib/220model.v

vlib verilog_libs/altera_ver
vlog -v2k -dbg -work altera_ver
c:/altera/91/quartus/eda/sim_lib/altera_primitives.v

vlib verilog_libs/stratixiv_ver

vlog -v2k -dbg -work stratixiv_ver
c:/altera/91/quartus/eda/sim_lib/stratixiv_atoms.v

vlib work

vlog -v2k -dbg -work work C:/project/simulation/activehdl/adder.vo
C:/project/adder.vt

vsim +access +r -t 1ps +transport_int_delays +transport_path_delays -L work
-L lpm_ver -L altera_mf_ver work.adder_vlg_vec_tst

add wave *
run
```

Performing Gate-Level Timing Simulation

The steps for gate-level timing simulation are almost same as the steps for post-synthesis simulation. The only difference is that the SDO file must be back-annotated for gate level-timing simulation.

For Verilog HDL designs, the back-annotating process is done within the **EDA_output_netlist.vo** script. Therefore, you are not required to back-annotate the SDO file again.

You cannot perform post-synthesis or post-fit simulation if you are targeting the Stratix V device family.

Simulating VHDL Designs Using the Active-HDL or Riviera-PRO Software

If you are running the simulation commands from the Active-HDL GUI console, you first create the workspace. The following commands create and open the workspace:

```
createdesign <workspace name> <your design path> ↵
opendesign -a <workspace name>.adf ↵
```

If you are running Riviera-PRO, you can skip the step above.

Performing Functional Simulation

Use the following commands to perform a functional simulation for VHDL designs with one of the libraries (lib1) listed in *Altera Functional Simulation Libraries* in Quartus II Help:

```
# Create and compile Altera libraries
vlib <lib1> ↵
vcom -strict93 -dbg -work <lib1> <lib1_component/pack.vhd> <lib1.vhd> ↵

vlib <lib1> ↵
vcom -strict93 -dbg -work <lib2> <lib2_component/pack.vhd> <lib2.vhd> ↵
```

```
# Create work library and compile design files and testbench file
vlib work ↵
vcom  - strict93  -dbg -work work <design_file1.vhd> <design file2.vhd> \
<testbench file.vhd> ↵

# Load Design
vsim +access +r -t 1ps -L work -L <lib1> -L <lib2> work.<testbench module \
name> ↵

#add signals at waveform and run
add wave * ↵
run ↵
```

Example

```
vlib vhdl_libs/lpm
vcom -strict93 -dbg -work lpm c:/altera/91/quartus/eda/sim_lib/220pack.vhd
vcom -strict93 -dbg -work lpm c:/altera/91/quartus/eda/sim_lib/220model.vhd

vlib vhdl_libs/altera_mf
vcom -strict93 -dbg -work altera_mf
c:/altera/91/quartus/eda/sim_lib/altera_mf_components.vhd
vcom -strict93 -dbg -work altera_mf
c:/altera/91/quartus/eda/sim_lib/altera_mf.vhd

vlib work
vcom -strict93 -dbg -work work C:/project/adder.vhd C:/project/adder.vht

vsim +access +r -t 1ps -L adder -L work -L lpm -L altera_mf
work.adder_vhd_vec_tst

#add signals at waveform and run
add wave *
run
```

Performing Post-Synthesis Simulation

Before you run post-synthesis simulation, generate post-synthesis simulation netlist files. Refer to the "Generating Post-Synthesis Simulation Netlist Files" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

You cannot perform post-synthesis or post-fit simulation if you are targeting the Stratix V device family.

Use the following commands to perform a functional simulation for VHDL designs with one of the libraries (lib1) listed in *Altera Post-Fit Libraries* in Quartus II Help:

```
# Create and compile Altera libraries
vlib <lib1> ↵
vcom -strict93 -dbg -work <lib1> <lib1_component/pack.vhd> <lib1.vhd> ↵

vlib <lib1> ↵
vcom -strict93 -dbg -work <lib2> <lib2_component/pack.vhd> <lib2.vhd> ↵

# Create work library and compile EDA output netlist files and testbench file
vlib work ↵
vcom  - strict93  -dbg -work work <EDA output netlist.vho> <testbench \
file.vhd> ↵

# Load Design
vsim +access+r -t 1ps +transport_int_delays +transport_path_delays -L work
\ -L <lib1> -L <lib2> work.<testbench module name> ↵

#add signals at waveform and run
add wave * ↵
run ↵
```

Example

```
vlib vhdl_libs/lpm
vcom -strict93 -dbg -work lpm c:/altera/91/quartus/eda/sim_lib/220pack.vhd
vcom -strict93 -dbg -work lpm c:/altera/91/quartus/eda/sim_lib/220model.vhd

vlib vhdl_libs/altera
vcom -strict93 -dbg -work altera
c:/altera/91/quartus/eda/sim_lib/altera_primitives_components.vhd
vcom -strict93 -dbg -work altera
c:/altera/91/quartus/eda/sim_lib/altera_primitives.vhd

vlib work
vcom -strict93 -dbg -work work C:/project/simulation/activehdl/adder.vho
C:/project/adder.vht

vsim +access +r -t 1ps +transport_int_delays +transport_path_delays -L adder
-L work -L lpm -L altera_mf  work.adder_vhd_vec_tst

add wave *
run
```

Performing Gate-Level Timing Simulation

The steps for gate-level timing simulation are almost same as the steps for post-synthesis simulation. The only difference is that the SDO file must be back-annotated for gate level-timing simulation.

You cannot perform post-synthesis or post-fit simulation if you are targeting the Stratix V device family.

For VHDL designs, the back-annotating process is done by adding the `-sdftyp` option.

Example

```
vsim +access +r -t 1ps +transport_int_delays +transport_path_delays
-sdftyp <instance path to design>= <path to SDO file> -L adder -L work
-L lpm -L altera_mf  work.adder_vhd_vec_tst
```

Compiling System Verilog Files

If your design includes multiple System Verilog files, you must compile the System Verilog files together with a single **alog** command.

If you have Verilog files and System Verilog files in your design, it is recommended that you compile the Verilog files, and then compile only the System Verilog files in the single **alog** command.

Simulating Designs that Include Transceivers

If your design includes Arria®, Arria II, Cyclone IV, HardCopy IV, Stratix, Stratix II, or Stratix IV transceivers, you must compile additional library files to perform 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 (**Design** tab of the **Start Simulation** dialog box). 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, respectively.

If your design contains PCI Express® hard IP, refer to the "Simulate the Design" section in the *PCI Express Compiler User Guide.*

Functional Simulation for Stratix II GX Devices

Functional simulation for Stratix II GX devices is similar to functional simulation for Arria GX devices. 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, replace the **stratixiigx_hssi** model file with the **arriagx_hssi** model file.

To perform an 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 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.

The <*alt2gxb entity name*>**.vho** or <*alt2gxb module name*>**.vo** is generated in the current project directory.

The ALT2GXB functional simulation library file generated by the Quartus II software references **stratixiigx_hssi** WYSIWYG atoms.

Performing Functional Simulation in VHDL

Type the commands in Example 5–1 to compile and simulate the design.

Example 5–1.

```
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 work <my design>.vhd <my testbench>.vhd
vsim -L lpm -L altera_mf -L sgate -L stratixgx_hssi work.<my testbench>
```

Performing Functional Simulation in Verilog HDL

Type the commands in Example 5–2 to compile and simulate the design.

Example 5–2.

```
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 work <my design>.v <my testbench>.v
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixgx_hssi \
   work.<my testbench>
```

Gate-Level Timing Simulation for Stratix II GX Devices

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 Gate-Level Timing Simulation in VHDL

Type the commands in Example 5–3 to compile and simulate the design.

Example 5–3.

```
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 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 Gate-Level Timing Simulation in Verilog HDL

Type the commands in Example 5–4 to compile and simulate the design.

Example 5–4.

```
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 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
```

Functional Simulation for Stratix GX Devices

To perform a 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. You must create these libraries to perform a simulation.

Performing Functional Simulation in VHDL

Type the commands in Example 5–5 to compile and simulate the design.

Example 5–5.

```
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 work<altgxb entity name>.vhd
vsim -L lpm -L altera_mf -L sgate -L altgxb work.<my testbench>
```

Performing Functional Simulation in Verilog HDL

Type the commands in Example 5–6 to compile and simulate the design.

Example 5–6.

```
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 work <altgxb module name>.v
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L altgxb_ver work.<my testbench>
```

Gate-Level Timing Simulation for Stratix GX Devices

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 Gate-Level Timing Simulation in VHDL

Type the commands in Example 5–7 to compile and simulate the design.

Example 5–7.

```
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 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 in Verilog HDL

Type the commands in Example 5–8 to compile and simulate the design.

Example 5–8.

```
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 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
```

Functional Simulation for Stratix IV GX Devices

Functional simulation for Stratix IV devices is similar to functional simulation for Arria II, Cyclone IV, and HardCopy IV devices.

The following example shows only the functional simulation for designs that include transceivers in Stratix IV devices. To simulate transceivers in Arria II, Cyclone IV, and HardCopy IV devices, replace the **stratixiv_hssi** model file with the **arriaii_hssi**, **cycloneiv_hssi**, and **hardcopyiv_hssi** model files, respectively.

To perform a functional simulation of your design that instantiates the ALTGX megafunction, which enables the gigabit transceiver blocks on Stratix IV devices, you must generate a functional simulation netlist and compile the **stratixiv_hssi** model file into the **stratixiv_hssi** library.

The **stratixiv_hssi** model file references the **lpm** and **sgate** libraries. You must create these libraries to perform a simulation.

Performing Functional Simulation in VHDL

Type the commands in Example 5–9 to compile and simulate the design.

Example 5–9.

```
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 stratixiv_hssi stratixiv_hssi.vhd\
stratixiv_hssi_components.vhd
vcom -work work <altgxb entity name>.vhd
vsim -L lpm -L altera_mf -L sgate -L stratixiv_hssi work.<my testbench>
```

Performing Functional Simulation in Verilog HDL

Type the commands in Example 5–10 to compile and simulate the design.

Example 5–10.

```
vlib work
vlib lpm_ver
vlib altera_mf_ver
vlib sgate_ver
vlib stratixiv_hssi_ver
vlog -work lpm_ver 220model.v
vlog -work altera_mf_ver altera_mf.v
vlog -work sgate_ver sgate.v
vlog -work stratixiv_hssi_ver stratixiv_hssi_.v
vlog -work work <altgxb module name>.v
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver \
-L stratixiv_hssi_ver work.<my testbench>
```

Gate-Level Timing Simulation for Stratix IV GX Devices

Perform a gate-level timing simulation of your design that includes a Stratix IV transceiver by compiling the **stratixiv_atoms** and **stratixiv_hssi_atoms** model files into the **stratixiv** and **stratixiv_hssi** libraries, respectively.

Performing Gate-Level Timing Simulation in VHDL

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 stratixiv stratixiv_atoms.vhd stratixiv_components.vhd
vcom -work stratixiv_hssi stratixiv_hssi_atoms.vhd \
stratixiv_hssi_components.vhd
vcom -work work <my design>.vho <my testbench>.vhd
vsim -L lpm -L altera_mf -L sgate -L stratixiv -L stratixiv_hssi\
work.<my testbench> -t ps\
-sdftyp <design instance>=<path to SDO file>.sdo \
+transport_int_delays +transport_path_delays
```

Performing Gate-Level Timing Simulation in Verilog HDL

Type the commands in Example 5–12 to compile and simulate the design.

Example 5–12.

```
vlog -work lpm_ver 220model.v
vlog -work altera_mf_ver altera_mf.v
vlog -work sgate_ver sgate.v
vlog -work stratixiv_ver stratixiv_atoms.v
vlog -work stratixiv_hssi_ver stratixiv_hssi_atoms.v
vlog -work work <my design>.vo <my testbench>.v
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver -L stratixiv_ver -L
stratixiv_hssi_ver \
work.<my testbench> -t ps +transport_int_delays +transport_path_delays
```

Functional Simulation for Stratix V GX Devices

Functional simulation for Stratix V devices is similar to functional simulation for Arria II, Cyclone IV, HardCopy IV, and Stratix IVdevices.

The following example shows only the functional simulation for designs that include transceivers in Stratix V devices. To simulate transceivers in Arria II, Cyclone IV, HardCopy IV, and Stratix V devices, replace the **stratixv_hssi** model file with the **arriaii_hssi**, **cycloneiv_hssi**, **hardcopyiv_hssi**, and **stratixiv_hssi** model files, respectively.

The transceiver module from the MegaWizard Plug-In Manager is created in **Interfaces/Transceiver PHY**. Select **Custom PHY**.

Performing Functional Simulation in VHDL

Type the commands in Example 5–13 to compile and simulate the design.

Example 5–13.

```
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
vlog +v2k –work stratixv_hssi \
quartus/eda/sim_lib/aldec/stratixv_hssi_atoms_ncrypt.v
vcom -work stratixv_hssi stratixiv_hssi.vhd \
stratixiv_hssi_components.vhd
vcom -work work <my design>.vhd
vsim -L lpm -L altera_mf -L sgate -L stratixv_hssi work.<my testbench>
```

Performing Functional Simulation in Verilog HDL

Type the commands in Example 5–14 to compile and simulate the design.

Example 5–14.

```
vlib work
vlib lpm_ver
vlib altera_mf_ver
vlib sgate_ver
vlib stratixv_hssi_ver
vlog -work lpm_ver 220model.v
vlog -work altera_mf_ver altera_mf.v
vlog -work sgate_ver sgate.v
vlog +v2k –work stratixv_hssi \
quartus/eda/sim_lib/aldec/stratixv_hssi_atoms_ncrypt.v
vlog -work stratixv_hssi_ver stratixiv_hssi_.v
vlog -work work <my design>.v
vsim -L lpm_ver -L altera_mf_ver -L sgate_ver \
-L stratixv_hssi_ver work.<my testbench>
```

The **stratixv_hssi** model file references the **lpm** and **sgate** libraries. You must create these libraries to perform a simulation.

In addition to the top-level variant wrapper, <*variant*>**.v**, you also get a simulation files subdirectory, <*variant*>**_sim/**. All Verilog (**.v**) and SystemVerilog (**.sv**) files in the simulation directory must also be compiled into the simulation project.

Transport Delays

By default, the Active-HDL or Riviera-PRO software filters out all pulses that are shorter than the propagation delay between primitives. Turning on the **transport delay** options in the Active-HDL or Riviera-PRO 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 5–15 shows an Active-HDL software command in command-line syntax to perform a gate-level timing simulation with the device family library.

Example 5–15.

```
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 or Riviera-PRO 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 the Active-HDL or Riviera-PRO software within the Quartus II software.

For more information, refer to the "Using the NativeLink Feature" section in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

Generating .vcd Files for the PowerPlay Power Analyzer

To generate a Value Change Dump File (**.vcd**) for the PowerPlay power analyzer, 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 analysis. 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, select **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 analysis, 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 or Riviera-PRO

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 or Riviera-PRO (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 or Riviera-PRO 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 ↵
```

or

```
set_global_assignment-name EDA_SIMULATION_TOOL "Riviera-PRO (Verilog)" ↵
set_global_assignment-name EDA_GENERATE_FUNCTIONAL_NETLIST ON ↵
```

Command Line

Use the following command to generate a simulation output file for the Active-HDL or Riviera-PRO software. Specify VHDL or Verilog HDL for the format:

```
quartus_eda <project name> --simulation=on --format=<format> \
    --tool=activehdl --functional ↵
```

or

```
quartus_eda <project name> --simulation=on --format=<format> \
    --tool=rivierapro --functional ↵
```

Generating a Gate-Level Timing Simulation Netlist for Active-HDL or Riviera-PRO

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)" ↵
```

or

```
set_global_assignment -name EDA_SIMULATION_TOOL "Riviera-PRO (Verilog)" ↵
set_global_assignment -name EDA_SIMULATION_TOOL "Riviera-PRO (VHDL)" ↵
```

Command Line

Generate a simulation output file for the Active-HDL or Riviera-PRO 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 ↵
```

or

```
quartus_eda <project name> --simulation=on --format=<format> \
    --tool=rivierapro ↵
```

Conclusion

Using the Active-HDL or Riviera-PRO simulation software within the Altera FPGA design flow allows you to easily and accurately perform 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 or Riviera-PRO software helps ensure your design functions correctly and, ultimately, a quick time-to-market.

Document Revision History

Table 5–1 shows the revision history for this chapter.

Table 5–1. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Linked to Quartus II Help ■ Revised simulation procedures ■ Added Stratix V simulation information ■ Added Riviera-PRO support ■ Minor text edits ■ Removed Referenced Documents section
November 2009	9.1.0	■ Updated Table 6–1 ■ Removed Simulation Library tables and EDA Simulation Library Compiler sections and referenced new *Simulating Designs with EDA Tools* chapter ■ Added "RTL Functional Simulation for Stratix IV Devices" and "Gate-Level Timing Simulation for Stratix IV Devices" sections ■ Minor text edits
March 2009	9.0.0	■ 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
November 2008	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
May 2008	8.0.0	Initial release

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

Take an online survey to provide feedback about this handbook chapter.

6. Simulating Altera IP in Third-Party Simulation Tools

QII53014-10.0.1

This chapter describes the process for instantiating the IP megafunctions in your design and simulating their functional simulation models in Altera-supported, third-party simulation tools.

The capacity and complexity of Altera® FPGAs continues to increase as the need for intellectual property (IP) becomes increasingly critical. Using IP megafunctions reduces overall 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 IP blocks 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.

In this chapter, IP megafunctions refer to Altera megafunctions, IP MegaCore® functions, and IP AMPP megafunctions.

Altera IP megafunctions support both Verilog and VHDL simulation. The manner in which dual-language simulation is supported for specific megafunctions might differ.

Simulation models may be provided as:

- Plain text VHDL
- Plain text Verilog
- IP Functioal Simulation (IPFS) models in both VHDL and Verilog
- IEEE 1364-2005 encrypted Verilog

When IEEE 1364-2005 encrypted Verilog models are provided, there is a separate model for each Altera-supported simulator. You need a simulator that is capable of VHDL/Verilog co-simulation if you want to simulate the model in a VHDL design. Some AMPP megafunctions might also use 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.

CAUTION

Use IPFS models for simulation only. Do not use them for synthesis or any other purpose. Using these models for synthesis results in a nonfunctional design.

This chapter discusses the following topics:

- "IP Functional Simulation Flow"
- "Instantiate the IP in Your Design" on page 6–4
- "Perform Functional Simulation" on page 6–4
- "Design Language Examples" on page 6–6

IP Functional Simulation Flow

The IP megafunction's MegaWizard™ interface allows you to quickly and easily view documentation, specify parameters, generate a simulation model, and output the files necessary to integrate a parameterized IP megafunction into your design. Within the Quartus II software, you can use the MegaWizard Plug-In Manager 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 use IPFS models, the Quartus II software can also generate a Verilog HDL Output File (**.vo**) or VHDL Output File (**.vho**) that contains an IPFS model after you have parameterized the megafunction. IPFS models are written to the Quartus II project directory.

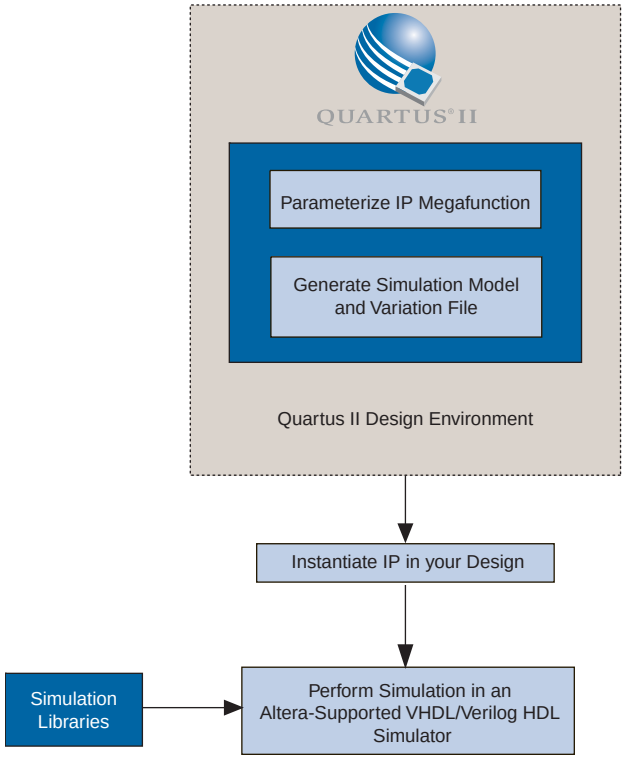
Most Altera megafunctions and IP MegaCore functions support functional simulation in Verilog HDL and VHDL for all Altera-supported third-party simulators. Simulation libraries are required to simulate IP megafunctions.

For a list of the RTL functional simulation library files in the Quartus II project directory, refer to *Altera Functional Simulation Libraries* in Quartus II Help.

For a list of the gate-level timing simulation library files in the Quartus II project directory, refer to *Altera Post-Fit Libraries* in Quartus II Help.

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 use IPFS models for functional simulation. The IPFS 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.

Simulator-independent IPFS primitives are located in the **quartus/eda/sim_lib** directory.

For Stratix V transceiver devices, IEEE 1364-2005 encrypted simulator-dependent RTL models are located in the following directories:

quartus/eda/sim_lib/mentor
quartus/eda/sim_lib/synopsys
quartus/eda/sim_lib/aldec
quartus/eda/sim_lib/cadence

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.

If you want to simulate an IEEE 1364-2005 encrypted model in a VHDL design, you will need a simulator that supports VHDL/Verilog co-simulation.

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.

For information about instantiating IP megafunctions, refer to the *Megafunction Overview User Guide*.

For 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 Functional Simulation

To perform functional simulation, in addition to adding your design files and testbench files, you also have to add the IP megafunction's variation file and other appropriate IP simulation files and model libraries 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.

The IP that uses the transceiver requires the Altera transceiver libraries for simulation. Refer to *Altera Functional Simulation Libraries* in Quartus II Help.

The PCI Express (PCIe) hard IP core in a Stratix IV device requires the PCIe libraries for simulation. Refer to *Altera Functional Simulation Libraries* in Quartus II Help.

The Quartus II software contains all the libraries required for setting up and running a successful simulation of Altera IP. If the IP megafunction you are using supports the Quartus II NativeLink feature, it is easy to use the NativeLink feature to set up your simulation. However, you can also simulate Altera IP directly with third-party simulators. Refer to the applicable IP megafunction user guide to determine whether the NativeLink feature is supported.

Simulating Altera IP Using the Quartus II NativeLink Feature

The Quartus II NativeLink feature eases the tasks 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.

Before running the simulation, you must open your Quartus II project and perform Analysis and Elaboration, as described in the following section.

Perform Analysis and Elaboration on Your Design

To perform Analysis and Elaboration on your design, on the Quartus II Processing menu, point to **Start**, then click **Start Analysis & Elaboration**.

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 if they are present in the Quartus II project directory.

You can now continue the simulation process using the NativeLink feature.

Run Simulation Using the Quartus II NativeLink Feature

For detailed information about using the NativeLink feature to simulate Altera IP, refer to the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

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. Instead, 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 the transceiver block, you must compile these libraries, which are device dependent.
- **PCIe**—includes the definition of PCI Express hard IP megafunctions for Arria II, Cyclone IV, HardCopy IV, Stratix IV, and Stratix V devices. If you use IP with the PCI Express hard IP for Stratix IV devices, 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, and no compilation is required.

Using the EDA Simulation Library Compiler

You can also compile these libraries with the EDA Simulation Library Compiler. The benefit of using this tool is that all of the required libraries are automatically compiled for your simulation use.

For more information about the EDA Simulation Library Compiler, refer to the section "EDA Simulation Library Compiler" in the *Simulating Designs with EDA Tools* chapter in volume 3 of the *Quartus II Handbook*.

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 SE software.

Verilog HDL Example: Simulating the IPFS Model in the ModelSim SE 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. To create a ModelSim project, perform 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.)

      If you are using the EDA Simulation Library Compiler, skip to step 3.

      - 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 specific.
      - The PCIe library files in Verilog HDL, if you use IP with the PCIe hard core for Arria II, Cyclone IV, HardCopy IV, Stratix IV, and Stratix V devices.

For a list of the RTL functional simulation library files in the Quartus II directory, refer to *Altera Functional Simulation Libraries* in Quartus II Help.

2. To create the required simulation libraries, type the following commands at the ModelSim prompt:

```
vlib sgate ↵
vlib lpm ↵
vlib altera_mf ↵
vlib <family>_hssi ↵      (If you use transceivers)
vlib <family>_pcie_hip ↵      (If you use PCI Express hard IP)
```

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 ↵
vmap <family>_hssi <family>_hssi ↵      (If you use transceivers)
vmap <family>_pcie_hip <family>_pcie_hip ↵      (If you use PCI Express hard IP)
```

☞ If you are using the EDA Simulation Library Compiler, skip to step 5.

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 ↵
vlog -work <family>_hssi <family>_hssi_atoms.v ↵      (If you use transceivers)
vlog -work <family>_pcie_hip <family>_pcie_hip_atoms.v ↵      (If you use PCI Express hard IP)
vlog -work <family>_hssi <Quartus_installation_directory> \
/eda/sim_lib/mentor/stratixv_hssi_atoms.v ↵      (For Stratix V devices)
vlog -work <family>_pcie_hip <Quartus_installation_directory> \
/eda/sim_lib/mentor/stratixv_pcie_hip_atoms.v ↵      (For Stratix V devices)
```

5. Compile the IPFS model by typing the following command at the ModelSim prompt:

```
vlog -work work <my_IP>.vo ↵
```

6. Compile the 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 ↵
```

8. Load the testbench by typing the following command at the ModelSim prompt:

```
vsim -L altera_mf -L lpm -L sgate -L <family>_pcie_hip \
work.<my_testbench> ↵
```

VHDL Example: Simulating the IPFS Model in the ModelSim SE Software

The following example shows how to perform a functional simulation of a VHDL-based, megafunction IPFS model. The example assumes that the megafunction variation and the IPFS model are generated.

If you want to simulate an IEEE 1364-2005 encrypted model in a VHDL design, you need a simulator that supports VHDL/Verilog co-simulation.

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:
      - 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.vhd**, **sgate_pack.vhd**, 2**20model.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 specific.
      - The PCIe library files in VHDL, if you use IP with the PCI Express hard IP for Stratix V, Stratix IV, HardCopy IV, Arria II, and Cyclone IV devices.

For a list of the RTL functional simulation library files in the Quartus II directory, refer to *Altera Functional Simulation Libraries* in Quartus II Help.

2. Create the required simulation libraries by typing the following commands at the ModelSim prompt:

```
vlib sgate ↵
vlib lpm ↵
vlib altera_mf ↵
vlib <family>_hssi ↵      (If you use transceivers)
vlib <family>_pcie_hip ↵      (If you use PCI Express hard IP)
```

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 ↵
vmap <family>_hssi <family>_hssi ↵      (If you use transceivers)
vmap <family>_pcie_hip <family>_pcie_hip ↵      (If you use PCI Express hard IP)
```

4. Compile the HDL into libraries by typing the following commands at the ModelSim prompt:

```
vlog -work <family>_hssi <Quartus_installation_directory> \
/eda/sim_lib/mentor/<family>_hssi_atoms.v ↵
```
(For Stratix V devices)

```
vlog -work <family>_pcie_hip <Quartus_installation_directory> \
/eda/sim_lib/mentor/<family>_pcie_hip_atoms.v ↵
```
(For Stratix V devices)

```
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 ↵
vcom -work <family>_hssi -93 -explicit <family>_hssi_atoms.vhd ↵
```
(If you use transceivers)

```
vcom -work <family>_hssi -93 -explicit <family>_hssi_components.vhd ↵
vcom -work <family>_pcie_hip -93 <family>_pcie_hip_atoms.vhd ↵
```
(If you use PCI Express hard IP)

```
vcom -work <family>_pcie_hip -93 <family>_pcie_hip_components.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 ↵
```

Conclusion

The Quartus II software enables you to simulate 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.

Document Revision History

Table 6–1 shows the revision history for this chapter.

Table 6–1. Document Revision History

Date	Version	Changes Made
August 2010	10.0.1	Corrected links
July 2010	10.0.0	■ Linked Altera Functional Simulation Libraries and Altera Post-Fit Libraries to Quartus II Help ■ Updated to add directories and information about simulator-dependent IPFS model simulation files ■ Removed Referenced Documents section
November 2009	9.1.0	■ Removed most NativeLink information ■ Added references to new chapter *Simulating Designs with EDA Tools* and *Megafunction Overview User Guide* ■ Minor text edits
March 2009	9.0.0	Removed figures.
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	■ Updated “Introduction.” ■ 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.

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

Take an online survey to provide feedback about this handbook chapter.

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 use PrimeTime to analyze your Quartus II projects.

This section includes the following chapters:

- Chapter 7, The Quartus II TimeQuest Timing Analyzer

  This chapter describes the Quartus II TimeQuest Timing Analyzer, which 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.

- Chapter 8, Best Practices for the Quartus II TimeQuest Timing Analyzer

  This chapter provides the steps to fully constrain an FPGA design with the Quartus II TimeQuest Timing Analyzer.

- Chapter 9, Switching to the Quartus II TimeQuest 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.

- Chapter 10, Quartus II Classic Timing Analyzer

  This chapter describes the Qurtus II Classic Timing Analyzer. Tcl commands are presented throughout this chapter to describe the alternative methods for making timing analysis assignments.

- Chapter 11, Synopsys PrimeTime Support

  This chapter describes the PrimeTime software that uses 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.

7. The Quartus II TimeQuest Timing Analyzer

QII53018-10.0.0

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 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.

This chapter contains the following sections:

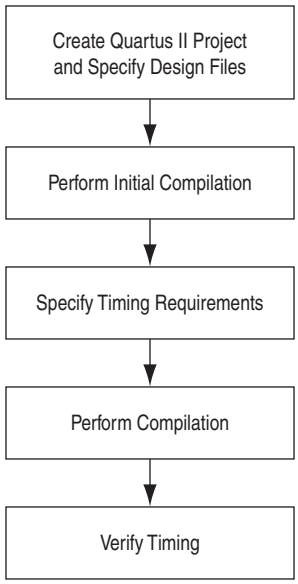
- "Getting Started with the TimeQuest Timing Analyzer" on page 7–2
- "Running the TimeQuest Timing Analyzer" on page 7–3
- "Timing Analysis Overview" on page 7–5
- "The Timing Netlist" on page 7–8
- "Clock Specification" on page 7–23
- "Collections" on page 7–20
- "I/O Specifications" on page 7–35
- "Timing Exceptions" on page 7–39
- "Constraint and Exception Removal" on page 7–43
- "Timing Reports" on page 7–43
- "Other Features" on page 7–57
- "Conclusion" on page 7–63

Getting Started with the TimeQuest Timing Analyzer

The 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 TimeQuest timing analyzer, including the necessary steps to properly constrain a design, perform a full place-and-route, and perform reporting on the design. Figure 7–1 shows the recommended design flow steps to maximize and leverage the benefits the TimeQuest timing analyzer. Details about each step are provided after the figure.

Figure 7–1. Design Flow with the TimeQuest Timing Analyzer



- **Create Quartus II Project and Specify 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**—In this step you create 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**—In this step you specify timing requirements to 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**—In this step you compile your design. 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**—In this step, you verify timing in your design with the TimeQuest timing analyzer.

Running the TimeQuest Timing Analyzer

You can run the TimeQuest timing analyzer in one of the following ways:

- Directly from the Quartus II GUI
- As a stand-alone GUI application
- At a system command prompt

This section describes each of the modes, and the behavior of the TimeQuest timing analyzer.

Running the TimeQuest Timing Analyzer from the Quartus II GUI

To run the TimeQuest timing analyzer from the Quartus II software, on the Tools menu, click **TimeQuest Timing Analyzer**. When you launch the TimeQuest timing analyzer from the Quartus II software, the current project is open by default.

The TimeQuest timing analyzer is available after you have created a post-fit or post-map database for the current project. After a database is created, you can create a timing netlist based on that database. You can only create a post-fit timing netlist in the TimeQuest timing analyzer if you have created a post-fit database for your project.

Use the TimeQuest timing analyzer Wizard to enter basic, initial constraints for your design, if none already exist. The wizard allows you to create clock, t_{SU}, t_H, t_{CO}, and t_{PD} constraints and save those constraints in an SDC file. To start the wizard, on the Assignments menu click **TimeQuest Timing Analyzer Wizard**.

For complete information about the TimeQuest timing analyzer GUI, refer to *About the TimeQuest Timing Analyzer* in Quartus II help.

Running the TimeQuest Timing Analyzer as a Stand-Alone Application

To run the TimeQuest timing analyzer as a stand-alone GUI application, 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.

Running the quartus_sta Executable at a System Command Prompt

Use the command-line executable, `quartus_sta`, for easy integration with scripted design flows. Table 7–1 provides a summary of the options available in the command-line mode.

Table 7–1. Summary of Command Line Options (Part 1 of 2)

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 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 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.

Table 7–1. Summary of Command Line Options (Part 2 of 2)

Command Line Option	Description
`--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 TimeQuest timing analyzer in command-line mode, type the following command at the command prompt:

```
quartus_sta <options> ↵
```

The TimeQuest timing analyzer also includes the Tcl packages `::quartus::sta`, `::quartus::sdc`, and `::quartus::sta_ext` for constraining and analyzing designs from Tcl scripts, or for use interactively with the `-s` and `--tcl_eval` options for `quartus_sta`.

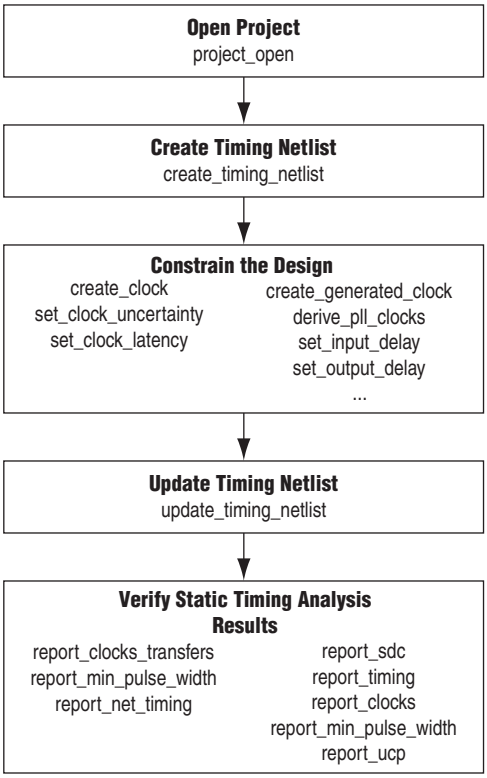
For more information about using Tcl commands to constrain and analyze your design, refer to *API functions for Tcl* in Quartus II Help. For a demonstration of interactive timing analysis, refer to the TimeQuest Quick Start Tutorial.

Timing Analysis Overview

This section provides an overview of TimeQuest timing analyzer concepts. Understanding these concepts allows you to take advantage of the powerful timing analysis features available in the TimeQuest timing analyzer.

The TimeQuest timing analyzer follows the flow shown in Figure 7–2 when it analyzes your design, including the most commonly used Tcl commands for each step.

Figure 7–2. The TimeQuest Timing Analyzer Flow



The following sections describe each of the steps shown in Figure 7–2.

Open the Project

Open the project in the Quartus II software or the stand-alone GUI version. In a fully scripted timing analysis flow, the project must be opened at the start of the process with the `project_open` command.

Create a Timing Netlist

After you perform a full compilation, create either a post-map or post-fit timing netlist based on the fully annotated database from the post-fit results.

To create the timing netlist with Tcl, double-click **Create Timing Netlist** in the **Tasks** pane, or use the following Tcl command:

```
create_timing_netlist ↵
```

Constrain the Design

After you create a timing netlist, add constraints either interactively or from an **.sdc** file. To read the **.sdc** file for the current revision of your project, double-click **Read SDC File** in the **Tasks** pane. To read the **.sdc** file interactively or from a script, use the following Tcl command:

```
read_sdc ↵
```

For more information about reading **.sdc** files in the TimeQuest timing analyzer, refer to "Synopsys Design Constraints File Precedence" on page 7–23.

Update Timing Netlist

You must update the timing netlist after you apply timing constraints. 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 use the following command:

```
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 7–43.

As you verify timing, you may encounter failures along critical paths. If this occurs, you use the timing reports to analyze your design and determine how best to optimize your design. If you modify, remove, or add constraints, you should perform a full compilation. This iterative process allows you to resolve your timing violations in the design.

Table 7–2 describes TimeQuest timing analyzer terminology.

Table 7–2. TimeQuest Timing Analyzer Terms

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.
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 Timing Netlist

The 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 TimeQuest timing analyzer generates a netlist equivalent to the one shown in Figure 7–4.

Figure 7–3. Sample Design

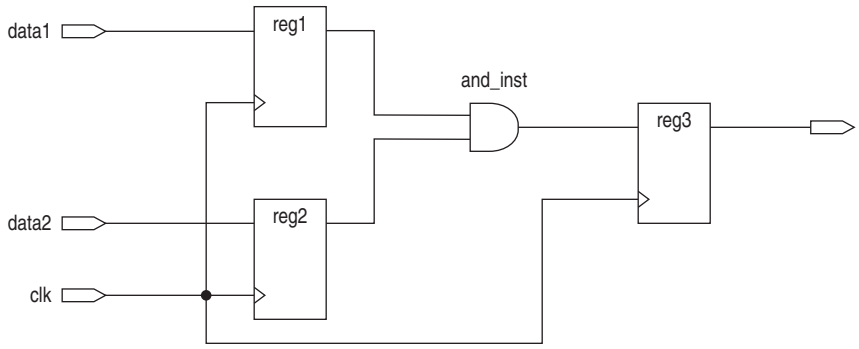


Figure 7–4. The TimeQuest Timing Analyzer Timing Netlist

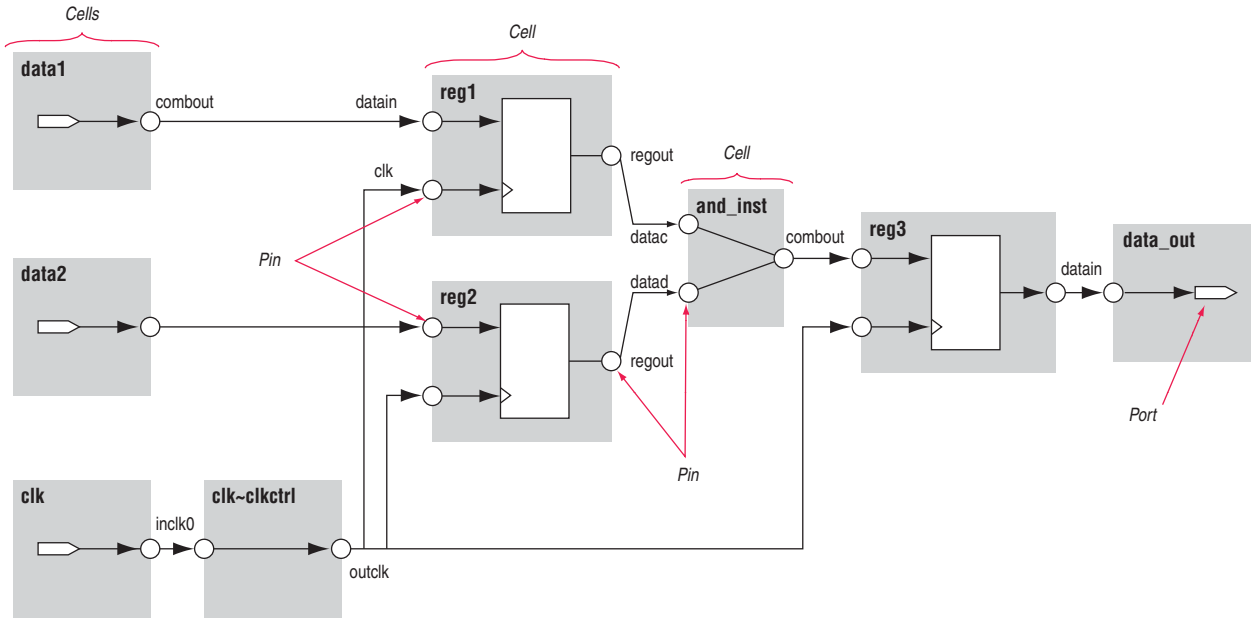


Figure 7–4 shows various cells, pins, nets, and ports. The following sample cell names are included:

- `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:

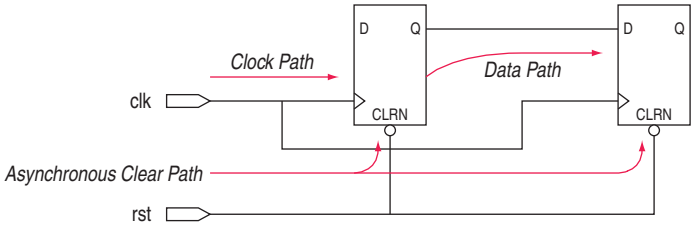
- `data1`
- `clk`
- `data_out`

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.

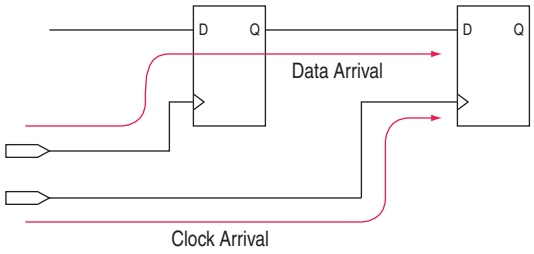
Figure 7–5. Path Types



After the TimeQuest timing analyzer identifies the path type, it can report data and clock arrival times at register pins. The 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 TimeQuest timing analyzer calculates clock arrival time by adding the delay from the clock source to the clock pin of the destination register. Figure 7–6 shows a data arrival path and a clock arrival path. The 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

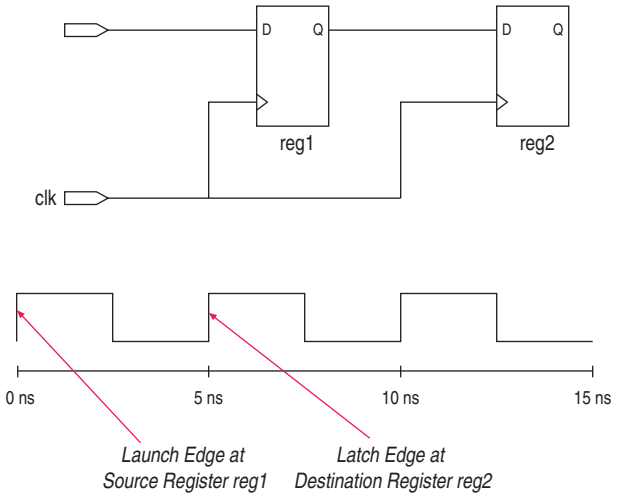


In addition to identifying various paths in a design, the TimeQuest timing analyzer analyzes clock characteristics to compute the worst-case requirement between any two registers in a single register-to-register path. You should constrain all clocks in your design before performing this analysis.

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 TimeQuest timing analyzer validates clock setup and hold requirements relative to the launch and latch edges.

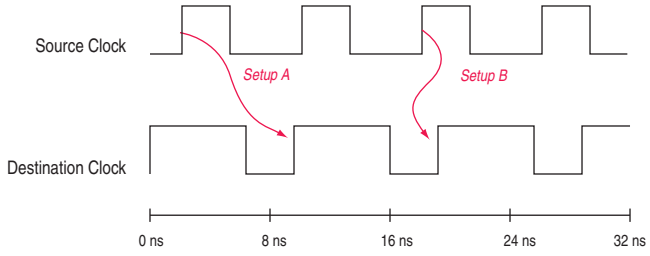
Clock Analysis

A comprehensive static timing analysis includes analysis of register-to-register, I/O, and asynchronous reset paths. The TimeQuest timing analyzer uses data required times, data arrival times, and clock arrival times to verify circuit performance and detect possible timing violations. The 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 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 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 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 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 TimeQuest timing analyzer uses the equations shown in Equation 7–2 to calculate setup slack.

Equation 7–2.

$$\begin{aligned}\text{Clock Setup Slack} &= \text{Data Required Time} - \text{Data Arrival Time}\\ \text{Data Arrival Time} &= \text{Launch Edge} + \text{Clock Network Delay} +\\ &\quad \text{Input Maximum Delay} + \text{Pin-to-Register Delay}\\ \text{Data Required Time} &= \text{Latch Edge} + \text{Clock Network Delay to Destination Register} - \mu t_{SU}\end{aligned}$$

If the data path is an internal register to an output port, the TimeQuest timing analyzer uses the equations shown in Equation 7–3 to calculate the setup slack.

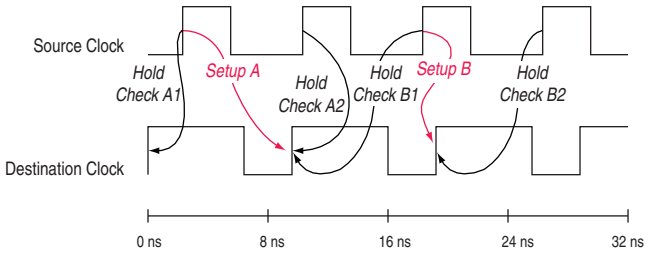
Equation 7–3.

$$\begin{aligned}\text{Clock Setup Slack} &= \text{Data Required Time} - \text{Data Arrival Time}\\ \text{Data Arrival Time} &= \text{Launch Edge} + \text{Clock Network Delay to Source Register} +\\ &\quad \mu t_{CO} + \text{Register-to-Pin Delay}\\ \text{Data Required Time} &= \text{Latch Edge} + \text{Clock Network Delay} - \text{Output Maximum Delay of Pin}\end{aligned}$$

Clock Hold Check

To perform a clock hold check, the TimeQuest timing analyzer determines a hold relationship for each possible setup relationship that exists for all source and destination register pairs. The TimeQuest timing analyzer checks all adjacent clock edges from all setup relationships to determine the hold relationships. The 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 TimeQuest timing analyzer selects the hold relationship that is the most restrictive. The hold relationship with the smallest difference between the latch and launch edges, i.e., launch – latch, 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 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 TimeQuest timing analyzer uses the equations shown in Equation 7–5 to calculate the hold slack.

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 TimeQuest timing analyzer uses the equations shown in Equation 7–6 to calculate the hold slack.

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 for the de-assertion of an asynchronous control signal. For example, signals such as `clear` and `preset` must be stable before the next active clock edge. The recovery slack calculation is similar to the clock setup slack calculation, but it applies to asynchronous control signals. If the asynchronous control signal is registered, the TimeQuest timing analyzer uses Equation 7–7 to calculate recovery slack.

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 TimeQuest timing analyzer uses the equations shown in Equation 7–8 to calculate recovery slack.

Equation 7–8.

Recovery Slack Time = Data Required Time – Data Arrival Time

Data Arrival Time = Launch Edge + Clock Network Delay + Maximum Input Delay + Port-to-Register Delay

Data Required Time = Latch Edge + Clock Network Delay to Destination Register Delay – μt_{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 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 TimeQuest timing analyzer removal slack calculation is similar to the clock hold slack calculation, but it applies asynchronous control signals. If the asynchronous control is registered, the TimeQuest timing analyzer uses the equations shown in Equation 7–9 to calculate the removal slack time.

Equation 7–9.

Removal Slack Time = Data Arrival Time – Data Required Time

Data Arrival Time = Launch Edge + Clock Network Delay to Source Register + μt_{CO} of Source Register + Register-to-Register Delay

Data Required Time = Latch Edge + Clock Network Delay to Destination Register + μt_{H}

If the asynchronous control is not registered, the TimeQuest timing analyzer uses the equations shown in Equation 7–10 to calculate the removal slack time.

Equation 7–10.

Removal Slack Time = Data Arrival Time – Data Required Time

Data Arrival Time = Launch Edge + Clock Network Delay + Input Minimum Delay of Pin + Minimum Pin-to-Register Delay

Data Required Time = Latch Edge + Clock Network Delay to Destination Register + μt_{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 TimeQuest timing analyzer to perform 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

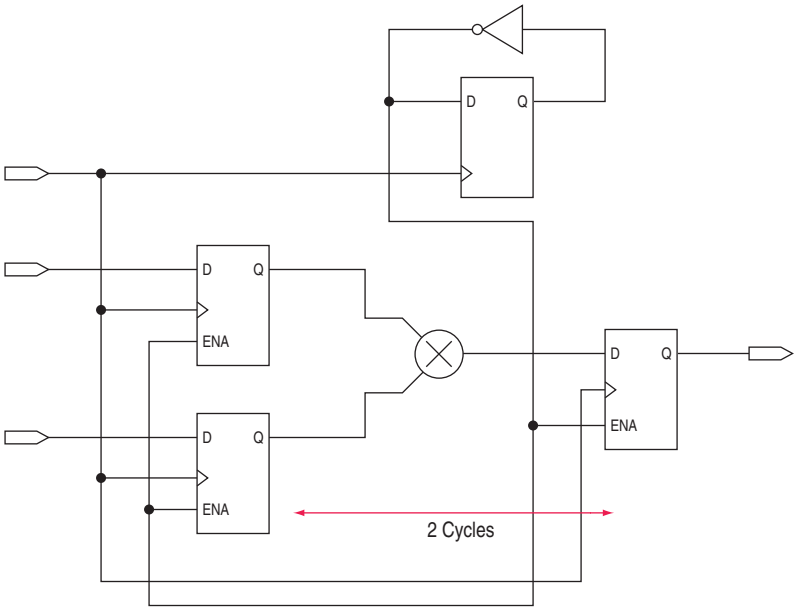


Figure 7–11 shows a register-to-register path.

Figure 7–11. Register-to-Register Path

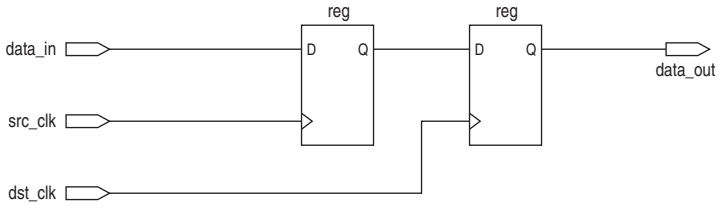
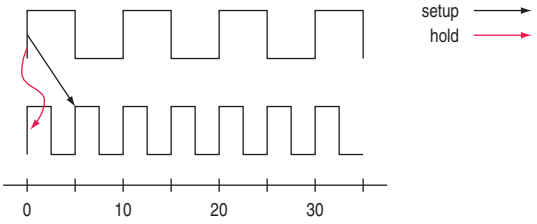


Figure 7–12 shows the respective timing diagrams for the source and destination clocks and the default setup and hold relationships, when the source clock, `src_clk`, has a period of 10 ns and the destination clock, `dst_clk`, has a period of 5 ns. The default setup relationship is 5 ns; the default 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.

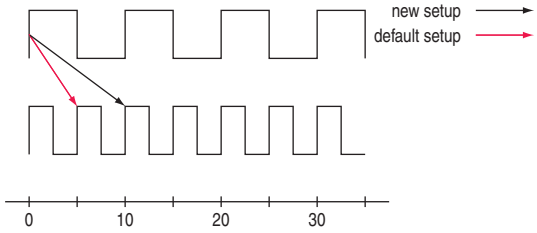
Table 7–3 shows the commands used to modify either the launch or latch edge times that the TimeQuest timing analyzer uses to determine a setup relationship or hold relationship.

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 between 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 7–46 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*.

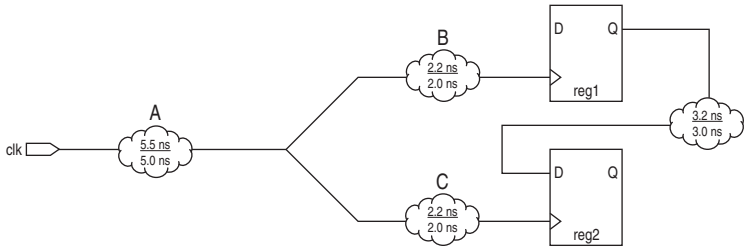
Common Clock Path Pessimism

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

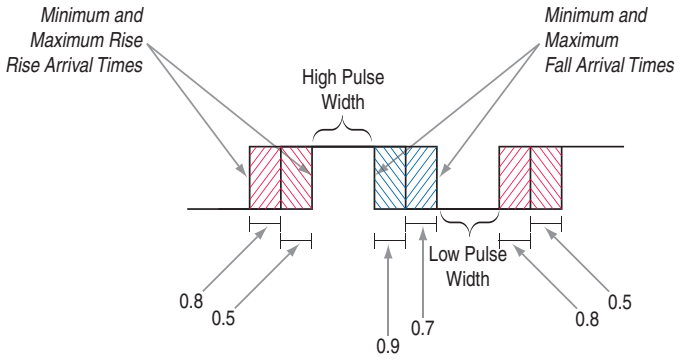


Segment A is the common clock path between reg1 and reg2. The minimum delay is 5.0 ns; the maximum delay is 5.5 ns. The difference between the maximum and minimum delay value equals the CCPP removal value; in this case, CCPP equals 0.5 ns. The CCPP removal value is then added to the appropriate slack 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 7–47 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**.

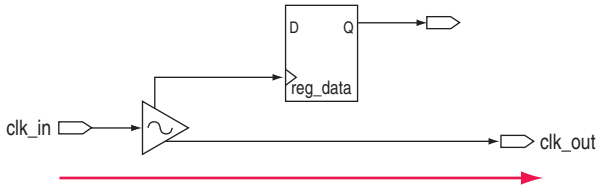
CCPP is supported for Stratix III, Cyclone III, and newer devices.

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.

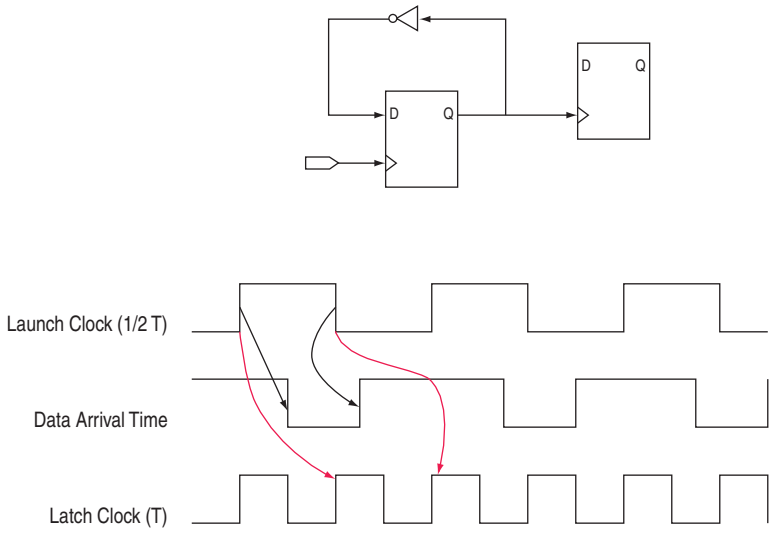
Figure 7–16. Simplified Source Synchronous Output



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.

Collections

The 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 TimeQuest timing analyzer.

Table 7–4 describes the collection commands supported by the TimeQuest timing analyzer.

Table 7–4. SDC 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 Altera SDC extension collection commands.

Table 7–5. Altera SDC Extension Collection Commands

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*>.
`get_registers <filter>`	Returns a collection of registers in the design.
`get_fanins <filter>`	Returns a collection of fan-in nodes starting from <*filter*>.
`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*>.

Avoid use of the `get_keepers` command if your design targets the HardCopy series of devices or you plan to migrate the design to the HardCopy series of devices.

For more information about collections, refer to *API functions for Tcl* in Quartus II help.

Adding and Removing Collection Items

Filters used with collection commands limit collection items identified by the command. For example, if a design contains registers named `src0`, `src1`, `src2`, and `dst0`, the collection command `[get_registers src*]` identifies registers `src0`, `src1`, and `src2`, but not register `dst0`. To identify register `dst0`, an additional command would be required, which is `[get_pins dst*]`.

To overcome this limitation the `add_to_collection` and `remove_from_collection` commands are available. The `add_to_collection` command allows you to add additional items to an existing collection. Example 7–1 shows the `add_to_collection` command and arguments.

Example 7–1. add_to_collection Command

```
add_to_collection <first collection> <second collection>
```

The `remove_from_collection` command allows you to remove items from an existing collection. Example 7–2 shows the `remove_from_collection` command and arguments.

Example 7–2. remove_from_collection Command

```
remove_from_collection <first collection> <second collection>
```

Example 7–3 shows examples of how to add elements to collections.

Example 7–3. Adding Items to a Collection

```
#Setting up initial collection of registers
set regs1 [get_registers a*]

#Setting up initial collection of keepers
set kprs1 [get_keepers b*]

#Creating a new set of registers of $regs1 and $kprs1
set regs_union [add_to_collection $kprs1 $regs1]

#OR
# Creating a new set of registers of $regs1 and b*
# Note that the new collection append only registers with name b*
# not all keepers
set regs_union [add_to_collection $regs1 b*]
```

Refer to `add_to_collection` and `remove_from_collection` in Quartus II Help for full syntax information, options, and example usage.

SDC Constraint Files

The 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 commands. Commands to manipulate the timing netlist or control the compilation flow should be run as part of a separate Tcl script.

The Quartus II software does not automatically create or update **.sdc** files. 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.

Example 7–4 shows the `write_sdc` command and arguments.

Example 7–4. write_sdc Command

```
write_sdc [-expand] [-history] [<file_name>]
```

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.

Refer to `write_sdc` in Quartus II Help for full syntax information, options, and example usage.

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 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 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 TimeQuest timing analyzer as the default timing analyzer to make the Fitter read the **.sdc** file.

Specifying SDC Files for Static Timing Analysis

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 TimeQuest timing analyzer, use the following command:

```
read_sdc [<SDC file name>] ↵
```

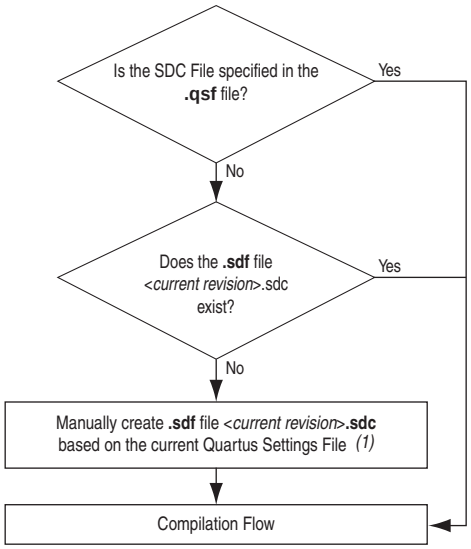
If you use the TimeQuest GUI to apply the **.sdc** file for timing analysis, in the TimeQuest timing analyzer, on the Constraints menu, click **Read SDC File**.

Synopsys Design Constraints File Precedence

The Quartus II Fitter and the 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 7–18.

Figure 7–18. Synopsys Design Constraints File Order of Precedence



Note to Figure 7–18:

(1) This occurs only in the TimeQuest timing analyzer and not during compilation in the Quartus II software. The TimeQuest timing analyzer has the ability to automate the conversion of the QSF timing assignments to SDC if no **.sdc** file exists when the 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 7–18 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 TimeQuest timing analyzer supports many SDC commands that accommodate various clocking schemes and any clock characteristics.

Use the `create_clock` command to create a clock at any register, port, or pin. You can create each clock with unique characteristics. Example 7–5 shows the `create_clock` command and arguments.

Example 7–5. create_clock Command

```
create_clock
-period <period value>
[-name <clock name>]
[-waveform <edge list>]
[-add]
<targets>
```

Example 7–6 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 7–6. 100MHz Clock Creation

```
create_clock -period 10 -waveform { 0 5 } clk
```

Example 7–7 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 7–7. 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 TimeQuest timing analyzer automatically computes the clock's network latency for non-virtual clocks.

Refer to `create_clock` in Quartus II Help for full syntax information, options, and example usage.

Generated Clocks

The 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 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 7–8 shows the `create_generated_clock` command and the available options.

Example 7–8. 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>
```

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 7–19 shows how to create an inverted generated clock based on a 10 ns clock.

Figure 7–19. 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]
```

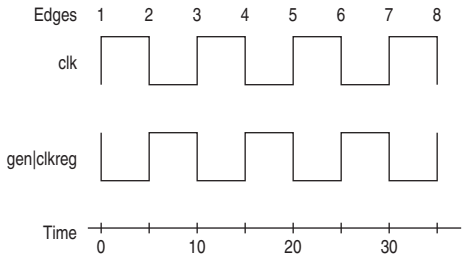


Figure 7–20 shows how to modify the generated clock using the `-edges` and `-edge_shift` options.

Figure 7–20. 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]
```

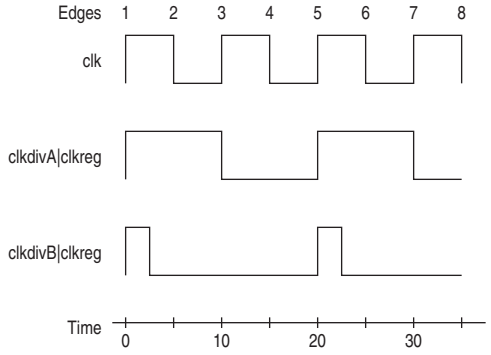
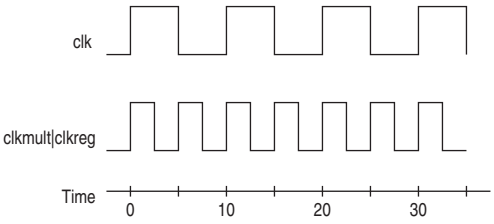


Figure 7–21 shows the effect of the `-multiply_by` option on the generated clock.

Figure 7–21. 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]
```



Refer to `create_generated_clock` in Quartus II Help for full syntax information, options, and example usage.

Virtual Clocks

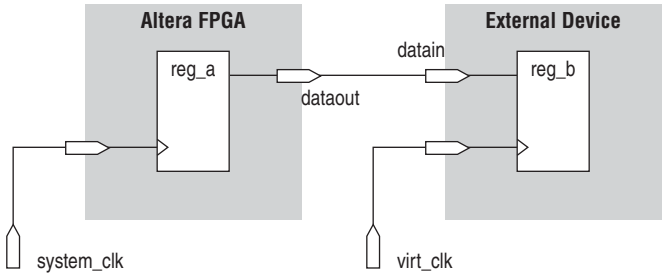
A virtual clock is a clock that does not have a real source in the design or that does not interact directly with the design. For example, if a clock feeds only an external device's clock port and not a clock port in the design, and the external device then feeds (or is fed by) a port in the design, 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 7–22 shows an example where a virtual clock is required for the 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 7–9 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 7–22. 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 7–9. 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]
```

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 7–10 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 TimeQuest timing analyzer uses both clocks when it performs timing analysis.

Example 7–10. 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. The `derive_clocks` command is equivalent to using `create_clock` for each register or port feeding the clock pin of a register. This command creates clocks on ports or registers to ensure every register in the design has a clock, and it applies one period to all base clocks in your design.

Example 7–11 shows the `derive_clocks` command and options.

Example 7–11. derive_clocks Command

```
derive_clocks
[-period <period value>]
[-waveform <edge list>]
```

Use the `derive_pll_clocks` command, instead of the `derive_clocks` command, to create clocks for PLL outputs.

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.

Refer to `derive_clocks` in Quartus II Help for full syntax information, options, and example usage.

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.

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 7–12 shows the `derive_pll_clocks` command and options.

Example 7–12. derive_pll_clocks Command

```
derive_pll_clocks
[-create_base_clocks]
[-use_net_name]
```

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 TimeQuest timing analyzer issues a warning message similar to Example 7–13 when the timing netlist is updated.

Example 7–13. 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.
```

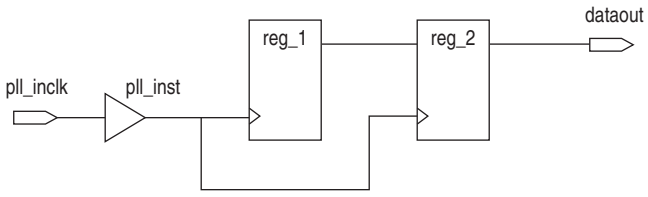
☞ You can use the `-create_base_clocks` option to create the input clocks for the PLL inputs automatically.

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 7–23 shows a simple PLL design with a register-to-register path.

Figure 7–23. 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 7–23, the messages shown in Example 7–14 are generated.

Example 7–14. 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 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.

Refer to `derive_pll_clocks` in Quartus II Help for full syntax information, options, and example usage.

Default Clock Constraints

If there are no clock constraints in the design, to provide a complete clock analysis, the TimeQuest timing analyzer, by default, automatically creates clocks for all detected clock nodes in your design that have not been constrained. The 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 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 7–15 shows the `set_clock_groups` command and options.

Example 7–15. set_clock_groups Command

```
set_clock_groups
[-asynchronous | -exclusive]
-group <clock name>
[-group <clock name>]
[-group <clock name>] ...
```

Refer to `set_clock_groups` in Quartus II Help for full syntax information, options, and example usage.

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 7–16 shows the constraints for this.

Example 7–16. 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 for 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 7–17 makes `clk_A` and `clk_B` related in the same group and unrelated with the second group which contains `clk_C`.

Example 7–17. Asynchronous Option Example 1

```
set_clock_groups -asynchronous -group {clk_A clk_B} -group {clk_C}
```

Example 7–18 shows an alternative method of specifying the same constraint as Example 7–17.

Example 7–18. 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 7–19 shows a `set_clock_groups` command and the equivalent `set_false_path` commands.

Example 7–19. 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 7–20 shows the `set_clock_latency` command and options.

Example 7–20. set_clock_latency Command

```
set_clock_latency
-source
[-clock <clock_list>]
[-rise | -fall]
[-late | -early]
<delay>
<targets>
```

Refer to `set_clock_latency` in Quartus II Help for full syntax information, options, and example usage.

The 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. You can specify the uncertainty separately for setup and hold. You can specify separate rising and falling clock transitions. The 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 7–21 shows the `set_clock_uncertainty` command and options.

Example 7–21. 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
```

Refer to `set_clock_uncertainty` in Quartus II Help for full syntax information, options, and example usage.

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 7–22 shows the `derive_clock_uncertainty` command and options.

Example 7–22. derive_clock_uncertainty Command

```
derive_clock_uncertainty
[-overwrite]
[-add]
```

Refer to `derive_clock_uncertainty` in Quartus II Help for full syntax information, options, and example usage.

The TimeQuest timing analyzer automatically applies clock uncertainties to clock-to-clock transfers in the design.

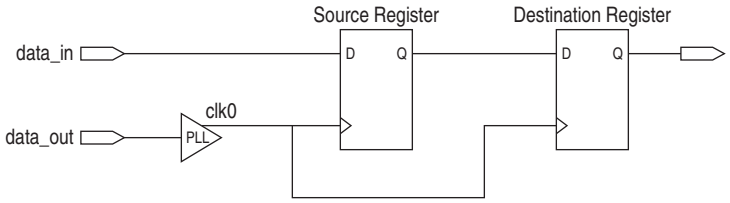
Any clock uncertainty constraints applied to source and destination clock pairs with the `set_clock_uncertainty` command have a higher precedence than the clock uncertainties derived with the `derive_clock_uncertainty` command for the same source and destination clock pairs. For example, if `set_clock_uncertainty` 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.

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 7–24.

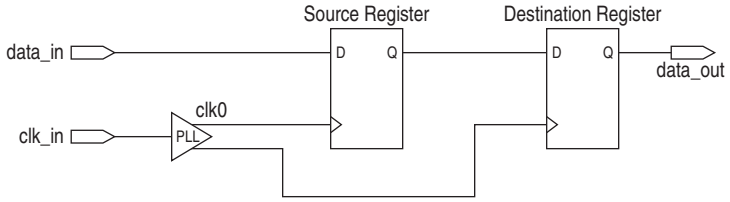
Figure 7–24. Intra-Clock Transfer



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 7–25.

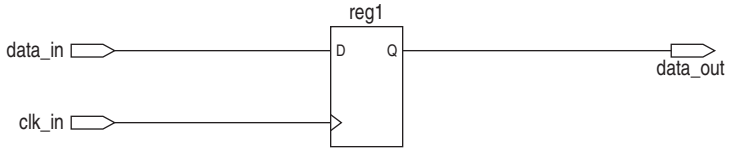
Figure 7–25. Inter-Clock Transfer



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 7–26.

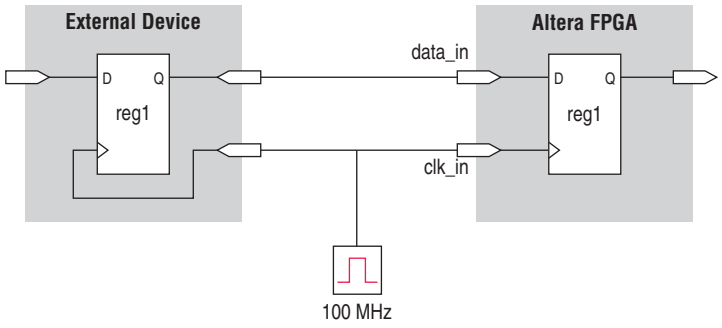
Figure 7–26. I/O Interface-Clock Transfer



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 7–27 shows a typical input I/O interface with the clock specifications.

Figure 7–27. I/O Interface Specifications



Example 7–23 shows the SDC commands to constrain the I/O interface shown in Figure 7–27.

Example 7–23. 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 TimeQuest timing analyzer supports the Synopsys Design Constraint format for constraining timing for the ports in your design. These constraints allow the 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.

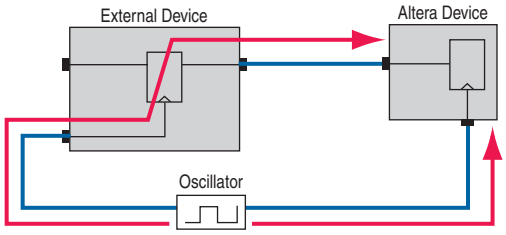
Input and Output Delay

Use input and output delay constraints to specify any external device or board timing parameters. When you apply these constraints, the 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 7–28 shows an input delay path.

Figure 7–28. Set Input Delay



Use the `set_input_delay` command to specify input delay constraints to ports in the design. Example 7–24 shows the `set_input_delay` command and options.

Example 7–24. 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>
```

Refer to `set_input_delay` in Quartus II Help for full syntax information, options, and example usage.

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`, and `-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.

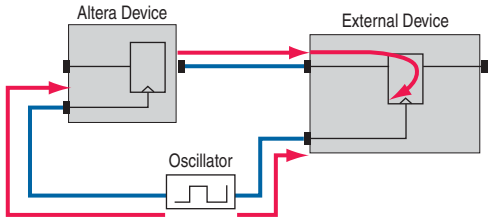
The `-rise` and `-fall` options are mutually exclusive, as are the `-min` and `-max` options.

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 7–29 shows an output delay path.

Figure 7–29. Output Delay



Example 7–25 shows the `set_output_delay` command and options.

Example 7–25. set_output_delay Command

```
set_output_delay
-clock <clock name>
[-clock_fall]
[-rise | -fall]
[-max | -min]
[-add_delay]
[-reference_pin <target>]
<delay value>
<targets>
```

Refer to `set_output_delay` in Quartus II Help for full syntax information, options, and example usage.

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 7–26 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.

Example 7–26. set_net_delay Command

```
set_net_delay
    -from <names>
    [-max]
    [-to <names>]
    [-min]
    <delay>
```

Refer to `set_net_delay` in Quartus II Help for full syntax information, options, and example usage.

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 7–27 shows the `set_max_skew` command and options.

Example 7–27. set_max_skew

```
set_max_skew
[-exclude <Tcl list>]
[-from <names>]
[-include <Tcl list>]
[-to <names>]
<skew>
```

Refer to `set_max_skew` in Quartus II Help for full syntax information, options, and example usage.

By default, the `set_max_skew` command excludes `set_input_delay` and `set_output_delay`. When this constraint is used, the results of max skew analysis are reported with the command `report_max_skew`.

Avoid use of the `set_max_skew` command if your design targets the HardCopy series of devices or you plan to migrate the design to the HardCopy series of devices.

Timing Exceptions

Timing exceptions modify the default analysis performed by the TimeQuest timing analyzer. This section describes the following available timing exceptions:

- "False Path"
- "Minimum Delay" on page 7–39
- "Maximum Delay" on page 7–40
- "Multicycle Path" on page 7–41

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 7–28 shows the `set_false_path` command and options.

Example 7–28. 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>
```

Refer to `set_false_path` in Quartus II Help for full syntax information, options, and example usage.

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 7–29 shows the `set_min_delay` command and options.

Example 7–29. 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>
```

Refer to `set_min_delay` in Quartus II Help for full syntax information, options, and example usage.

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 7–30 shows the `set_max_delay` command and options.

Example 7–30. 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>
```

Refer to `set_max_delay` in Quartus II Help for full syntax information, options, and example usage.

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 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 adjusts 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 7–31 shows the `set_multicycle_path` command and options.

Example 7–31. 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>
```

Refer to `set_multicycle_path` in Quartus II Help for full syntax information, options, and example usage.

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.

For example usage of the `set_multicycle_hold` command, refer to "Multicycle Path" on page 7–41.

Delay Annotation

The TimeQuest timing analyzer provides two commands that allow you to modify the default delay values used during timing analysis. The two commands are `set_annotated_delay` and `set_timing_derate`. 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 7–32 shows the `set_annotated_delay` command and options.

Example 7–32. set_annotated_delay Command

```
set_annotated_delay
   [-cell|-net]
   [-from <names>]
   [-max|-min]
   [-operating_conditions <operating_conditions>]
   [-rr|-fr|-rf|-ff]
   [-to <names>]
   <delay>
```

Refer to `set_annotated_delay` in Quartus II Help for full syntax information, options, and example usage.

With the `-operating_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.

Use the `set_timing_derate` command to specify global derating factors for the current design. The maximum and minimum delays of all timing arcs in the design are multiplied by the specified derating factors using the `-late` and `-early` options respectively.

Only positive derate factors are allowed. If neither the `-cell_delay` nor `-ic_delay` option is used, the derating factors apply to both cell and interconnect delays.

Specifying a derate value of less than 1.0 for the `-late` option or a derate value of greater than 1.0 for the `-early` option reduces delay pessimism, which might incorrectly lead to optimistic results from timing analysis.

The effect of the `set_timing_derate` command is deferred until the next time `update_timing_netlist` is called. To reset derate factors to original values, use the `reset_timing_derate` command.

Example 7–33. set_timing_derate Command and Options

```
set_timing_derate
    [-cell_delay]
    [-early]
    [-ic_delay]
    [-operating_conditions <operating_conditions>]
    [<cells>]
    [<derate_value>]
```

Refer to `set_timing_derate` in Quartus II Help for full syntax information, options, and example usage.

Constraint and Exception Removal

When using the TimeQuest timing analyzer interactively, it is sometimes necessary to remove a constraint or exception. In cases where constraints and exceptions either become outdated or have been erroneously entered, the TimeQuest timing analyzer provides a convenient way to remove them.

Table 7–6 lists commands that allow you to remove constraints and exceptions from a design.

Table 7–6. 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.
`reset_timing_derate`	Resets all timing derating factors to default values.

Timing Reports

The 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 TimeQuest timing analyzer.

report_timing

Use the `report_timing` command to generate a setup, hold, recovery, or removal report. Example 7–34 shows the `report_timing` command and options.

Example 7–34. 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>]
[-rise_from <names>|-fall_from <names>]
[-rise_to <names>|-fall_to <names>]
[-rise_through <names>|-fall_through <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>]
```

Refer to `report_timing` in Quartus II Help for full syntax information, options, and example usage.

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 7–7.

Table 7–7. 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 7–8 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 7–8 provides a description of the possible node type in the `report_timing` reports.

Table 7–8. Type Description

Type Name	Description
CELL	Indicates the element is either a register or a combinational element in the FPGA; the CELL can be a register in the ALM, memory blocks, DSP blocks, or I/O block.
COMP	Indicates the PLL clock network compensation delay.
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 7–35 shows the `report_exceptions` command and options.

Example 7–35. 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>]
```

Refer to `report_exceptions` in Quartus II Help for full syntax information, options, and example usage.

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 7–36. report_metastability command

```
report_metastabiity
[-append]
[-file <name>]
[-panel_name <name>]
[-stdout]
```

Refer to `report_metastability` in Quartus II Help for full syntax information, options, and example usage.

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 7–37 shows the `report_clock_transfers` command and options.

Example 7–37. report_clock_transfers Command

```
report_clock_transfers
[-append]
[-file <name>]
[-hold]
[-setup]
[-stdout]
[-recovery]
[-removal]
[-panel_name <name>]
```

Refer to `report_clock_transfers` in Quartus II Help for full syntax information, options, and example usage.

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 7–38 shows the `report_clocks` command and options.

Example 7–38. report_clocks Command

```
report_clocks
[-append]
[-desc]
[-file <name>]
[-stdout]
[-panel_name <name>]
```

Refer to `report_clocks` in Quartus II Help for full syntax information, options, and example usage.

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 7–39 shows the `report_min_pulse_width` command and options.

Example 7–39. report_min_pulse_width Command

```
report_min_pulse_width
[-append]
[-file <name>]
[-nworst <number>]
[-stdout]
[<targets>]
[-detail_<summary | full_path>]
[-panel_name <name>]
```

Refer to `report_min_pulse_width` in Quartus II Help for full syntax information, options, and example usage.

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 7–40 shows the `report_net_timing` command and options.

Example 7–40. report_net_timing Command

```
report_net_timing
[-append]
[-file <name>]
[-nworst_delay <number>]
[-nworst_fanout <number>]
[-stdout]
[-panel_name <name>]
```

Refer to `report_net_timing` in Quartus II Help for full syntax information, options, and example usage.

report_sdc

Use the `report_sdc` command to generate a report of all the Synopsys design constraints in the project.

Example 7–41 shows the `report_sdc` command and options.

Example 7–41. report_sdc Command

```
report_sdc
[-ignored]
[-append]
[-file]
[-stdout]
[-panel_name <name>]
```

Refer to `report_sdc` in Quartus II Help for full syntax information, options, and example usage.

report_ucp

Use the `report_ucp` command to generate a report of all unconstrained paths in the design.

Example 7–42 shows the `report_ucp` command and options.

Example 7–42. report_ucp Command

```
report_ucp
[-append]
[-file <name>]
[-hold]
[-setup]
[-stdout]
[-summary]
[-panel_name <name>]
```

Refer to `report_ucp` in Quartus II Help for full syntax information, options, and example usage.

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 7–43 shows the `report_bottleneck` command and options.

Example 7–43. 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`.

Refer to `report_bottleneck` in Quartus II Help for full syntax information, options, and example usage.

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`), output enable (`tzx`), minimum output enable (`mintzx`), output disable (`txz`), minimum output disable (`mintxz`), propagation delay (`tpd`), and minimum propagation delay (`mintpd`) times. Example 7–44 shows the `report_datasheet` command and options.

Example 7–44. report_datasheet Command

```
report_datasheet
[-append]
[-file <name>]
[-stdout]
[panel_name <name>]
```

Refer to `report_datasheet` in Quartus II Help for full syntax information, options, and example usage.

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 7–45 shows the `report_rskm` command and options.

Example 7–45. report_rskm Command

```
report_rskm
[-append]
[-file <name>]
[-panel_name <name>]
[-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.

Refer to `report_rskm` in Quartus II Help for full syntax information, options, and example usage.

report_tccs

Use the `report_tccs` command to generate a report that details the channel-to-channel skew margin for LVDS transmitters. TCCS is the timing difference between the fastest and slowest output transitions, including t_{CO} variations and clock skew.

Example 7–46 shows the `report_tccs` command and options.

Example 7–46. report_tccs Command

```
report_tccs
[-append]
[-file <name>]
[-panel_name <name>]
[-quiet]
[-stdout]
```

Refer to `report_tccs` in Quartus II Help for full syntax information, options, and example usage.

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 7–47 shows the `report_partitions` command and options.

Example 7–47. report_partitions Command

```
report_partitions
[-nworst <number>]
[-panel_name <name>]
[-stdout]
```

Refer to `report_partitions` in Quartus II Help for full syntax information, options, and example usage.

report_path

Use the `report_path` command to generate a report that details the longest delay paths between any two arbitrary keeper nodes.

Example 7–48 shows the `report_path` command and options.

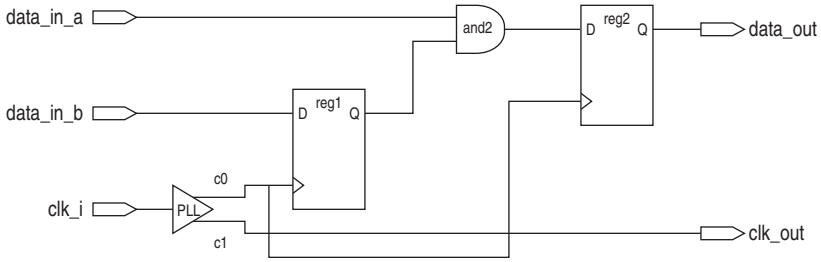
Example 7–48. report_path Command

```
report_path
[-append]
[-file <name>]
[-from <names>]
[-min_path]
[-npaths <number>]
[-nworst <number>]
[-panel_name <name>]
[-stdout]
[-summary]
[-through <names>]
[-to <names>]
```

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 7–30 shows a simple design with a register-to-register path.

Figure 7–30. Simple Register-to-Register Path



Example 7–49 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 7–49. 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 7–50 shows the report generated from the following command:

```
report_path -from [get_ports data_in_a] -to [get_pins {reg2|regout}] \
-npaths 1
```

Example 7–50. report_path from Keeper-to-Keeper Output Pin

```
Info: Report Path: No paths were found
0 0.000
```

No paths were reported in Example 7–50 because the destination passes through an input pin of a keeper node.

Refer to `report_path` in Quartus II Help for full syntax information, options, and example usage.

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 7–51 shows the `report_net_delay` command and options.

Example 7–51. report_net_delay Command

```
report_net_delay
    [-append]
    [-file <name>]
    [-nworst <number>]
    [-panel_name <name>]
    [-stdout]
```

Refer to `report_net_delay` in Quartus II Help for full syntax information, options, and example usage.

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 path matching the criteria set by that `set_max_skew` command. These results are ordered based on the slack value. Example 7–52 shows the `report_max_skew` command and options.

Example 7–52. 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]
```

No results are displayed if the `-from/-from_clock` and `-to/-to_clock` are applied to less than two paths.

Refer to `report_max_skew` in Quartus II Help for full syntax information, options, and example usage.

report_skew

This report performs skew analysis on specified paths. If valid `set_max_skew` constraints exist, this report computes skew with respect to the latest and the earliest arrival of each path.

Unlike the `report_max_skew` command, the `report_skew` command does not depend on the existence of `set_max_skew` assignments.

Skew for the latest arrival is computed by comparing the latest arrival of each path with the earliest arrival of the path that has the smallest value for early arrival of all other paths included in the constraint. Similarly, skew for the earliest arrival is computed by comparing the earliest arrival of each path with the latest arrival of the path that has the largest value for late arrival of all other paths included in the constraint. No path is compared with itself.

The return value of this command is a two-element list. The first number is the number of paths found in the analysis. The second is the worst-case skew, in terms of the current default time unit.

Example 7–53 shows the `report_skew` command and options.

Example 7–53. report_skew Command Options

```
report_skew
-detail <summary|path_only|path_and_clock|full_path>]
   [-exclude <Tcl list>]
   [-fall_from_clock <names>]
   [-fall_to_clock <names>]
   [-file <name>]
   [-from <names>]
   [-from_clock <names>]
   [-greater_than_skew <slack limit>]
   [-include <Tcl list>]
   [-npaths <number>]
   [-panel_name <name>]
   [-rise_from_clock <names>]
   [-rise_to_clock <names>]
   [-show_routing]
   [-stdout]
   [-through <names>]
   [-to <names>]
   [-to_clock <names>]
```

Refer to `report_skew` in Quartus II help for full syntax information, options, and example usage.

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 7–54 shows the `check_timing` command and options.

Example 7–54. check_timing Command

```
check_timing
[-append]
[-file <name>]
[-include <check_list>]
[-stdout]
[-panel_name <name>]
```

Refer to `check_timing` in Quartus II help for full syntax information, options, and example usage.

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 7–55 shows the `report_clock_fmax_summary` command and options.

Example 7–55. report_clock_fmax_summary Command

```
report_clock_fmax_summary
[-append]
[-file <name>]
[-panel_name <name>]
[-stdout]
```

Refer to `report_clock_fmax_summary` in Quartus II help for full syntax information, options, and example usage.

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 7–9.

Table 7–9. 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 7–56 shows the `create_timing_summary` command and options.

Example 7–56. create_timing_summary Command

```
create_timing_summary
[-append]
[-file <name>]
[-hold]
[-mpw]
[-panel_name <name>]
[-recovery]
[-removal]
[-setup]
[-stdout]
```

Refer to `create_timing_summary` in Quartus II Help for full syntax information, options, and example usage.

Other Features

The TimeQuest timing analyzer supports many features that enhance and provide a through analysis of your designs, such as multi-corner analysis, and advanced I/O timing.

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 7–57 shows the `set_operating_conditions` command and options.

Example 7–57. set_operating_conditions Command

```
set_operating_conditions
[-model <fast|slow>]
[-speed <speed grade>]
[-temperature <value in ºC>]
[-voltage <value in mV>]
[-force_dat]
[-grade <c | i>]
[<operating condition Tcl object>]
```

Table 7–10 describes the options for the `set_operating_conditions` command.

Table 7–10. 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 ºC>`	Specifies the operating temperature.
`-voltage <value in mV>`	Specifies the operating voltage.
`-force_dat`	Forces a re-annotation of the timing delays for the design.
`-grade <i|c>`	Allows the specification of industrial (i) or commercial (c) speed grades. For timing sign-off, the value of the -grade option should match the speed grade of the targeted device.
`<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.

Use the `get_available_operating_conditions-all` command to obtain a list of available operating conditions for the target device.

Table 7–11 shows the operating conditions of each model for device families that support only two operating conditions, that is, fast and slow.

Table 7–11. 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 7–11:

(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 7–58 shows how to set the operating conditions for a Stratix III design to the slow model, 1100 mV, and 85° C.

Example 7–58. Setting Operating Conditions with Individual Values

```
set_operating_conditions -model slow -temperature 85 -voltage 1100
```

Alternatively, you can set the operating conditions in Example 7–58 with the Tcl object as shown in Example 7–59.

Example 7–59. Setting Operating Conditions with a Tcl Object

```
set_operating_conditions 4_slow_1100mv_85c
```

Example 7–60 shows how to use the set_operating_conditions command to generate different reports for various operating conditions.

Example 7–60. Script Excerpt for Analysis of Various Operating Conditions

```
s#Specify initial operating conditions
set_operating_conditions -model slow -speed 3 -grade c -temperature 85
-voltage 1100

#update the timing netlist with the initial conditions
update_timing_netlist

#Perform reporting

#Change initial operating conditions. Use a temperature of 0C
set_operating_conditions -model slow -speed 3 -grade c -temperature 0 -
voltage 1100

#update the timing netlist with the new operating condition
update_timing_netlist

#Perform reporting

#Change initial operating conditions. Use a temperature of 0C and a
model of fast
set_operating_conditions -model fast -speed 3 -grade c -temperature 0 -
voltage 1100

#update the timing netlist with the new operating condition
update_timing_netlist

#Perform reporting
```

Advanced I/O Timing and Board Trace Model Assignments

The 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 enable the Advanced I/O Timing feature, in the **Settings** dialog box, on the **TimeQuest Timing Analyzer** page, turn on **Enable Advanced I/O Timing**.

If you change the **Advanced I/O Timing** setting or change any board trace model assignments and do not recompile before you analyze timing, you must use the `-force_dat` command when you create a timing netlist. Type the following command in the Tcl console of the TimeQuest timing analyzer:

```
create_timing_netlist -force_dat ↵
```

If you change the **Advanced I/O Timing** settings or change any board trace model assignments and recompile before you analyze timing, you do not have to use the `-force_dat` command when you create the timing netlist.

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 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 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 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 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 7–12 shows the results of using these search strings.

Table 7–12. Sample Search Strings and Search Results

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)*
`get_pins -compatibility *|datac`	`foo|bar|datac`
`get_pins -compatibility *|*|datac`	`foo|bar|datac`

Note to Table 7–12:

(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 7–61 shows the `locate` command and options.

Example 7–61. locate Command

```
locate
[-chip]
[-color <black|blue|brown|green|grey|light_grey|orange|purple|red|white>]
[-cps]
[-label <label>]
[-rpe]
[-tmv]
<items>
```

Table 7–13 describes the options for the `locate` command.

Table 7–13. 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 7–62 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 7–62. 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 Quartus II Software Options and Compilation Report

The Quartus II software allows you to configure various options for the 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 7–14.

Table 7–14. TimeQuest Timing Analyzer Settings

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.
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 7–14 control only the reports generated in the Compilation Report, and do not control the reports generated in the TimeQuest timing analyzer.

You can also use the `TIMEQUEST_REPORT_WORST_CASE_TIMING_PATHS` assignment to generate a report of the worst-case timing paths for each clock domain. This report contains worst case timing data for setup, hold, recovery, removal, and minimum pulse width checks.

Use the `TIMEQUEST_REPORT_NUM_WORST_CASE_TIMING_PATHS` assignment to specify the number of paths to report for each clock domain.

Example 7–63 shows the `TIMEQUEST_REPORT_WORST_CASE_TIMING_PATHS` and `TIMEQUEST_REPORT_NUM_WORST_CASE_TIMING_PATHS` assignments as they appear in a **.qsf**.

Example 7–63. Specifying generation of a Worst Case Timing Report

```
#Enable Worst Case Timing Report
set_global_assignment -name TIMEQUEST_REPORT_WORST_CASE_TIMING_PATHS ON
#Report 10 paths per clock domain
set_global_assignment -name TIMEQUEST_REPORT_NUM_WORST_CASE_TIMING_PATHS 10
```

Conclusion

The 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 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.

Document Revision History

Table 7–15 shows the revision history for this chapter.

Table 7–15. Document Revision History (Part 1 of 2)

Date	Version	Changes Made
July 2010	10.0.0	Updated to link to content on SDC commands and the TimeQuest timing analyzer GUI in Quartus II help.
November 2009	9.1.0	Updated for the Quartus II software version 9.1, including: ■ Added information about commands for adding and removing items from collections ■ Added information about the set_timing_derate and report_skew commands ■ Added information about worst-case timing reporting ■ Minor editorial updates

Table 7–15. Document Revision History (Part 2 of 2)

Date	Version	Changes Made
November 2008	8.1.0	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
May 2008	8.0.0	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

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

Take an online survey to provide feedback about this handbook chapter.

8. Best Practices for the Quartus II TimeQuest Timing Analyzer

QII53024-10.0.0

Timing constraints and exceptions are vital to all designs that target FPGAs, because they allow designers to specify requirements and verify timing of their systems or FPGAs. This chapter provides the steps to fully constrain an FPGA design with the Quartus® II TimeQuest Timing Analyzer in the following sections:

- "Clock Requirements"
- "I/O Requirements" on page 8–4
- "Exceptions" on page 8–5

The sections are ordered in the recommended flow for applying timing constraints and exceptions in the TimeQuest timing analyzer.

For more information about constraints and exceptions supported by the TimeQuest timing analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Clock Requirements

The TimeQuest timing analyzer supports the following 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 your design.

Clocks should be the first constraints specified in Synopsys Design Constraint (**.sdc**) files. This is important because the TimeQuest timing analyzer reads **.sdc** constraints and exceptions from top to bottom in the file. Clocks should be defined first, because other constraints may reference previously defined clocks.

Base Clocks

Base clocks are the primary input clocks generated into the FPGA. Unlike phase-locked loops (PLLs) that are derived in the FPGA, base clocks are usually generated in off-chip oscillators or forwarded from an external device. Base clocks are defined first, because derived clocks and other constraints can reference the base clocks.

Use the `create_clock` SDC command to constrain all primary input clocks. The target for the `create_clock` command is usually an FPGA device pin. To specify the FPGA device pin as the target, you must use the `get_ports` commands. Example 8–1 shows how to specify a 100-MHz requirement on a `clk_sys` input clock port.

Example 8–1. create_clock Command

```
create_clock -period 10 [get_ports clk_sys]
```

You can apply multiple clocks on the same clock node with the `-add` option. If two oscillators drive the same clock port on the FPGA, the following SDC commands are used:

```
create_clock -period 10 -<name> clk_100 [get_ports clk_sys] ↵
create_clock -period 5 -<name> clk_200 [get_ports clk_sys] -add ↵
```

Derived Clocks

Derived clocks are generated in the FPGA and can modify or synthesize a source clock signal. Derived clocks, which are PLLs or register clock dividers, are constrained after all base clocks are constrained in the **.sdc** file. Derive clocks capture all clock delays/latency where the derive clock target is defined. This ensures that all base clock properties are accounted for in the derived clock.

You can use the `create_generated_clock` command to constrain all generated clocks in your design. The source of the `create_generated_clock` command should be a node in your design and not a previously constrained clock.

The TimeQuest timing analyzer supports the `derive_pll_clocks` command to automatically constrain all PLL outputs in the FPGA.

Example 8–2 shows a divide-by-two clock divider.

Example 8–2. Clock Divider

```
create_clock -period 10 [get_ports clk_sys]
create_generated_clock -name clk_div_2 -divide_by 2 -source [get_pins reg|clk] [get_pins
reg|regout]
```

When using the `create_generated_clock` constraint, Altera recommends that the `-source` option point to the closest clock pin of the specified target and not the clock port. In Example 8–2, the `-source` option points to the clock pin of the register instead of the clock port `clk` feeding the register's reg clock pin. Use this when multiple clock constraints are specified for the same pin in a design.

The TimeQuest provides the `derive_pll_clocks` command that automatically generates derive clocks for all PLL clock outputs. The properties of the generated clocks on the PLL outputs will match those that have been defined for the PLL.

Virtual Clocks

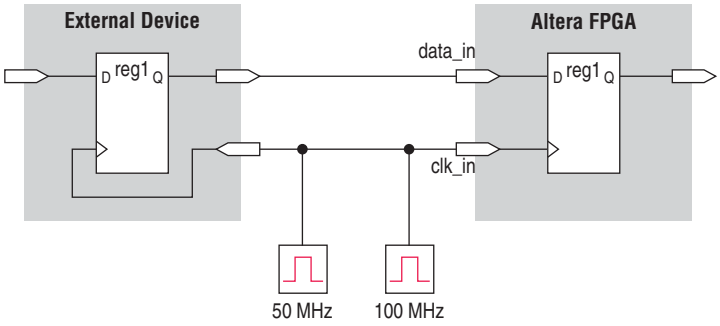
A virtual clock does not have a real source in your design and does not interact directly with your design. Virtual clocks are created with the `create_clock` command, but no targets are specified. Example 8–3 shows the command to create a 10 ns virtual clock.

Example 8–3. Creating a 10 ns Virtual Clock

```
create_clock -period 10 -name my_virt_clk ↵
```

Figure 8–1 and Example 8–4 show when a virtual clock is 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 is modeled with a virtual clock.

Figure 8–1. Inter-Clock Transfer



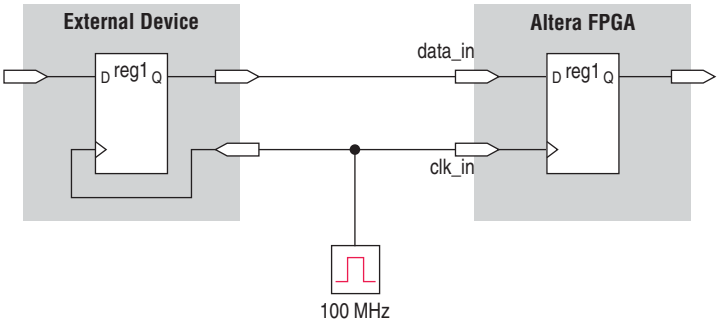
Example 8–4. Virtual Clock Example 1

```
#create base clock for the design
create_clock -period 5 [get_ports clk_in]
#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 data_out]
```

Altera recommends using virtual clocks if you model external delays with the `set_input_delay` and `set_output_delay` constraints. This is important if you use the `derive_clock_uncertainty` command for your 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.

Altera recommends creating an equivalent virtual clock for each clock in your design that feeds an input or output port (refer to Figure 8–2 and Example 8–5).

Figure 8–2. I/O Interface Specifications



Example 8–5. .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

You should specify all timing requirements to fully analyze a design. This includes specifying internal and external timing requirements. With external timing requirements specified, the I/O interface or periphery of the FPGA is verified against any system specification. The TimeQuest timing analyzer supports input and output external delay modeling.

You should specify I/O requirements after all clocks in your design are constrained. When specifying I/O requirements, Altera recommends referencing a virtual clock in the constraints.

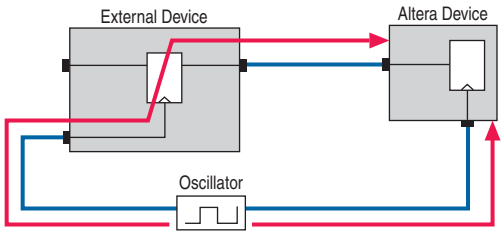
For guidelines to determine delay values for `set_input_delay` and `set_output_delay`, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Input Requirements

Input requirements allow all the external delays feeding into the FPGA to be specified. The requirements are specified for all input ports in your design.

You can use the `set_input_delay` command to specify external input delay requirements. Altera recommends that the `set_input_delay` command reference a virtual clock for the `-clock` option. Using a virtual clock allows the TimeQuest timing analyzer to correctly derive clock uncertainties for inter and intra clock transfers. 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 are defined. Figure 8–3 shows an example of an input delay referencing a virtual clock.

Figure 8–3. Set Input Delay

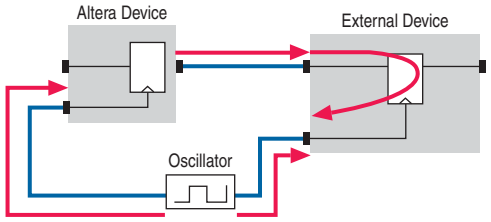


Output Requirements

The output requirements allow all external delays from the FPGA to be specified for all output ports in your design.

You can use the `set_output_delay` command to specify external output delay requirements. The `set_output_delay` command must 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 are 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 TimeQuest timing analyzer provide a way to modify the default timing analysis behavior to match the analysis required by your design. The TimeQuest timing analyzer supports the following three major categories of exceptions:

- False paths
- Minimum and maximum delays
- Multicycles

Timing exceptions are specified after clocks, and input and output delay constraints, because timing exceptions can modify the default analysis.

False Paths

You remove a specified path from analysis when you specify a false path in your design. This path is either a point-to-point or clock-to-clock path. An example is a static configuration register that is written once during power-up initialization, but will not change state again. Although these signals often cross clock domains, you may not want to make false path exceptions to a clock-to-clock path, because some data may transfer across those clock domains. However, you can selectively make false path exceptions from the static configuration register to all endpoints.

Example 8–6 shows how to make false path exceptions from all registers beginning with **A** to all registers beginning with **B**.

Example 8–6. False Path

```
set_false_path -from [get_pins A*] -to [get_pins B*]
```

The TimeQuest timing analyzer assumes all clocks are related unless you specify otherwise. Setting clock groups is an efficient way of making false path exceptions for clock-to-clock timing relationships in your design. It requires fewer line entries to make false path exceptions 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 where multiple clocks are 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. The clocks cannot physically exist in your design at the same time.

Minimum and Maximum Delays

Use the `set_max_delay` and `set_min_delay` constraints for asynchronous signals that do not have a specific clock relationship in your design, but require a bounded maximum and minimum path delay. This timing exception applies to paths that go from port to port through the FPGA without a register stage in the path. If you use this timing exception to constrain the path delay, specify both the maximum and minimum delay of the path. Do not constrain only the maximum or the minimum value. The `set_max_delay` and `set_min_delay` commands modify the setup and hold relationship equal to the constraint's value.

You can also use the `set_net_delay` constraint to specify the maximum, minimum, or skew for any edge in your design. This constraint is used when 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. You must understand the design functionality to understand 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 your design can increase performance of the Fitter and provide better quality of results during compilation, because the coverage relaxes some invalid setup and hold edge relationships.

An example of a potential multicycle path would be long combinational paths in which the latching register does not require data stability on every clock edge, but only on every second clock edge. This multicycle path is dependent on the endpoint register's use of that signal. In this case, the `set_multicycle_path -setup 2` command states that data is stable at the endpoint every two clock cycles of the endpoint latch clock.

If you specify a multicycle path, Altera recommends defining both the setup and hold multicycle relationships. For the preceding example, setting data at the endpoint can take two clock cycle, and the minimum hold time relationship is defined with a multicycle as well. The `set_multicycle_path -hold` value is (N – 1), in which N is equal to the `set_multicycle_path -setup` value for a register-to-register path in the same clock domain. However, if data crosses 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 multicycles 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 your 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. Follow the guidelines and flow in this chapter for successful design implementation in the Quartus II software.

Document Revision History

Table 8–1 shows the revision history for this chapter.

Table 8–1. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	Minor editorial changes to document. No updates to technical content.
November 2009	9.1.0	■ Added example to "Derived Clocks" on page 8–2. ■ Updated "Base Clocks" on page 8–1.
March 2009	9.0.0	Initial release.

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

Take an online survey to provide feedback about this handbook chapter.

9. Switching to the Quartus II TimeQuest Timing Analyzer

QII53019-10.0.0

This chapter describes the benefits of switching to the Quartus II TimeQuest Timing Analyzer, the differences between the TimeQuest timing analyzer and Classic timing analyzer, and the process you should follow to switch a design from using the Classic timing analyzer to the 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.

These features ease constraint and analysis of modern, complex designs. SDC constraints support complex clocking schemes and high-speed interfaces. 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 TimeQuest timing analyzer.

There are three main differences between the Classic timing analyzer and the TimeQuest timing analyzer. Unlike the Classic timing analyzer, the TimeQuest timing analyzer has the following three benefits:

- All clocks are related by default. Refer to "Related and Unrelated Clocks" on page 9–10.
- The default hold multicycle value is zero. Refer to "Hold Multicycle" on page 9–18.
- You must constrain all ports and ripple clocks. Refer to "Automatic Clock Detection" on page 9–15.

Chapter Contents

This chapter contains the following sections:

- "Switching Your Design" describes the four-step process you should follow to switch a design from the Classic Timing Analyzer to the TimeQuest timing analyzer.
- "Differences Between the Timing Analyzers" on page 9–3 describes terminology, constraints, clocks, hold multicycle, and other differences.
- "Timing Assignment Conversion" on page 9–24 is a comprehensive guide to converting Quartus II Classic QSF timing assignments to Quartus II TimeQuest SDC constraints.

- "Conversion Utility" on page 9–41 describes a utility that helps you convert Classic QSF timing assignments to the Quartus II TimeQuest SDC constraints.
- "Notes" on page 9–50 includes notes about support for specific features in the current version of the TimeQuest timing analyzer.

Switching Your Design

You should use the following process to switch a design from the Classic timing analyzer to the TimeQuest timing analyzer. The process is composed of the following steps, which are described in detail in the next sections:

1. Compile your design in the Quartus II software with the Classic timing analyzer set to perform timing analysis (page 9–2).
2. Create a Synopsys Design Constraints (**.sdc**) file that contains timing constraints (page 9–2).
3. Perform timing analysis with the TimeQuest timing analyzer and examine the reports (page 9–3).
4. Set the default timing analyzer to the TimeQuest Timing Analyzer (page 9–3).

Compile Your Design

To begin, compile your design with the Quartus® II software, and with the Classic timing analyzer. To run the Classic timing analyzer in the Quartus II GUI, on the Processing menu, click **Start**, then click **Start Classic Timing Analyzer**. To run the Classic timing analyzer if you are a command-line user, type `quartus_tan` <*project*> ↵ at a system command prompt.

Create an .sdc File

The 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 "Start the Quartus II TimeQuest Timing Analyzer" on page 9–3. Name the **.sdc** file <*revision*>**.sdc** (where <*revision*> is the current revision of your project) and save it in your project directory.

For more information about TimeQuest SDC constraints, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

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 the following command at a system command prompt:

```
quartus_sta --qsf2sdc <project name> ↵
```

The **.sdc** file created by the conversion utility is named <*revision*>**.sdc**.

After conversion, review the **.sdc** file to ensure it is correct and complete, and make changes if necessary. Refer to "Constraint File Priority" on page 9–7 for recommendations.

The conversion utility cannot convert some types of Classic timing analyzer assignments if no corresponding SDC constraint exists, or if there is more than one potentially equivalent SDC constraint. In these cases, correct conversion requires knowledge of the intended function of your design. Manually convert any ambiguous assignments based on the guidelines in "Timing Assignment Conversion" on page 9–24.

Start the Quartus II TimeQuest Timing Analyzer

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 TimeQuest timing analyzer, type `quartus_staw ↵` to open the Quartus II TimeQuest GUI, or type `quartus_sta -s ↵` to start the TimeQuest timing analyzer Tcl shell.

Set the Default Timing Analyzer

To set the TimeQuest timing analyzer as the default timing analyzer for your project, 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
```

The default timing analyzer setting is specific to each project, so you can decide on a per-project basis whether to use the TimeQuest timing analyzer or the Classic timing analyzer.

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 Classic timing analyzer uses assignments in the **.qsf** file, and the 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 the Timing Analyzers

The differences between the TimeQuest timing analyzer and the Classic timing analyzer are described in the following sections:

- "Terminology" on page 9–4
- "Constraints" on page 9–5

- "Clocks" on page 9–10
- "Hold Multicycle" on page 9–18
- "Fitter Behavior" on page 9–20
- "Reporting" on page 9–20
- "Timing Assignment Conversion" on page 9–24

Terminology

This section introduces the industry-standard SDC terminology used by the TimeQuest timing analyzer.

Netlists

The 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 Classic timing analyzer. In the 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 sample 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

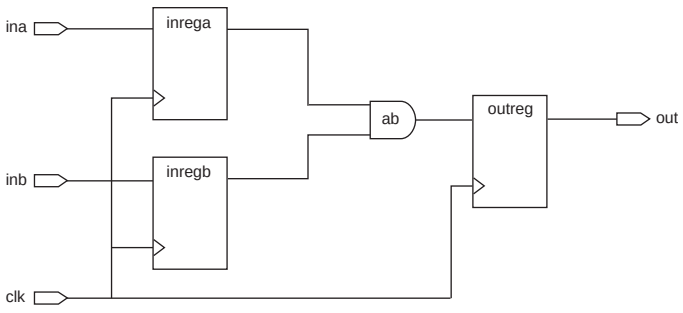
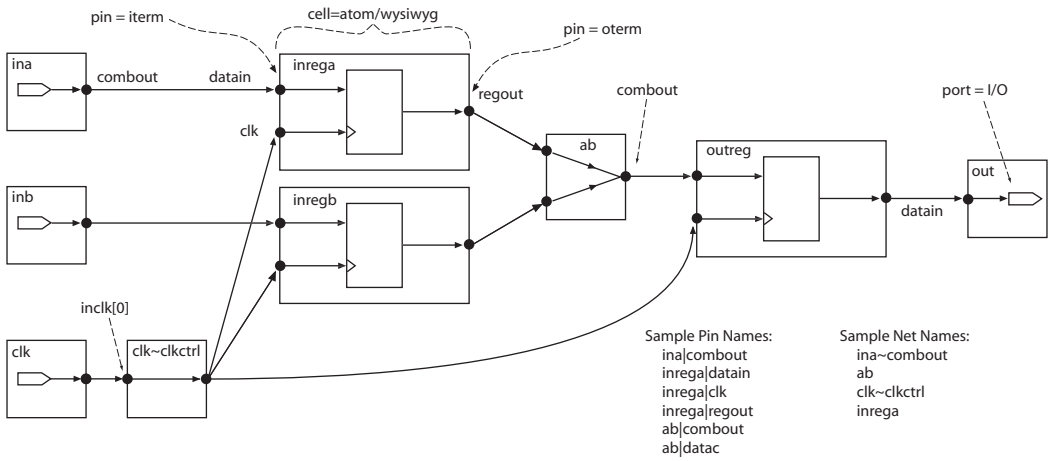


Figure 9–2. Quartus II TimeQuest Timing Analyzer Netlist



Collections

In addition to standard SDC collections, the 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.

For more information about TimeQuest Timing Analyzer terminology, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Constraints

The Quartus II Classic and TimeQuest timing analyzers store constraints in different files, support different methods for constraint entry, and prioritize constraints differently. The following sections explain these differences.

Constraint Files

The TimeQuest timing analyzer stores all SDC timing constraints in **.sdc** files. The Classic timing analyzer stores all timing assignments in the **.qsf** file for your project.

When you use the TimeQuest timing analyzer, your **.qsf** file contains all assignments and settings except for SDC constraints. The TimeQuest timing analyzer ignores the timing assignments in your **.qsf** file except when the conversion utility converts 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. You must manually keep the constraints synchronized, if so desired.

Constraint Entry

With the Classic timing analyzer, you enter timing assignments with the **Settings** dialog box, the Assignment Editor, or with commands in Tcl scripts. With the TimeQuest timing analyzer, 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 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 TimeQuest timing analyzer GUI

You can enter timing assignments for the Classic timing analyzer even if no timing netlist exists for your design. The TimeQuest timing analyzer requires that a netlist exist for interactive constraint entry. Each 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 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 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 TimeQuest timing analyzer before you can enter constraints with either of the following interactive methods:

- At the Tcl console of the TimeQuest timing analyzer
- With commands on the Constraints menu in the TimeQuest timing analyzer GUI

If you enter constraints with a text editor separate from the TimeQuest timing analyzer, no timing netlist is required.

To create a timing netlist in the 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 TimeQuest timing analyzer to enter constraints interactively, you must synthesize your design before you create a timing netlist. To synthesize your design from the command line, 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 point to **Start** on the Processing menu, and click **Start Analysis & Synthesis**.

To create the timing netlist, open the TimeQuest timing analyzer. Then, on the Netlist menu, click **Create Timing Netlist**, select **Post-map** in the **Create Timing Netlist** dialog box, 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 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 after a value is not part of the standard SDC format. Unit specification is a TimeQuest timing analyzer extension.

You can specify clock constraints with period or frequency in the TimeQuest timing analyzer. For example, you can use any 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 function. 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 make changes to the **.sdc** file, you can manually copy the updated MegaCore timing constraints to your **.sdc** file.

Bus Name Format

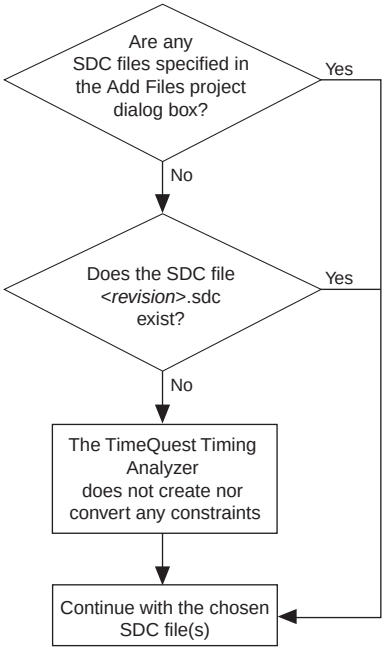
In the Classic timing analyzer, you can make a timing assignment to all bits in a bus with the bus name (or the bus name followed by an asterisk enclosed in square brackets) as the target. For example, to make an assignment to all bits of a bus called address, use `address` or `address[*]` as the target of the assignment.

In the TimeQuest timing analyzer, you must use the bus name followed by square brackets enclosing an asterisk, as follows: `address[*]` to make all timing assignments to all bits within a bus.

Constraint File Priority

The 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 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 TimeQuest timing analyzer reads constraint files from the files list in the order they are listed.

If you use an **.sdc** file created by the conversion utility, you should place it first 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.

The 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. IYou may be prompted 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 9–47 for information about reviewing 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 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 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 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 Classic timing analyzer timing assignments. However, the 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 ordered second.

Example 9–2. Quartus II TimeQuest Timing Analyzer Multicycle Exceptions

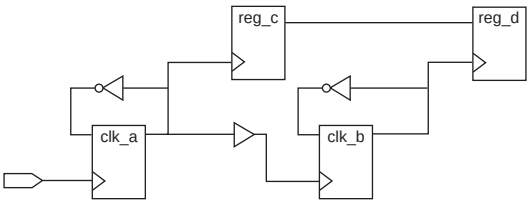
```
set_multicycle_path -from [get_keepers A_reg] -to [get_keepers B*] 2
set_multicycle_path -to [get_keepers B_reg] 3
```

Ambiguous Constraints

Because of capabilities in the 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 TimeQuest timing analyzer.

Figure 9–4. Ripple Clock Circuit



In the 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 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

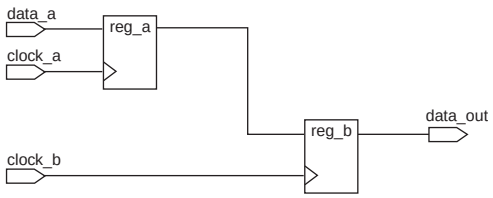
The Quartus II Classic and TimeQuest timing analyzers detect, analyze, and report clocks differently. The following sections describe these differences.

Related and Unrelated Clocks

In the TimeQuest timing analyzer, all clocks are related by default, and you must add assignments to indicate unrelated clocks. However, in the 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.

Figure 9–5 shows a circuit with a path between two clock domains. The TimeQuest timing analyzer analyzes the path from reg_a to reg_b because all clocks are related by default. The 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 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 TimeQuest timing analyzer does not analyze paths between the two clock domains.

```
set_clock_groups -exclusive -group {clock_a} -group {clock_b}
```

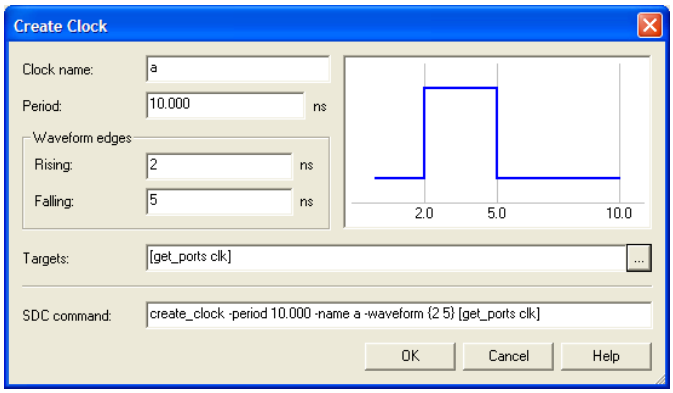
Clock Offset

In the TimeQuest timing analyzer, clocks can have non-zero values for the rising edge of the waveform, a feature that the 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 TimeQuest timing analyzer GUI.

Figure 9–6. Create Clock dialog box



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 use the clock latency constraint, instead of clock offset to emulate latency. Refer to "Offset and Latency Example" on page 9–11 for an example that illustrates the different effects of offset and latency.

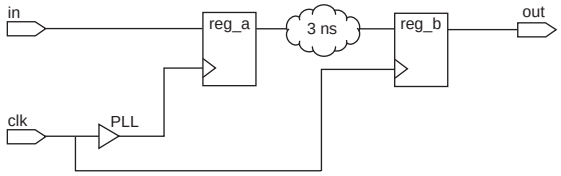
Clock Latency

The TimeQuest timing analyzer does not ignore early clock latency and late clock latency differences when the clock source is the same, as the 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 Classic timing analyzer. This option is in the **More Settings** box of the **Timing Settings** dialog box.

Figure 9–7. Offset and Latency Example Circuit

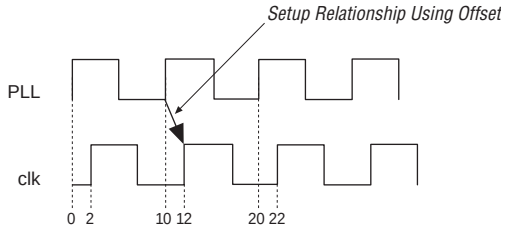


The period for `clk` is 10 ns, and the period for the PLL output is 10 ns. The PLL compensation value is –2 ns. The network delay from the PLL to `reg_a` equals the network delay from `clk` to `reg_b`. Finally, the delay from `reg_a` to `reg_b` is 3 ns.

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.

slack = clock arrival – data arrival

slack = setup relationship + clock skew – reg_to_reg delay

slack = 2ns + 0ns – 3ns

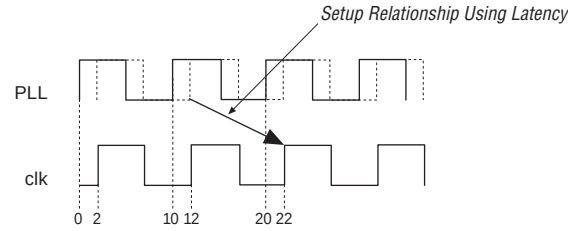
slack = –1ns

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.

slack = clock arrival – data arrival

slack = setup relationship + clock skew – reg_to_reg delay

slack = 10ns + 2ns – 3ns

slack = 9ns

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 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 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 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 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 TimeQuest timing analyzer is 1 ns less than in the Classic timing analyzer because of the way they analyze clock uncertainty.

Figure 9–10. Quartus II Classic Timing Analyzer Behavior

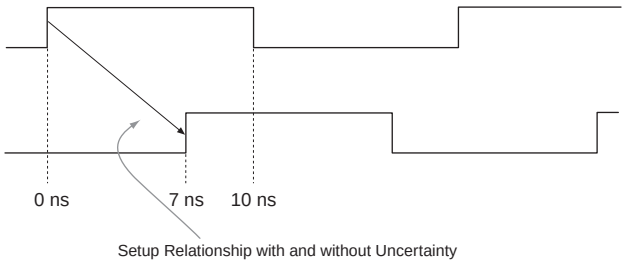
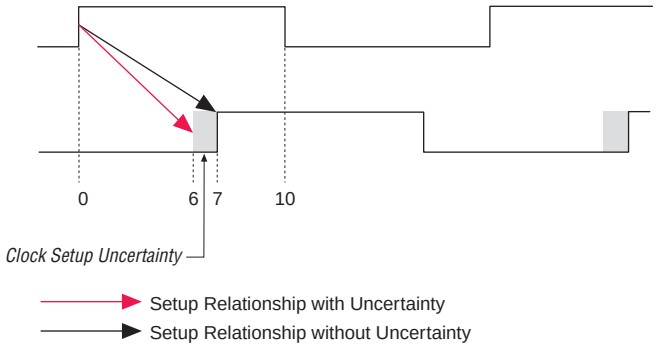


Figure 9–11. Quartus II TimeQuest Timing Analyzer Behavior



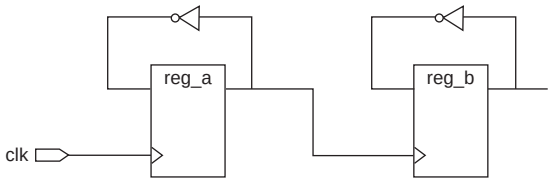
Derived and Generated Clocks

Generated clocks in the TimeQuest timing analyzer are different than derived clocks in the Classic timing analyzer. In the Classic timing analyzer, the source of a derived clock must be a base clock. However, in the 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 TimeQuest timing analyzer, you can specify generated clocks relative to specific edges and edge shifts of a master clock, a feature that the Classic timing analyzer does not support.

Figure 9–12 shows a simple ripple clock that you should constrain with generated clocks in the TimeQuest timing analyzer.

Figure 9–12. Generated Clocks Circuit



The 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 TimeQuest timing analyzers identify clock sources of registers that do not have a defined clock source. The Classic timing analyzer traces back along the clock network, through registers and logic, until it reaches a top-level port in your design. The TimeQuest timing analyzer also traces back along the clock network, but it stops at registers.

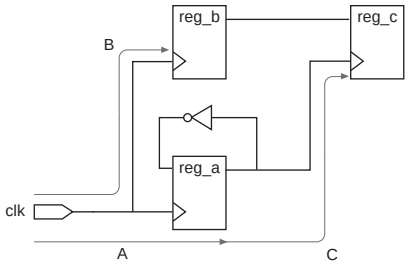
You can use two SDC commands in the TimeQuest timing analyzer to automatically detect and create clocks for unconstrained clock sources:

- **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 circuit with a divide-by-2 register and indicates the clock network delays as A, B, and C. The following explanation describes how the Classic and TimeQuest timing analyzers detect the clocks in Figure 9–13.

Figure 9–13. Example Circuit for derive_clocks



The 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 Classic timing analyzer computes the clock arrival time for `reg_c` as A + C.

The **derive_clocks** command in the 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 TimeQuest timing analyzer compute the same clock arrival time (A + C) as the Classic timing analyzer for `reg_c`, make the following modifications to the clock constraints created by the **derive_clocks** command:

1. Change the base clock named `reg_a` to a generated clock
2. Make the source the clock pin of `reg_a` (`reg_a|clk`) or the port `clk`
3. Add a `-divide_by 2` option

These modifications cause the clock arrival times to `reg_c` to match between the Classic timing analyzer and the 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` option.

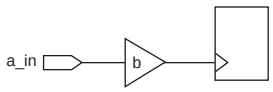
You can use the **derive_clocks** command at the beginning of your design cycle when you do not know all of the clock constraints for your design, but you should not use it during timing sign-off. Instead, you should constrain each clock in your design with the **create_clock** or **create_generated_clocks** 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 circuit shown in Figure 9–14.

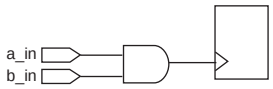
Figure 9–14. Detectable Clock



- If you do not make any clock settings, and then you run **derive_clocks**, it detects `a_in` as a clock according to rule 11, 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 11a, because `a_in` feeds only another clock.

The following examples show how the **derive_clocks** command detects clocks in the circuit shown in Figure 9–15.

Figure 9–15. Two Detectable Clocks



- If you do not make any clock settings and then you run **derive_clocks**, it selects a clock arbitrarily, according to rule 11c.
- 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 11b, because another clock (`a_in`) feeds the register.

derive_pll_clocks Command

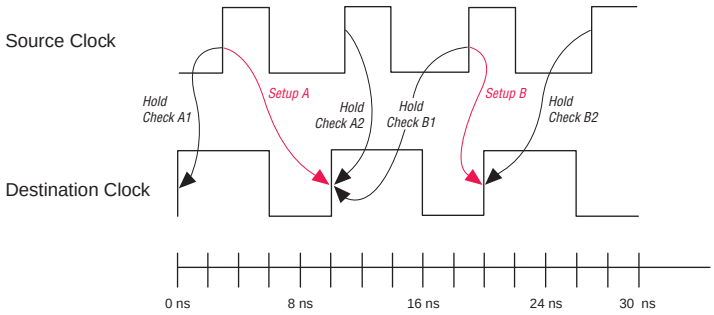
The **derive_pll_clocks** command names the generated clocks according to the names of the PLL output pins by default, and you cannot change these generated clock names. If you want to use your own clock names, you must use the **create_generated_clock** command for each PLL output clock and specify 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 9–10.

Hold Relationship

The Quartus II TimeQuest and 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 Classic timing analyzer first identifies the worst-case setup relationship. The worst-case setup relationship is **Setup B**. Then the Classic timing analyzer chooses the worst-case hold relationship (Hold Check B1 or Hold Check B2) for that specific setup relationship, **Setup B**. The Classic timing analyzer chooses Hold Check B2 for the worst-case hold relationship.

However, the TimeQuest timing analyzer calculates worst-case hold relationships for all possible setup relationships and chooses the absolute worst-case hold relationship. The TimeQuest timing analyzer checks two hold relationships for every setup relationship:

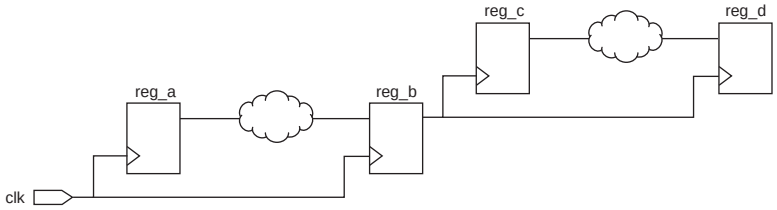
- Data launched by the current launch edge not captured by the previous latch edge (Hold Check A1 and Hold Check B1)
- Data launched by the next launch edge not captured by the current latch edge (Hold Check A2 and Hold Check B2)

The TimeQuest timing analyzer chooses Hold Check A2 as the absolute worst-case hold relationship.

Clock Objects

The 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 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 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 TimeQuest timing analyzer, clocks are abstract objects that are associated with nodes in the timing netlist. The TimeQuest timing analyzer analyzes the path from `reg_a` to `reg_b` even if there is a clock assigned to `reg_b`.

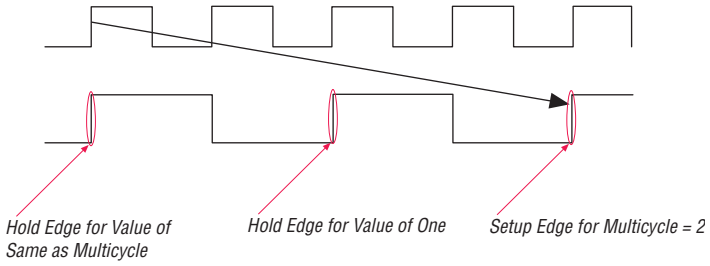
Hold Multicycle

The hold multicycle value numbering scheme is different in the Quartus II Classic and TimeQuest timing analyzers. Also, you can choose between two values for the default hold multicycle value in the Classic timing analyzer but you cannot change the default value in the 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 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 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 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 Classic timing analyzer to compute the equivalent value for the TimeQuest timing analyzer.

In the 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 Classic timing analyzer.

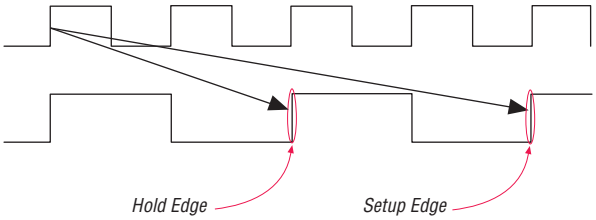
Figure 9–18. Difference Between One and Same As Multicycle



If the default value is **One**, the Classic timing analyzer uses the clock edge one before the setup edge for hold analysis. If the default value is **Same as Multicycle**, the 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 TimeQuest timing analyzer, only a multicycle exception of 2 is required to constrain the design for the indicated setup and hold relationships.

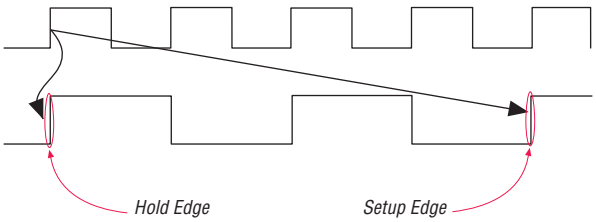
In the Classic timing analyzer, if the **Default Hold Multicycle** value is **One**, only a multicycle assignment of 2 is required to constrain the design.

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 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 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 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 Classic timing analyzer, use the **Default Hold Multicycle** setting.

Fitter Behavior

The behavior for one value of the **Optimize hold time** Fitter assignment differs between the TimeQuest timing analyzer and the Classic timing analyzer. When you set the 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 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 TimeQuest timing analyzer provides a more flexible and powerful interface for reporting timing analysis results than the 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 Classic timing analyzer does not support rise/fall analysis, but the 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.

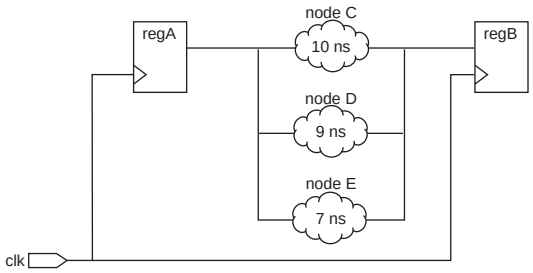
Both analyzers report identical delays along identically constrained paths in your design. The TimeQuest timing analyzer allows you to constrain some paths that you could not constrain with the Classic timing analyzer. Differently constrained paths result in different reported values, but for identical paths in your design that are constrained identically, the delays are exactly the same. Both timing analyzers use the same timing models.

Paths and Pairs

In reporting, the most significant difference between the two analyzers is that the TimeQuest timing analyzer reports paths, while the 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 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 Classic timing analyzer reports only worst-case path `regA` → `C`→ `regB`. The 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 TimeQuest timing analyzer generates only a small number of reports by default, as compared to the Classic timing analyzer, which generates every available report by default. With the TimeQuest timing analyzer, you generate desired reports on demand.

Netlist Names

The Classic timing analyzer uses register names in reporting, but the 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 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 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

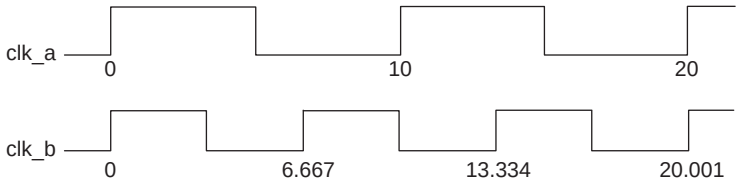
```
Info:      3.788      0.250     uTco  inst
Info:      3.788      0.000 RR  CELL  inst|regout
Info:      4.093      0.305 RR    IC  inst3~feeder|datad
Info:      4.242      0.149 RR  CELL  inst3~feeder|combout
Info:      4.242      0.000 RR    IC  inst3|datain
Info:      4.326      0.084 RR  CELL  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 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 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 situation shown 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 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 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 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
```

Timing Assignment Conversion

This section describes Quartus II Classic QSF timing assignments and their equivalent Quartus II TimeQuest constraints. In some cases, there is more than one potentially equivalent SDC constraint. In these cases, correct conversion requires knowledge of the intended function of your design.

This section includes the following topics:

- "Clock Enable Multicycle" on page 9–28
- "Clock Latency" on page 9–25
- "Clock Uncertainty" on page 9–25
- "Cut Timing Path" on page 9–38
- "Default Required fMAX Assignment" on page 9–26
- "Hold Relationship" on page 9–25
- "Input and Output Delay" on page 9–29
- "Inverted Clock" on page 9–25
- "Maximum Clock Arrival Skew" on page 9–39
- "Maximum Data Arrival Skew" on page 9–39
- "Maximum Delay" on page 9–38
- "Minimum Delay" on page 9–39
- "Minimum tCO Requirement" on page 9–35
- "Minimum tPD Requirement" on page 9–38
- "Multicycle" on page 9–27
- "Not a Clock" on page 9–26
- "Setup Relationship" on page 9–24
- "tCO Requirement" on page 9–33
- "tH Requirement" on page 9–31
- "tPD Requirement" on page 9–37
- "tSU Requirement" on page 9–29
- "Virtual Clock Reference" on page 9–26

Setup Relationship

The **Setup Relationship** assignment overrides the setup relationship between two clocks. By default, the 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 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 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 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 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 the clock latency Classic timing analyzer assignments.

Table 9–1. Quartus II Classic and SDC Equivalent Constraints for Clock Latency Assignments

Quartus II Classic Timing Assignment		SDC Constraint
Assignment Name	**QSF Variable**	
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 TimeQuest timing analyzer, refer to "Clock Latency" on page 9–11.

Clock Uncertainty

This section describes the conversion for the following Classic timing analyzer assignments:

- Clock Setup Uncertainty
- Clock Hold Uncertainty

Table 9–2 shows the equivalent SDC constraints for each of these Classic timing analyzer assignments.

Table 9–2. Quartus II Classic and SDC Equivalent Constraints for Clock Uncertainty Assignments

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 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 Classic timing analyzer analysis. The QSF variable for the **Inverted Clock** assignment is INVERTED_CLOCK. The TimeQuest timing analyzer detects inverted clocks automatically, regardless of the type of inversion circuit, in designs that target device families that support unateness. 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.

Not a Clock

The **Not a Clock** assignment directs the Classic timing analyzer to identify specified node as 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 TimeQuest timing analyzer and there is no equivalent constraint. Appropriate clock constraints are created in the TimeQuest timing analyzer only.

Default Required f_{MAX} Assignment

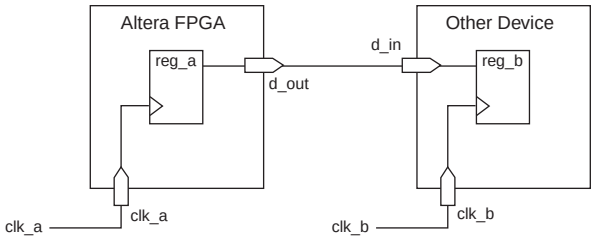
The **Default Required f_{MAX}** assignment allows you to specify an f_{MAX} requirement for the Classic timing analyzer for all unconstrained clocks in your design. The QSF variable for the **Default Required f_{MAX}** assignment is FMAX_REQUIREMENT. You can use the **derive_clocks** command to create 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, instead of the **derive_clocks** command. Refer to "Automatic Clock Detection" on page 9–15 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 TimeQuest timing analyzer supports virtual clocks by default, while the 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 TimeQuest timing analyzer, use the **create_clock** or **create_generated_clock** commands with the `-name` option and no targets.

Figure 9–23 shows a 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 Circuit



Assume the circuit has the following assignments in the 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 Classic timing analyzer includes a variety of assignments to describe clock settings. These include duty cycle, f_{MAX}, offset, and others. In the 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 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 TimeQuest timing analyzers. Refer to "Hold Multicycle" on page 9–18 for more information about the difference between the default value and numbering scheme for the hold multicycle value in the Quartus II Classic and 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 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 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 Classic timing analyzer. However, the 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 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 "tH Requirement" on page 9–31 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 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 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.

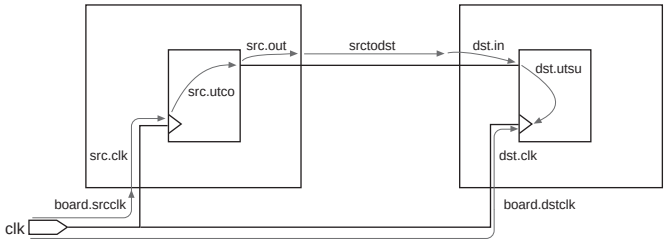
To avoid this warning, your **.sdc** file must specify both the `-max` and `-min` options for each port, or specify neither. If a device I/O in your design includes both the maximum and minimum delay assignments in the Classic timing analyzer, the conversion utility converts both, and no warning appears about that device I/O. If a device I/O has only maximum or minimum delay assignments in the 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.

$$\text{required} - \text{arrival} > 0$$

$$\text{required} = \text{latch} + \text{board.dstclk} + \text{dst.clk} - \text{dst.utsu}$$

$$\text{arrival} = \text{launch} + \text{board.srcclk} + \text{src.clk} + \text{src.utco} + \text{src.out} + \text{srctodst} + \text{dst.in}$$

$$\text{input_delay} = \text{board.srcclk} + \text{src.clk} + \text{src.utcu} + \text{src.out} + \text{srctodst} - \text{board.dstclk}$$

$$\text{required} = \text{latch} + \text{dst.clk} - \text{dst.utsu}$$

$$\text{arrival} = \text{launch} + \text{input_delay} + \text{dst.in}$$

$$(\text{latch} + \text{dst.clk} - \text{dst.utsu}) - (\text{launch} + \text{input_delay} + \text{dst.in}) > 0$$

$$t_{su}\ \text{requirement} - \text{actual}\ t_{su} > 0$$

$$\text{actual}\ t_{su} = \text{dst.in} + \text{dst.utsu} - \text{dst.clk}$$

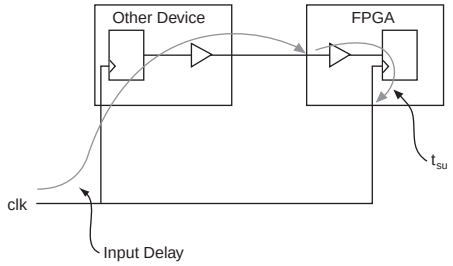
$$t_{su}\ \text{requirement} - (\text{dst.in} + \text{dst.utsu} - \text{dst.clk}) > 0$$

$$t_{su}\ \text{requirement} = \text{latch} - \text{launch} - \text{input_delay}$$

$$\text{input_delay} = \text{latch} - \text{launch} - t_{su}\ \text{requirement}$$

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 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.

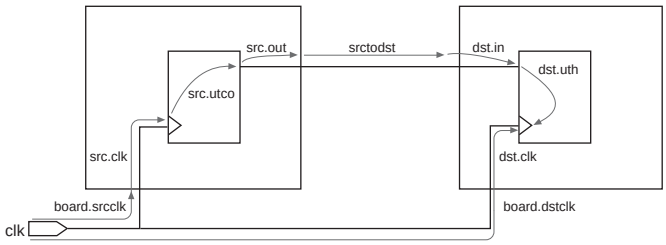
Example 9–10. Converting a Global t_{SU} Assignment to an Equivalent SDC Exception

```
set_max_delay -from [all_inputs] -to [all_registers] <t_SU 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 **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 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 9–29 for additional information.

To convert a global t_H assignment to an equivalent SDC exception, use the command shown in Example 9–11.

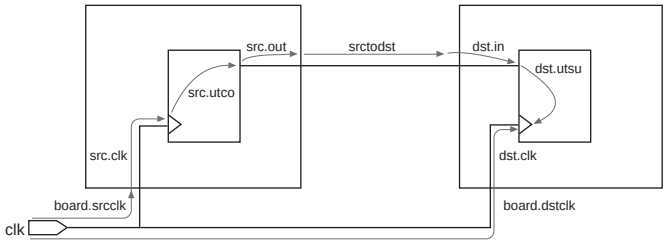
Example 9–11. Converting a Global t_H Assignment to an Equivalent SDC Exception

```
set_min_delay -from [all_inputs] -to [all registers] <negative t_H 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



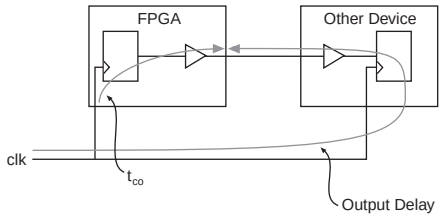
Equation 9–7 shows the derivation of this conversion.

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 Classic timing analyzer, and the corresponding options for the **set_max_delay** exception.

Table 9–8. t_{CO} Requirement and set_max_delay Equivalence (Part 1 of 2)

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 *]

Table 9–8. t_{CO} Requirement and set_max_delay Equivalence (Part 2 of 2)

t_{CO} Requirement Options	set_max_delay Options
-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.

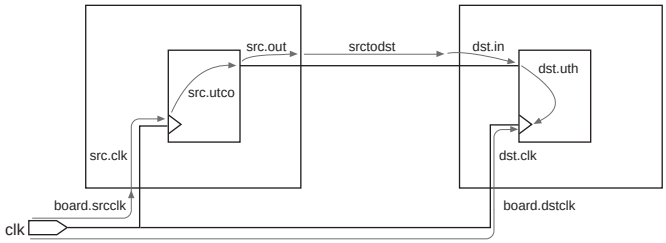
Example 9–12. Converting a Global t_{CO} Assignment to an Equivalent SDC Exception

```
set_max_delay -from [all registers] -to [all_outputs] <t_CO 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.

$$\text{arrival} + \text{required} > 0$$
$$\text{arrival} = \text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$\text{required} = \text{latch} - \text{output_delay}$$
$$\text{output_delay} = \text{srctodst} + \text{dst.in} - \text{dst.uth} - \text{dst.clk} - \text{board.dstclk} + \text{board.srcclk}$$
$$(\text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}) - (\text{latch} - \text{output_delay}) > 0$$
$$\text{minimum } t_{co} - \text{minimum } t_{co}\text{requirement} > 0$$
$$\text{minimum } t_{co} = \text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$(\text{launch} + \text{src.clk} + \text{src.utco} + \text{src.out}) - \text{minimum } t_{co} \text{ requirement} > 0$$
$$\text{minimum } t_{co} \text{ requirement} = \text{latch} - \text{launch} - \text{output_delay}$$
$$\text{output_delay} = \text{latch} - \text{launch} - \text{minimum } t_{co} \text{ requirement}$$

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.

$$\text{arrival} - \text{required} > 0$$
$$\text{arrival} = \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$\text{required} = \text{min_delay}$$
$$(\text{src.clk} + \text{src.utco} + \text{src.out}) - (\text{set_min_delay}) > 0$$
$$\text{minimum } t_{co} - \text{minimum } t_{co} \text{ requirement} > 0$$
$$\text{minimum } t_{co} = \text{src.clk} + \text{src.utco} + \text{src.out}$$
$$(\text{src.clk} + \text{src.utco} + \text{src.out}) - \text{minimum } t_{co}\text{requirement} > 0$$
$$\text{set_min_delay} = \text{minimum } t_{co} \text{ requirement}$$

Table 9–9 shows the different ways you can make minimum t_{CO} assignments in the 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>
```

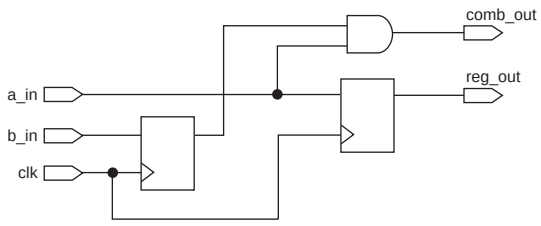
t_{PD} Requirement

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 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 circuit to illustrate the following example of converting a **t_{PD} Requirement** to a **set_max_delay** constraint.

Figure 9–30. t_{PD} Example



Assume the circuit has the following assignments in the 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 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 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 9–37 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 Classic timing analyzer is equivalent to the **set_false_path** command in the 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 TimeQuest timing analyzer.

The **Maximum Delay** assignment for the Classic timing analyzer is not related to the **set_max_delay** exception in the 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 TimeQuest timing analyzer.

The **Minimum Delay** assignment for the Classic timing analyzer is not related to the **set_min_delay** exception in the TimeQuest timing analyzer.

Maximum Clock Arrival Skew

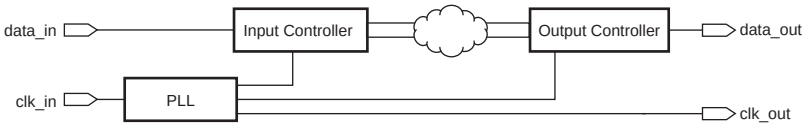
The **Maximum Clock Arrival Skew** assignment specifies the maximum clock skew between a set of registers. The QSF variable for the **Maximum Clock Arrival Skew** assignment is MAX_CLOCK_ARRIVAL_SKEW. In the Classic timing analyzer, this assignment is specified between a clock node name and a set of registers. **Maximum Clock Arrival Skew** is not supported in the 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 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

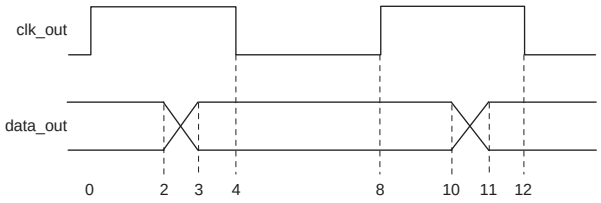


Constraining Skew on an Output Bus

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 `clk_in` and `clk_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.

latch – launch = 0ns

output delay = latch – launch – 2ns

output delay = –2ns

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.

latch – launch = 8ns

output delay = latch – launch – 3ns

output delay = 5ns

Refer to "I/O Constraints" on page 9–28 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 clk_in \
             [get_ports data_out*]
derive_pll_clocks
set_output_delay -add_delay \
    -clock ddr_pll_1_inst|altpll_component|pll|CLK[0] \
    -reference_pin [get_ports 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 [get_ports clk_out] \
    -max [expr $period - 3.000] \
    [get_ports data_out*]
```

Conversion Utility

The 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 9–2 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 9–29
t_{H} Requirement	TH_REQUIREMENT	page 9–31
t_{CO} Requirement	TCO_REQUIREMENT	page 9–33
Minimum t_{CO} Requirement	MIN_TCO_REQUIREMENT	page 9–35
t_{PD} Requirement	TPD_REQUIREMENT	page 9–37
Minimum t_{PD} Requirement	MIN_TPD_REQUIREMENT	page 9–38

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 Classic timing analyzer with the specified values corresponding to the behavior of the TimeQuest timing analyzer.

- **Cut off clear and preset signal paths**— The TimeQuest timing analyzer does not support this functionality. You should use Recovery and Removal analysis instead to analyze register control paths. The Classic timing analyzer does not support this option.
- **Cut off feedback from I/O pins**—The TimeQuest timing analyzer does not match the functionality of the Classic timing analyzer when this assignment is OFF.
- **Cut off read during write signal paths**—The TimeQuest timing analyzer does not match the functionality of the Classic timing analyzer when this assignment is OFF.
- **Analyze latches as synchronous elements**—The TimeQuest timing analyzer analyzes latches as synchronous elements by default and does not match the functionality of the Classic timing analyzer when this assignment is OFF. The Classic timing analyzer analyzes latches as synchronous elements by default.
- **Enable Clock Latency**—The TimeQuest timing analyzer includes clock latency in its calculations. The TimeQuest timing analyzer does not match the functionality of the 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 9–11 for an example of the different effects of offset and latency. When you turn on **Enable Clock Latency** in the 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 TimeQuest timing analyzer because they do not include the entity name when this option is ON.

If your design meets timing requirements in the Classic timing analyzer without all of the settings recommended in Table 9–11 on page 9–41, you should perform one of the following actions:

- Change the settings and re-constrain and reverify as necessary.
  or
- Add or modify SDC constraints as appropriate because analysis in the 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_net_name** command in the **.sdc** file it creates. The `-use_net_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 Classic timing analyzer.

Including the `-use_net_name` option ensures that the conversion utility converts clock-to-clock exceptions properly. If you remove the `-use_net_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 9–15 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 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 9–11 for information about how to use offset values in the 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 (Part 1 of 2)

Assignment Name	QSF Variable	More Information
Late Clock Latency	LATE_CLOCK_LATENCY	page 9–25
Early Clock Latency	EARLY_CLOCK_LATENCY	
Clock Setup Uncertainty	CLOCK_SETUP_UNCERTAINTY	page 9–25
Clock Hold Uncertainty	CLOCK_HOLD_UNCERTAINTY	
Multicycle *(1)*	MULTICYCLE	page 9–27
Source Multicycle *(2)*	SRC_MULTICYCLE	
Multicycle Hold *(3)*	HOLD_MULTICYCLE	
Source Multicycle Hold	SRC_HOLD_MULTICYCLE	
Clock Enable Multicycle	CLOCK_ENABLE_MULTICYCLE	page 9–28
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	

Table 9–12. Instance Timing Assignments (Part 2 of 2)

Assignment Name	QSF Variable	More Information
Cut Timing Path	CUT	page 9–38
Input Maximum Delay	INPUT_MAX_DELAY	page 9–29
Input Minimum Delay	INPUT_MIN_DELAY	
Output Maximum Delay	OUTPUT_MAX_DELAY	
Output Minimum Delay	OUTPUT_MIN_DELAY	

Notes to Table 9–1 2:

(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 9–29 for an explanation of why the warning occurs and how to avoid it.

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 9–18 for more information on hold multicycle differences between the Quartus II Classic and 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 9–37
Minimum t_{PD} Requirement *(1)*	MIN_TPD_REQUIREMENT	page 9–38
Setup Relationship	SETUP_RELATIONSHIP	page 9–24
Hold Relationship	HOLD_RELATIONSHIP	page 9–25

Note to Table 9–1 3:

(1) Refer to "tPD and Minimum tPD Requirement Conversion" on page 9–52 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 9–29
t_H Requirement *(1)*	set_min_delay	set_input_delay -min	page 9–31
t_{CO} Requirement *(2)*	set_max_delay	set_output_delay -max	page 9–33
Minimum t_{CO} Requirement *(2)*	set_min_delay	set_output_delay -min	page 9–35

Notes to Table 9–1 4:

(1) Refer to "tPD and Minimum tPD Requirement Conversion" on page 9–52 for more information about how the conversion utility converts this type of assignment.

(2) Refer to "tCO Requirement Conversion" on page 9–46 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 formula in Equation 9–13:

Equation 9–13.

$$\text{<correct value>} = \text{<converted value>} - \frac{(\text{<pll output period>} \times \text{<phase shift>})}{360}$$

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 equation:

$$1.0 - \frac{(5 \times 90)}{360} = -0.25$$

Then, change the value so the SDC command so that it 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]] 5.0
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 9–27
Source Multicycle	SRC_MULTICYCLE	
Multicycle Hold	HOLD_MULTICYCLE	
Source Multicycle Hold	SRC_HOLD_MULTICYCLE	
Setup Relationship	SETUP_RELATIONSHIP	page 9–24
Hold Relationship	HOLD_RELATIONSHIP	page 9–25
Cut Timing Path	CUT	page 9–38

You must manually convert all other entity-specific timing assignments.

Paths Between Unrelated Clock Domains

The conversion utility can create exceptions that cut paths between unrelated clock domains, which matches the default behavior of the Classic timing analyzer. When **Cut paths between unrelated clock domains** is turned 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 9–25
Maximum Clock Arrival Skew	MAX_CLOCK_ARRIVAL_SKEW	page 9–39
Maximum Data Arrival Skew	MAX_DATA_ARRIVAL_SKEW	page 9–39
Maximum Delay	MAX_DELAY	page 9–38
Minimum Delay	MIN_DELAY	page 9–39
Virtual Clock Reference	VIRTUAL_CLOCK_REFERENCE	page 9–26

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 Classic timing analyzer and 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 9–7 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 9–24 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 9–41 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 TimeQuest timing analyzer is closest to the 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 TimeQuest timing analyzer analysis may be different from the Classic timing analyzer analysis. Refer to "Recommended Global Assignments" on page 9–41 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 fMAX Assignment" on page 9–26 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 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 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 9–46 lists the entity-specific assignments the script can convert automatically.

Refer to "Timing Assignment Conversion" on page 9–24 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 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 9–11 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 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 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 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 TimeQuest timing analyzer. You must cut unrelated clock domains. Refer to "Related and Unrelated Clocks" on page 9–10 for information about how to cut unrelated clock domains.

Path Details

If you have unexpected differences between the Quartus II Classic and TimeQuest timing analyzers on some paths, follow these steps to identify the cause of the difference.

1. List the path in the Classic timing analyzer.
2. Report timing on the path in the TimeQuest timing analyzer.
3. Compare slack values.
4. Compare source and destination clocks.
5. Compare the launch/latch times in the TimeQuest timing analyzer to the setup/hold relationship in the Classic timing analyzer. The times are absolute in the TimeQuest timing analyzer and relative in the 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 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 Classic timing analyzer, but there are unconstrained paths in the TimeQuest timing analyzer, some assignments may not have been converted properly. Also, some of the assignments could be ambiguous. The TimeQuest timing analyzer analyzes more paths than the Classic timing analyzer does, so any unconstrained paths might be paths you could not constrain in the Classic timing analyzer.

Bus Names

If your design includes 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 9–7 for more information.

Other

Review the notes listed in "Conversion Utility" on page 9–52.

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 9–7. Any method described in "Conversion Utility" on page 9–2 forces the conversion utility to run even if it can find an **.sdc** file.

Notes

This section describes notes for the TimeQuest timing analyzer.

Output Pin Load Assignments

The 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 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

The 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 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 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.

Unsupported SDC Features

Some SDC commands and features are not supported by the current version of the 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 and Optimization

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 TimeQuest timing analyzer, you must update the SDC constraints. You can use the conversion utility, or update the **.sdc** file manually.

If you use physical synthesis with the TimeQuest timing analyzer, the design may have lower performance.

Clock Network Delay Reporting

In the Quartus II software version 6.0, the TimeQuest timing analyzer reports delay on the clock network as a single number, rather than node-to-node segments, as the 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.

Project Management

If you use the **project_open** Tcl command in the TimeQuest timing analyzer to open a project compiled with an earlier version of the Quartus II software, the 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.

For more detailed information about the features and capabilities of the TimeQuest timing analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Document Revision History

Table 9–17. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	Minor updates to content.
November 2009	9.1.0	No change to content.
March 2009	9.0.0	This was chapter 8 in version 8.1.
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	■ 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.

Take an online survey to provide feedback about this handbook chapter.

10. Quartus II Classic Timing Analyzer

QII53004-10.0.0

This chapter details the aspects of timing analysis using the Quartus® II Classic Timing Analyzer.

Static timing analysis is a method for analyzing, debugging, and validating the timing performance of a design. Static timing analysis, used in conjunction with functional simulation, allows you to verify overall design operation. As part of the compilation flow, the Classic Timing Analyzer automatically performs a static timing analysis so you do not have to use a third-party timing analysis tool. The Quartus II software also includes the TimeQuest Timing Analyzer, which provides advanced constraints and analysis capabilities that cater to all designs. Like the Classic Timing Analyzer, the TimeQuest Timing Analyzer can also be an automatic part of your compilation flow, and should be used for all new designs.

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*.

This chapter assumes you have some Tcl expertise; Tcl commands are used throughout this chapter to describe the alternative methods for making timing analysis assignments. For more information about GUI-equivalent timing constraints, refer to "Timing Analysis Using the Quartus II GUI" on page 10–35.

New device families are not supported in the Classic Timing Analyzer. Altera recommends using the TimeQuest Timing Analyzer for all new designs.

This chapter discusses the following topics:

- "Timing Analysis Tool Setup" on page 10–2
- "Static Timing Analysis Overview" on page 10–2
- "Clock Settings" on page 10–8
- "Clock Types" on page 10–9
- "Clock Uncertainty" on page 10–10
- "Clock Latency" on page 10–11
- "Timing Exceptions" on page 10–13
- "I/O Analysis" on page 10–22
- "Asynchronous Paths" on page 10–25
- "Skew Management" on page 10–29
- "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 10–35
- "Scripting Support" on page 10–39

■ "MAX+PLUS II Timing Analysis Methodology" on page 10–43

Timing Analysis Tool Setup

The Quartus II software includes two static timing analysis tools:

- Classic Timing Analyzer
- TimeQuest Timing Analyzer

Use the **Timing Analysis** option under the Settings menu to set the timing analyzer that is used in the compilation process.

Altera recommends using the TimeQuest Timing Analyzer for new designs. The TimeQuest Timing Analyzer provides advanced constraints and analysis capabilities that cater to all designs.

To use the Classic Timing Analyzer, follow these steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, expand **Timing Analysis Settings** and select **Use Classic Timing Analyzer during compilation**.

For more information about the advanced features available in the TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Static Timing Analysis Overview

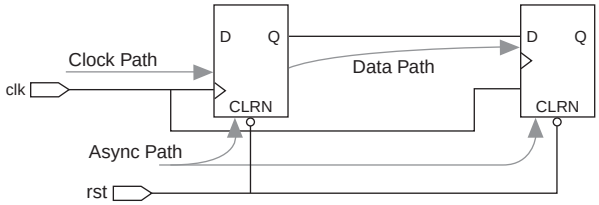
This section provides information about the static timing analysis concepts. A complete understanding of the concepts presented in this section allows you to take advantage of the static timing analysis features available in the Quartus II software.

Various paths exist in 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 shows the commonly analyzed paths types.

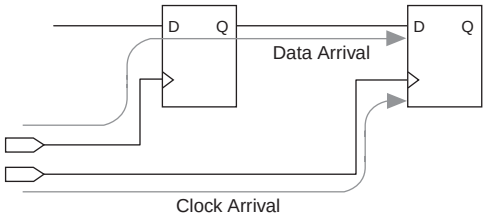
Figure 10–1. Path Types



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 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

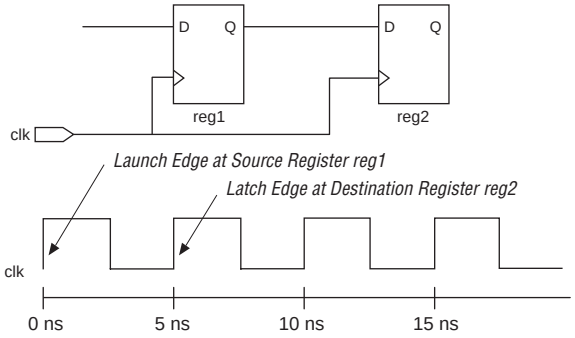


In addition to identifying various paths within a design, the Classic Timing Analyzer analyzes clock characteristics to compute the worst-case requirement between any two registers in a single register-to-register path. You must use timing constraints to specify the characteristics of all clock signals 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 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 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 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 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 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 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 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.

$$\text{Data Arrival Time} = \text{Launch Edge} + \text{Longest Clock Path to Source Register} + \text{micro } t_{CO} + \text{Longest Data Delay}$$

The 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.

$$\text{Largest Register-to-Register Requirement} = \text{Setup Relationship between Source and Destination} + \text{largest clock skew} - \text{micro } t_{CO} \text{ of Source Register} - \text{micro } t_{SU} \text{ of Destination Register}$$

Equation 10–8.

$$\text{Setup Relationship between Source \& Destination Register} = \text{Latch Edge} - \text{Launch Edge Setup Uncertainty}$$

Equation 10–9.

$$\text{Largest Clock Skew} = \text{Shortest Clock Path to Destination Register} - \text{Longest Clock Path to Source Register}$$

Both sets of equations are used to determine the slack value of any path.

Clock Hold Check

To prevent hold violations, the 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 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 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 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 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. For more information about multicycle exceptions, refer to "Multicycle" on page 10–14.

Figure 10–4. Example Diagram of a Multicycle Path

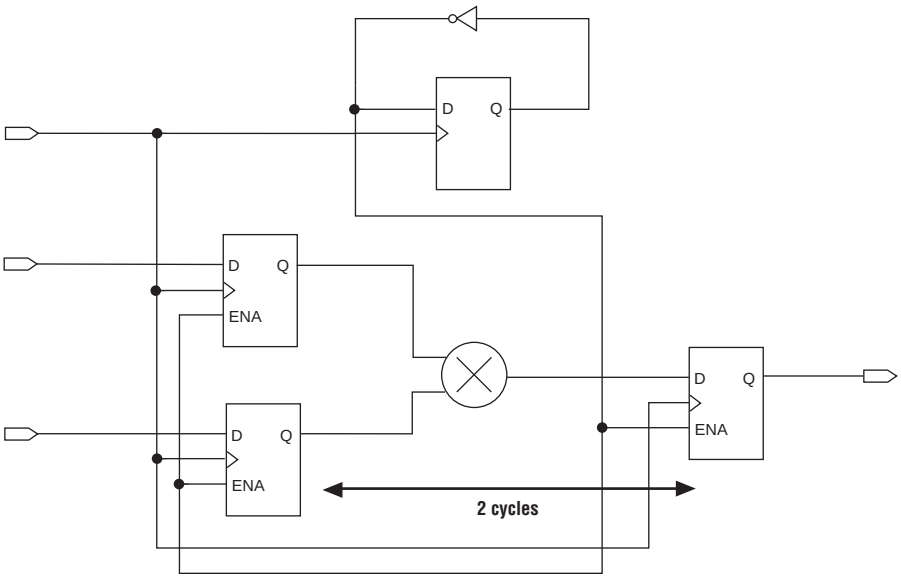


Figure 10–5 shows the default clock setup analysis launch and latch edges where multicycle assignment is equal to 1.

Figure 10–5. Default Clock Setup Analysis

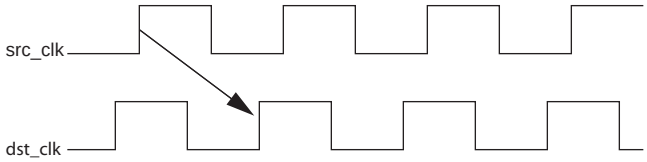
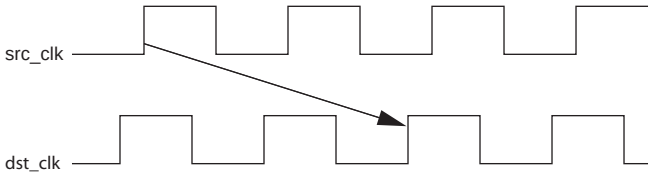


Figure 10–6 shows an analysis similar to Figure 10–5, but with a multicycle of 2.

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 that the Quartus II Fitter achieves the desired performance requirements and the 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 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 Classic Timing Analyzer reports actual data delay for undefined clocks, but because no clock requirements exist for undefined clocks, the 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 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 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 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 Classic Timing Analyzer.

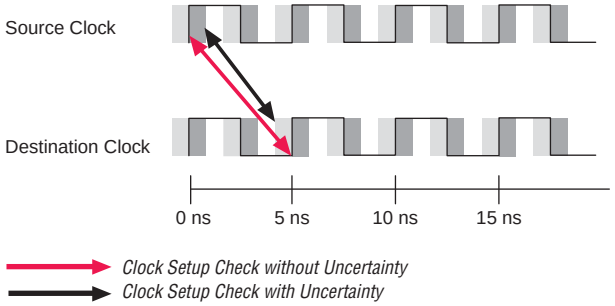
The ALTPLL megafunction is supported in only Stratix I, Stratix II, and Stratix III of the Stratix device family.

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

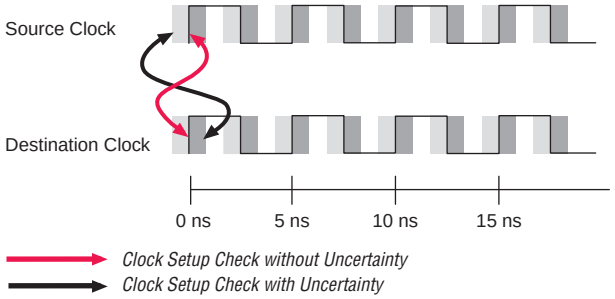


You can create clock uncertainty assignments using the Tcl command `set_clock_uncertainty`. The `set_clock_uncertainty` assignment used with the switch `-setup` specifies a clock setup uncertainty assignment. The following example creates a **Clock Setup Uncertainty** assignment with a value of 2 ns 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 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

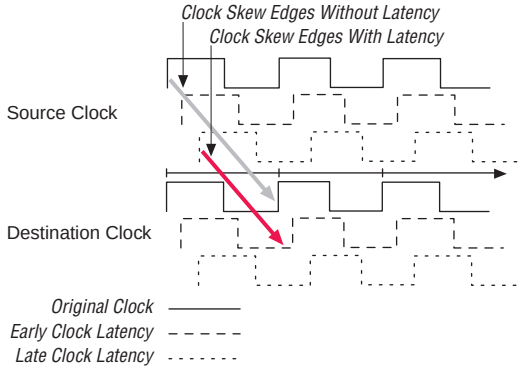
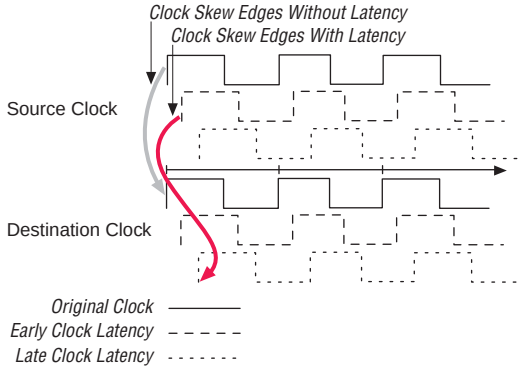


Figure 10–10 shows the clock edges used to calculate clock skew for a hold check when the **Early Clock Latency** and **Late Clock Latency** assignments are used.

Figure 10–10. Clock Hold Check Clock Skew



The 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 Classic Timing Analyzer to analyze clock latency. When this option is set to **ON**, the 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**, use the following Tcl command:

```
set_global_assignment -name ENABLE_CLOCK_LATENCY ON
```

When the **Enable Clock Latency** option is turned on, the 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 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

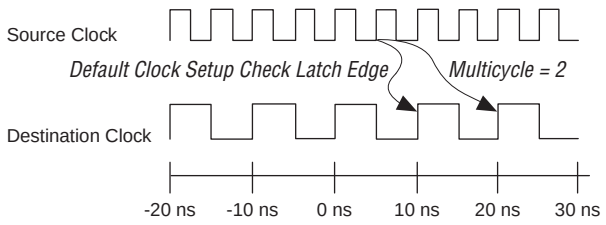
By default, the Classic Timing Analyzer performs a single-cycle analysis for all valid register-to-register paths in the design. Multicycle assignments specify the number of clock periods before a source register launches the data or a destination register latches the data. Multicycle assignments adjust 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_assingment`, 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:

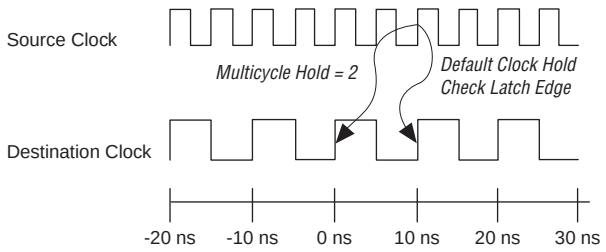
```
set_multicycle_assignment -setup -end -from reg1 -to reg2 2
```

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.

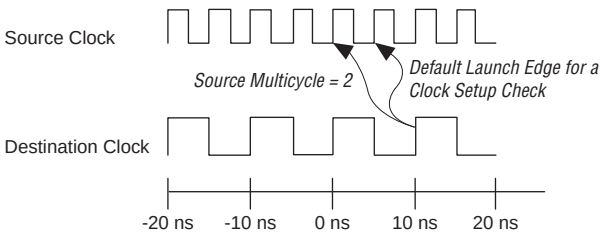
If your design requires a hold multicycle value not equal to the setup multicycle or 1, you must explicitly apply a hold multicycle assignment to the path.

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.

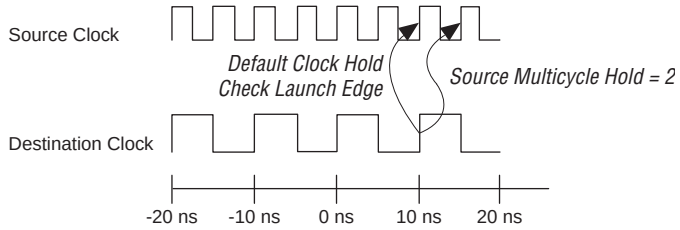
If your design requires a hold multicycle value not equal to the setup multicycle or 1, you must explicitly apply a hold multicycle assignment to the path.

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 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.

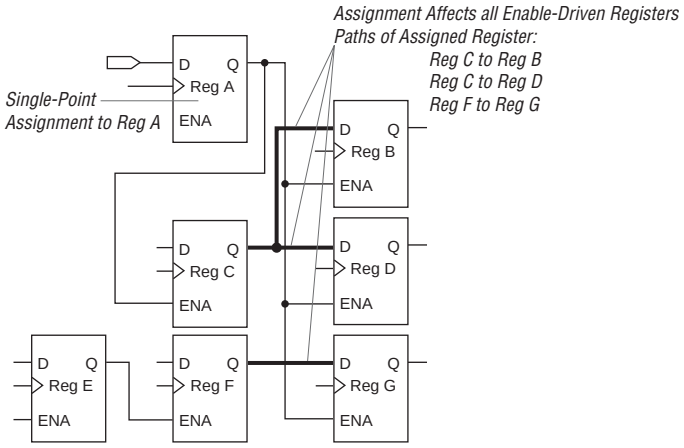
The **Clock Enable Multicycle** modifies the latching edge when a clock setup check is performed for all registers driven by the specified clock enables, and the **Clock Enable Multicycle Hold** modifies the latching edge when a clock hold check is performed for all registers driven by the specified 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

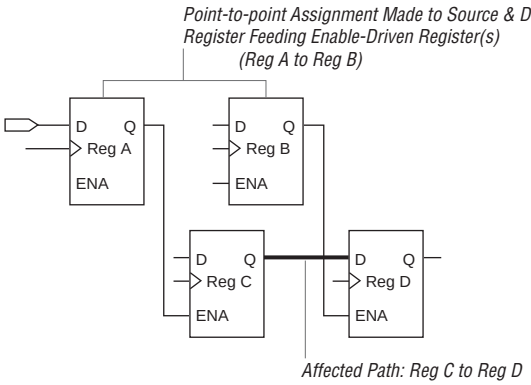
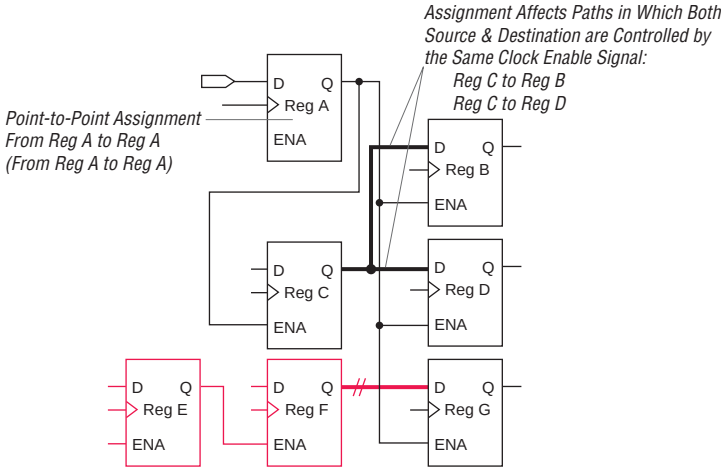


Figure 10–17 shows an example of a point-to-point assignment made to the same source and destination register. In this example, register `Reg A` has been specified as both the source and register for the assignment. Only register-to-register paths that have both the `source-enable` port and `destination-enable` 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 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 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 the operation of a circuit, 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.

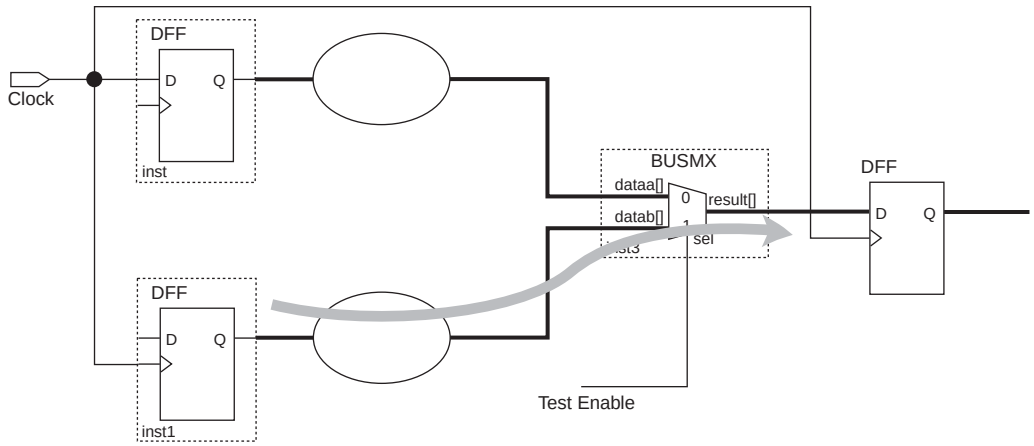
You can use the `set_global_assignment -name CUT_OFF_READ_DURING_WRITE_PATHS ON` Tcl command to cut the path from the `write-enable` register through memory element to a destination register.

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 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 Classic Timing Analyzer can perform a full system analysis. By providing **Input Delays** and **Output Delays**, the 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 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 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.

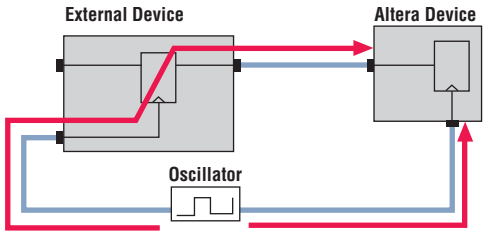
The **Input Minimum Delay** defaults to the **Input Maximum Delay** and the **Input Maximum Delay** defaults to the Input **Minimum Delay** if only one is specified.

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 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 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 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.

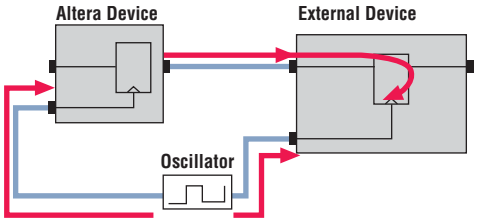
The Output Minimum Delay default value is the same as the Output Maximum Delay, and the Output Maximum Delay default value is the same as the Output Minimum Delay if only one is specified.

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 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 Classic Timing Analyzer can analyze asynchronous signals that connect to the clear, preset, or load ports of a register. This section explains how the 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 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 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 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 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 Classic Timing Analyzer to perform recovery analysis on that path.

Removal Report

When you set ENABLE_RECOVERY_REMOVAL_ANALYSIS to **ON**, the 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 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 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 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 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 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 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 Classic Timing Analyzer provides two assignments for analyzing and constraining skew for data and clock signals.

Maximum Clock Arrival Skew

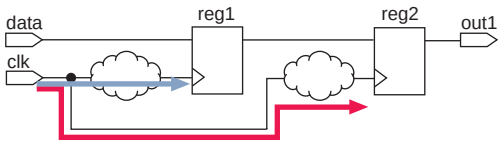
Make **Maximum Clock Arrival Skew** assignments to specify the maximum allowable clock arrival skew between a clock signal and various destination registers. The Classic Timing Analyzer compares the longest clock path to the registers' clock port and the shortest clock path to the registers' clock port 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 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

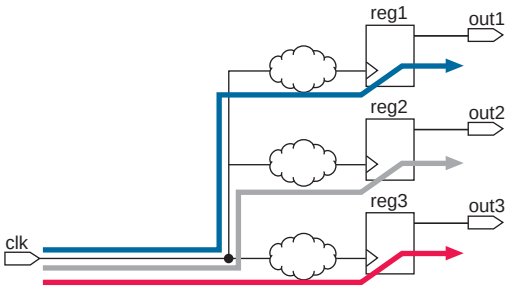
Make **Maximum Data Arrival Skew** assignments to specify the maximum allowable data arrival skew to various destination registers or pins. The Classic Timing Analyzer compares the longest data arrival path to the shortest data arrival path to determine if your specified maximum data arrival skew is 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 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 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 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 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 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.

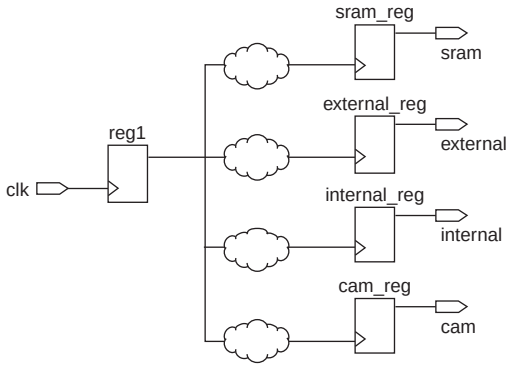
Use the `timegroup` Tcl command to create an assignment group. The following example creates an assignment group `srcgrp` and adds nodes with names that match `src1*` to the group:

```
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 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
```

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 Chip Planner. 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 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 Classic Timing Analyzer can analyze these latches as synchronous elements rather than as combinational elements. The clock enables are analyzed as inverted clocks. The 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

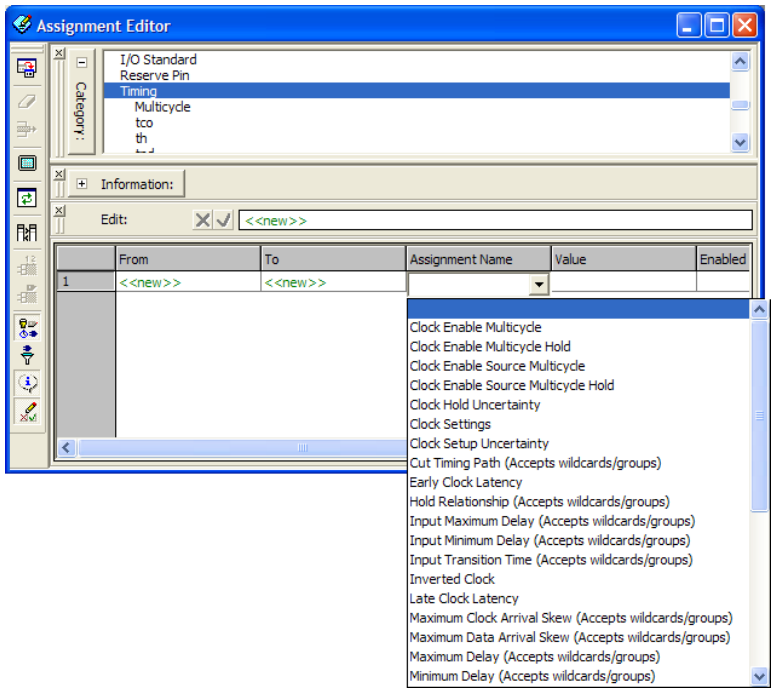
In addition to the extensive scripting support available in the Classic Timing Analyzer, the Quartus II software provides the Assignment Editor and other user interface tools, giving you access to the Classic Timing Analyzer features and assignments.

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, expand **Timing Analysis Settings**. (Ensure 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, expand **Timing Analysis Settings**. (Ensure that the **Use Classic Timing Analyzer during compilation** radio button is turned on.) Click **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 Classic Timing Analyzer report is a section of the Compilation Report containing the static timing analysis results. The 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.

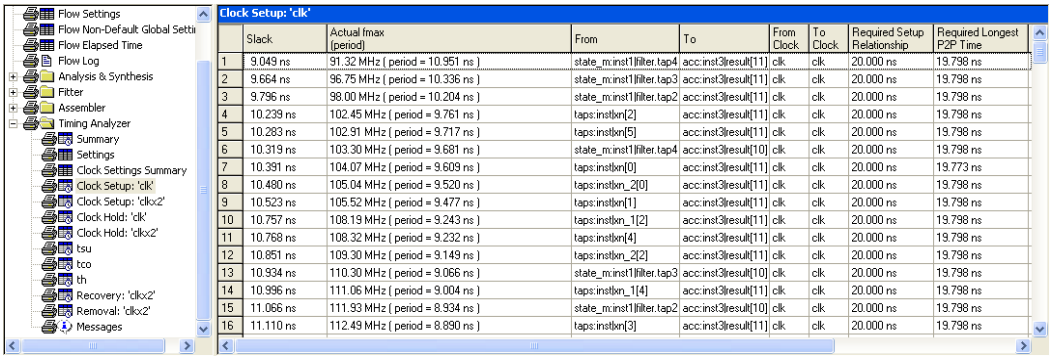
In the **Settings** dialog box, you can specify the range of information to be reported in the timing analysis of the Compilation Report. To access this page, on the Assignments menu, click **Settings**. In the **Category** list, expand **Timing Analysis Settings**. (Ensure that the **Use Classic Timing Analyzer during compilation** radio button is turned on.) Expand the **Classic Timing Analyzer Settings** list. Click **Classic Timing Analyzer Reporting**. The **Classic Timing Analyzer Reporting** dialog box appears.

If there are no timing assignments for the design, the Classic Timing Analyzer does not generate slack reports for any detected clock nodes. The 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



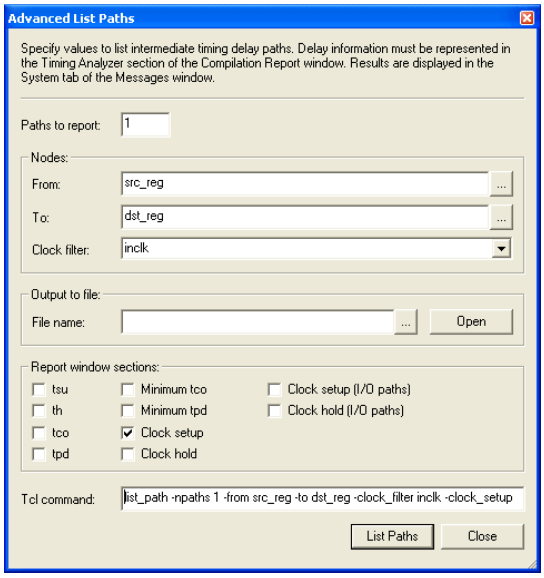
	Slack	Actual fmax (period)	From	To	From Clock	To Clock	Required Setup Relationship	Required Longest P2P Time
1	9.049 ns	91.32 MHz (period = 10.951 ns)	state_minst1|filter.tap4	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
2	9.664 ns	96.75 MHz (period = 10.336 ns)	state_minst1|filter.tap3	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
3	9.796 ns	98.00 MHz (period = 10.204 ns)	state_minst1|filter.tap2	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
4	10.239 ns	102.45 MHz (period = 9.761 ns)	taps:inst|xn[2]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
5	10.283 ns	102.91 MHz (period = 9.717 ns)	taps:inst|xn[5]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
6	10.319 ns	103.30 MHz (period = 9.681 ns)	state_minst1|filter.tap4	acc:inst3|result[10]	clk	clk	20.000 ns	19.798 ns
7	10.391 ns	104.07 MHz (period = 9.609 ns)	taps:inst|xn[0]	acc:inst3|result[11]	clk	clk	20.000 ns	19.773 ns
8	10.480 ns	105.04 MHz (period = 9.520 ns)	taps:inst|xn_2[0]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
9	10.523 ns	105.52 MHz (period = 9.477 ns)	taps:inst|xn[1]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
10	10.757 ns	108.19 MHz (period = 9.243 ns)	taps:inst|xn_1[2]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
11	10.768 ns	108.32 MHz (period = 9.232 ns)	taps:inst|xn[4]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
12	10.851 ns	109.30 MHz (period = 9.149 ns)	taps:inst|xn_2[2]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
13	10.934 ns	110.30 MHz (period = 9.066 ns)	state_minst1|filter.tap3	acc:inst3|result[10]	clk	clk	20.000 ns	19.798 ns
14	10.996 ns	111.06 MHz (period = 9.004 ns)	taps:inst|xn_1[4]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns
15	11.066 ns	111.93 MHz (period = 8.934 ns)	state_minst1|filter.tap2	acc:inst3|result[10]	clk	clk	20.000 ns	19.798 ns
16	11.110 ns	112.49 MHz (period = 8.890 ns)	taps:inst|xn[3]	acc:inst3|result[11]	clk	clk	20.000 ns	19.798 ns

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

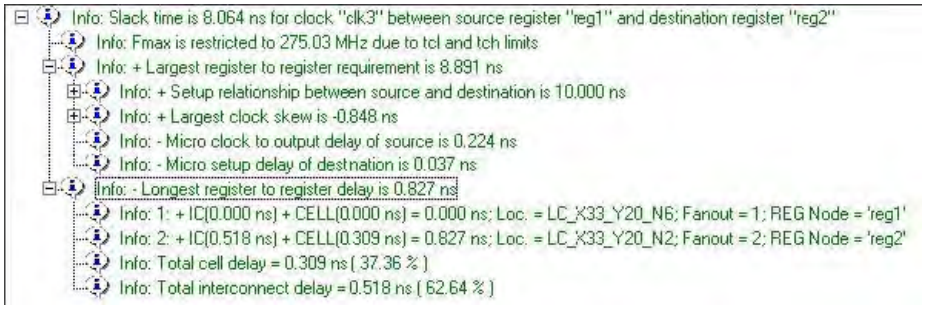


You can perform a list path command directly from the Timing Analysis report. To do this, right click a path and click **List Path** (Figure 10–27). To launch the **Advanced List Paths** dialog box, right-click a path and in the menu that appears, and select **Advanced List Paths**.

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 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 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



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 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*.

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 10–8 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.

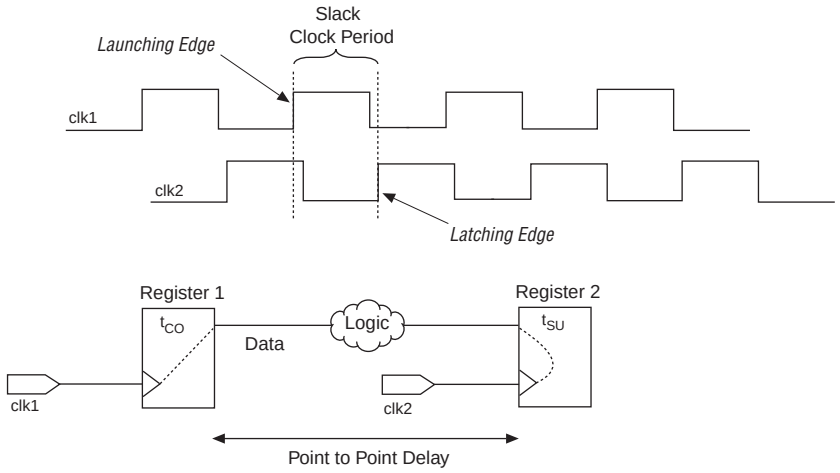
$$\text{Clock Hold Slack} = \text{Shortest Register-to-Register Delay} - \text{Smallest Register-to-Register Requirement}$$

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

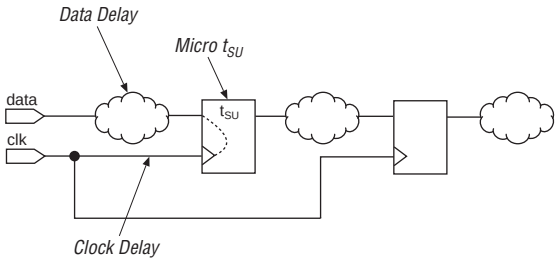
This section describes the basic measurements made for I/O timing in the Quartus II software.

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.

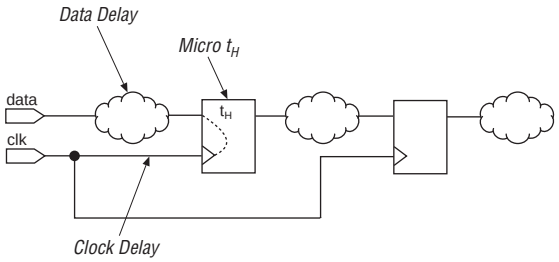
Equation 10–39.

$$t_{SU} = \text{Longest Data Delay} - \text{Shortest Clock Delay} + \text{micro } t_{SU} \text{ of Input Register}$$

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.

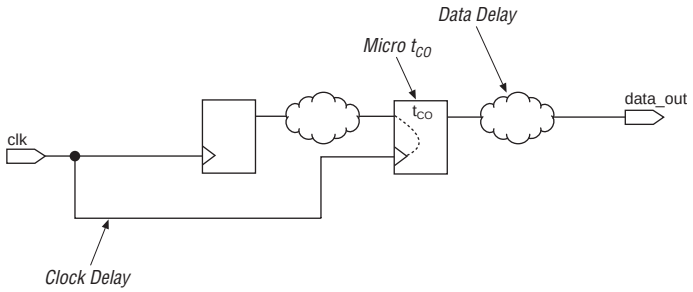
$$t_H = \text{Longest Clock Delay} - \text{Shortest Data Delay} + \text{micro } t_H \text{ of Input Register}$$

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 (refer to 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 (refer to 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 (refer to Equation 10–44).

Equation 10–44.

$$\min t_{PD} = \text{Shortest Pin-to-Pin Delay}$$

Conclusion

Static timing analysis is important to help you verify design timing requirements. Without full design constraints and a complete static timing analysis, you risk circuit failure in complex designs. The Classic Timing Analyzer provides legacy static timing analysis features to analyze your design.

For more advanced constraints and analysis capabilities, Altera recommends that you switch to the TimeQuest Timing Analyzer.

Document Revision History

Table 10–2 shows the revision history for this chapter.

Table 10–2. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Updated "PLL Clocks" on page 10–10. ■ Updated "Early Timing Estimation" on page 10–33. ■ Removed The Timing Analyzer Tool. ■ This was chapter 11 in the Quartus II 9.1 software release.
November 2009	9.1.0	■ Updated the "Introduction" section. ■ Updated "Timing Analysis Tool Setup" on page 10–2.
March 2009	9.0.0	■ This was chapter 9 in version 8.1.
November 2008	8.1.0	■ Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	■ 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.

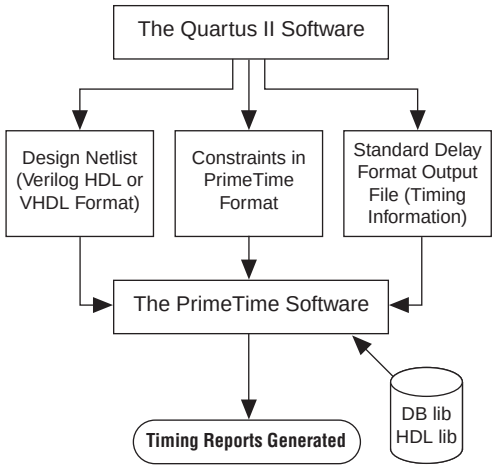
Take an online survey to provide feedback about this handbook chapter.

11. Synopsys PrimeTime Support

QII53005-10.0.0

PrimeTime is the Synopsys stand-alone full chip, gate-level static timing analyzer. 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 11–17

Quartus II Settings for Generating the PrimeTime Software Files

To set up 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 **Settings**, and then 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 HDL** or **VHDL**.

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 Classic Timing Analyzer or the 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* chapter 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 HDL** 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 you select the TimeQuest analyzer, only a Verilog HDL PrimeTime netlist can be generated.

The .sdo File

The Quartus II software saves the **.sdo** file as either <*revision_name*>**_v.sdo** or <*revision_name*>**_vhd.sdo**, depending on whether you select **Verilog HDL** 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 you enable the Classic Timing Analyzer, 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 the TimeQuest analyzer 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 Synopsys Design Constraints File (**.sdc**) file is generated.

After the EDA Netlist Writer has finished, two **.sdc** files are created: <*revision_name*>**_v.sdo** (slow corner) and <*revision_name*>**_v_fast.sdo** (fast corner).

Generating Multiple Operating Conditions with the TimeQuest Analyzer

You can specify different operating conditions to the EDA Netlist Writer for PrimeTime analysis. The different operating conditions are reflected in the .**sdo** file generated by the EDA Netlist Writer.

From the TimeQuest analyzer 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. Enter each command at the command prompt.

The `--tq2pt` option for `quartus_sta` is required only if the project does not specify that the PrimeTime tool is 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 are 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 0º 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 are 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, 85º 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, 0º 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, 0º C):
  .vo file—primetime_one_slow_corner_files/<*project name*>**.vo**
  .sdo file—primetime_one_slow_corner_files/<*project name*>**_v.sdo**

The **primetime_one_slow_corner_files** directory 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 select **Verilog HDL** or **VHDL** in the **Format for output netlist** list on the **Timing Analysis** page of the **Settings** dialog box, when the 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, perform the following:

1. On the Assignments menu, click **Settings**, and then click **EDA Tool Settings**
2. Expand **EDA Tool Settings** under the **Category** list.

In the dialog box, you can specify VHDL or Verilog HDL for the format of 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 HDL 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 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 **.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 11–1 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 11–1. 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 TimeQuest analyzer is turned on, the EDA Netlist Writer generates and saves the script as <*revision_name*>**.pt.tcl**.

The EDA Netlist Writer converts all TimeQuest 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 select.

Table 11–2 shows the files that are generated for the PrimeTime software when the Classic Timing Analyzer is selected.

Table 11–2. Classic Timing Analyzer-Generated PrimeTime Files

File	Description
<*revision_name*>**.vho** | <*revision_name*>**.vo**	The PrimeTime software output netlist. Either a VHDL Output File (**.vho**) or a Verilog Output File (**.vo**) 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 (**vhd.sdo**) or a Verilog Standard Delay Output File (**v.sdo**) 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 File (**vhd.tcl**) or a Verilog Tcl Script File (**v.tcl**) is generated, depending on the output netlist language set.

Table 11–3 shows the files that are generated for the PrimeTime software when the TimeQuest analyzer is selected. The EDA Netlist Writer supports the output netlist format only when the TimeQuest analyzer is enabled.

Table 11–3. TimeQuest Timing Analyzer-Generated PrimeTime Files

File	Description
<*revision_name*>**.vo**	The PrimeTime software output netlist. When the TimeQuest analyzer is enabled, only PrimeTime (Verilog HDL) is supported.
<*revision_name*>**_v.sdo** | <*revision_name*>**_v_fast.sdo**	The PrimeTime software standard delay file. When the TimeQuest analyzer is enabled, only PrimeTime (Verilog HDL) is supported.
<*revision_name*>**.pt.tcl**	PrimeTime setup and constraint script. When the TimeQuest analyzer is enabled, only PrimeTime (Verilog HDL) is supported.
<*revision_name*>**.collections.sdc**	Contains the mapping from the TimeQuest analyzer netlist to the PrimeTime netlist.
<*revision_name*>**.constraints.sdc**	Contains the converted TimeQuest 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 using the first line of Example 11–1:

```
set quartus_root "altera/quartus/" set search_path [list . [format
"%s%s" $quartus_root "eda/synopsys/primetime/lib"]  ]
```

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 TimeQuest 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 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 timing 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`, and 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 Classic Timing Analyzer and the PrimeTime Software

Both the Classic Timing Analyzer and the TimeQuest 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 TimeQuest analyzer and PrimeTime use an equivalent set of equations when reporting the static timing analysis results for a design. However, the Classic Timing Analyzer uses slightly different reporting equations when reporting the static timing analysis results for a design. This section describes the differences between the Classic Timing Analyzer and the PrimeTime software.

The timing report generated by the Classic Timing Analyzer differs from the report generated by the PrimeTime software. Both tools provide the same data, but the data is presented in different formats. The following sections show how the PrimeTime software reports the following slack values differently from the Classic Timing Analyzer report:

- "Clock Setup Relationship and Slack" on page 11–9
- "Clock Hold Relationship and Slack" on page 11–12
- "Input Delay and Output Delay Relationships and Slack" on page 11–15

Clock Setup Relationship and Slack

The Classic Timing Analyzer performs a setup check that ensures that the data launched by source registers is latched correctly at the destination registers. The 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 is not met. The 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.

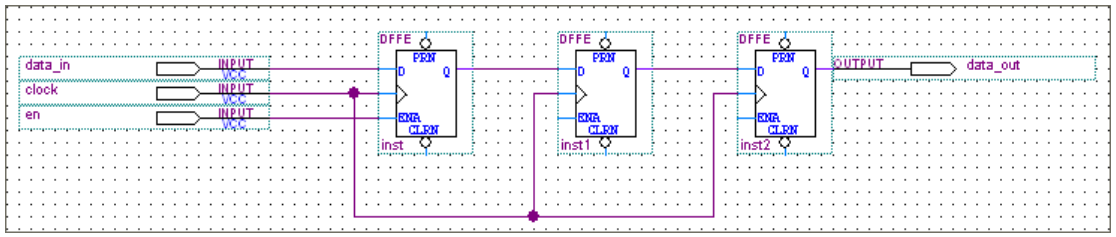
$$\text{Largest Register-to-Register Requirement} = \text{Setup Relationship between Source and Destination} + \text{Largest Clock Skew} - \text{Micro } t_{co} \text{ of Destination Register} - \text{Micro } t_{su} \text{ of Destination Register}$$

$$\text{Setup Relationship between Source and Destination} = \text{Latch Edge} - \text{Launch Edge}$$

$$\text{Clock Skew} = \text{Shortest Clock Path to Destination} - \text{Longest Clock Path to Source}$$

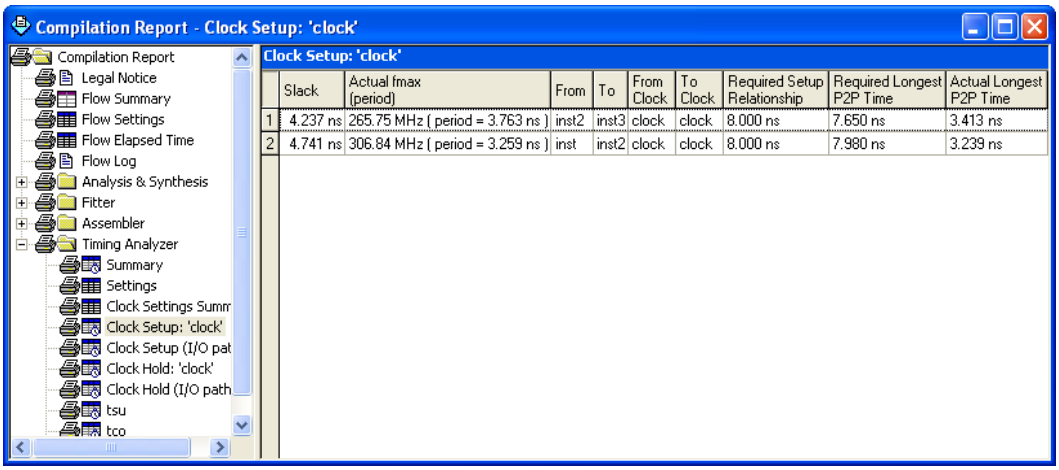
Figure 11–2 shows a simple three-register design.

Figure 11–2. Simple Three-Register Design



The Classic Timing Analyzer generates a report for the design, as shown in Figure 11–3.

Figure 11–3. Timing Analyzer Report from Figure 11–2



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 through Equation 11–7, used by the PrimeTime software, are essentially the same as those used by the 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.

$$\text{Data Required} = \text{Clock Arrival} - \text{Micro } t_{su}$$

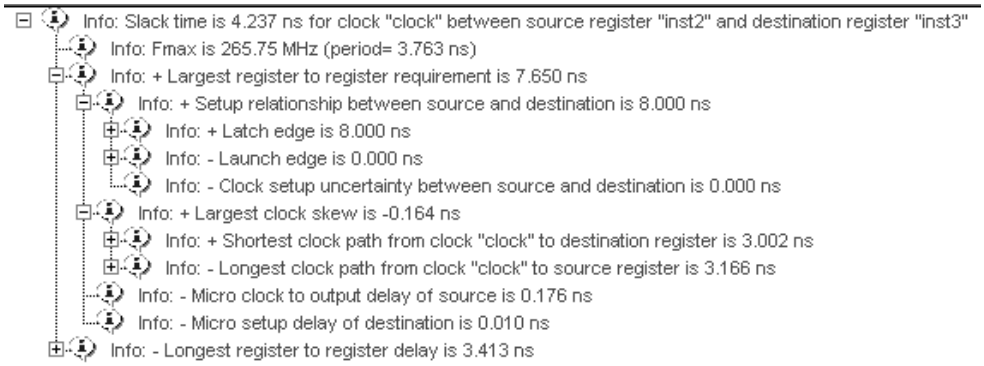
Equation 11–7.

$$\text{Data Arrival} = \text{Launch Edge} + \text{Longest Clock Path to Source} + \text{Micro } t_{co} + \text{Longest Data Delay}$$

The longest data delay in the previous equation is equal to register-to-register data delay.

Figure 11–4 shows a clock setup check in the Quartus II software.

Figure 11–4. Clock Setup Check Reporting with the Classic Timing Analyzer



The results in Equation 11–8 are obtained by extracting the numbers from the Classic Timing Analyzer report and applying them to the clock setup slack equations from the 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 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 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.

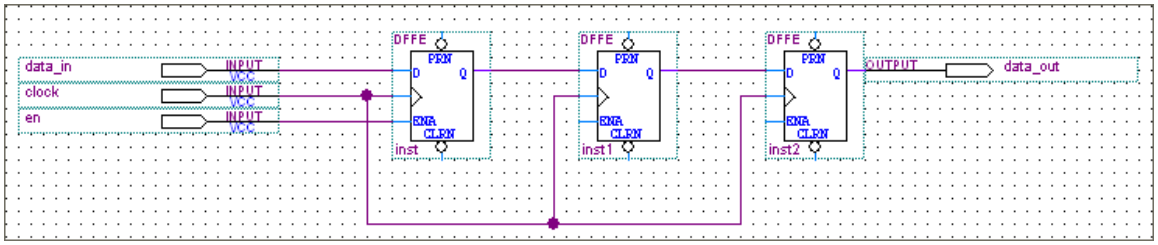
Hold Relationship between Source & Destination = Latch Edge – Launch Edge

Equation 11–13.

Smallest Clock Skew = Longest Clock Path from Clock to Destination Register – Shortest Clock Path from Clock to Source Register

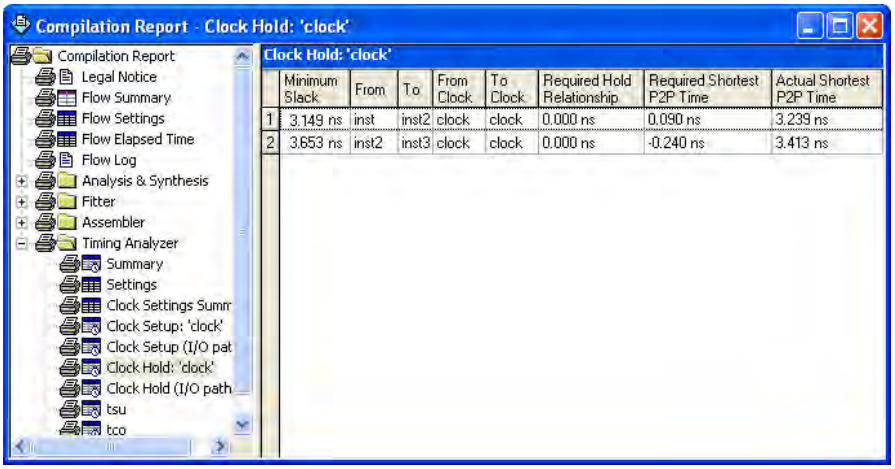
Figure 11–5 shows a simple three-register design.

Figure 11–5. Simple Three-Register Design



The Classic Timing Analyzer generates a report as shown in Figure 11–6.

Figure 11–6. Timing Analyzer Report Generated from the Three-Register Design



The previous equations are similar to those found in the Quartus II software. Equation 11–14 through Equation 11–17 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.

$$\text{Clock Arrival} = \text{Latch Edge} + \text{Longest Clock Path to Destination}$$

Equation 11–16.

$$\text{Data Required} = \text{Clock Arrival} - \text{Micro } t_H$$

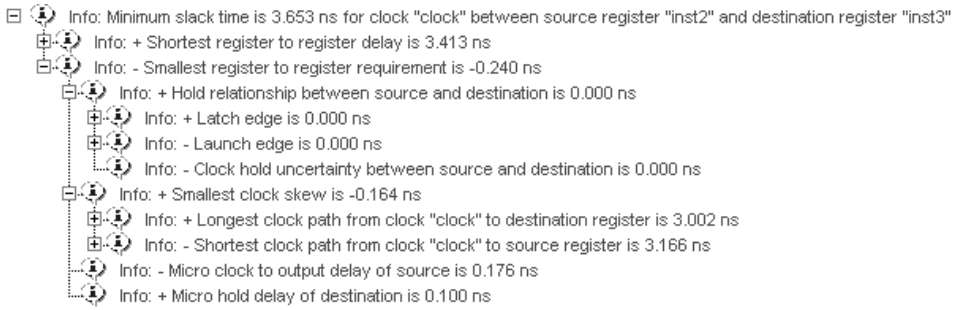
Equation 11–17.

$$\text{Data Arrival} = \text{Launch Edge} + \text{Longest Clock Path to Source} + \text{Micro } t_{co} + \text{Shortest Data Delay}$$

The shortest register-to-register delay in the previous equation is equal to register-to-register data delay.

Figure 11–7 shows a clock setup check with the Classic Timing Analyzer.

Figure 11–7. Clock Hold Check Reporting with the Classic Timing Analyzer



The results in Equation 11–18 are obtained by extracting the numbers from the Timing Analysis report and applying the clock setup slack equations from the 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 Classic Timing Analyzer are similar to the clock setup and clock hold relationship reports. Figure 11–8 shows the input delay and output delay report for the design shown in Figure 11–5 on page 11–13.

Figure 11–8. Input and Output Delay Reporting with the Classic Timing Analyzer

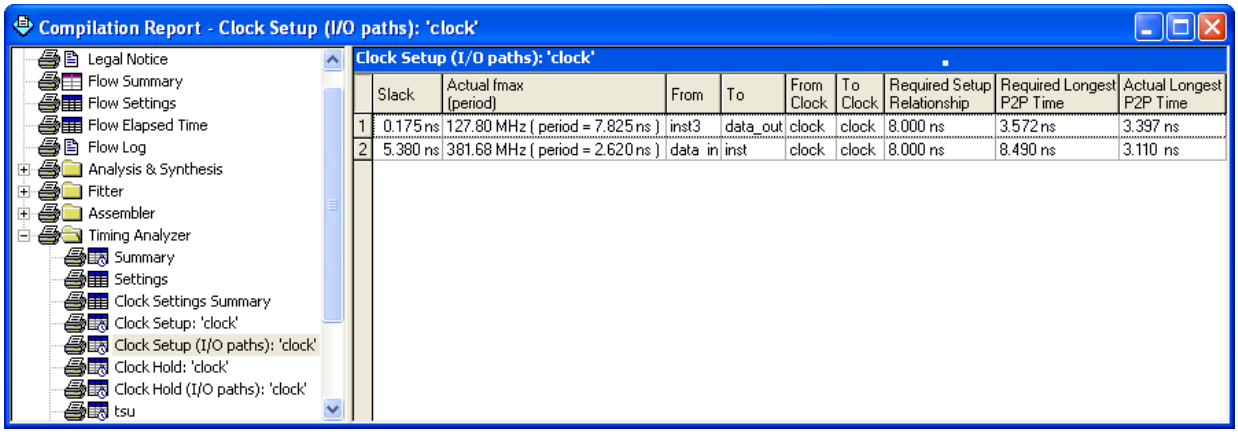
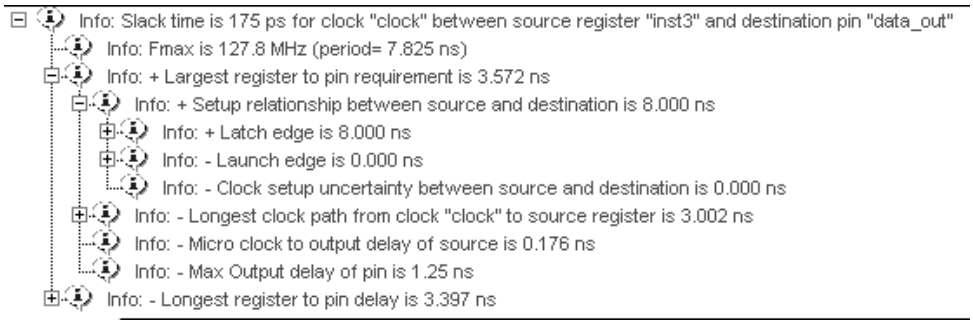


Figure 11–9 shows the fully expanded view for the output delay path.

Figure 11–9. Output Delay Path Reporting with the Classic Timing Analyzer



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 analyzer, and the PrimeTime software. The following sections explain the differences between the two static timing analysis engines and the PrimeTime software.

Classic Timing Analyzer and PrimeTime Software

The following section describes the differences between the Classic Timing Analyzer and the PrimeTime software.

Rise/Fall Support

The Classic Timing Analyzer does not support rise/fall analysis. However, rise/fall support is available in PrimeTime.

Minimum and Maximum Delays

The Classic Timing Analyzer 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 of 2 to 3%.

Recovery/Removal Analysis

The Classic Timing Analyzer performs a more pessimistic recovery/removal analysis for asynchronous paths than PrimeTime. This can result in different delays reported between the two tools.

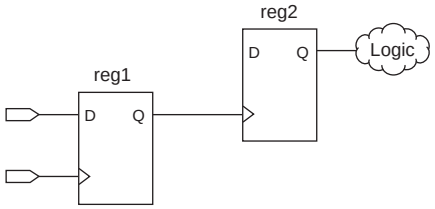
Encrypted Intellectual Property Blocks

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 Classic Timing Analyzer to perform a complete static timing analysis on the design. However, licensed and encrypted IP blocks do not permit output netlists to be generated when using PrimTime as the static timing analysis tool. (The EDA Netlist Writer does not generate **.vho** or **.vo** netlist files.)

Registered Clock Signals

Registered clock signals are clock signals that pass through a register before reaching the clock port of a sequential element. Figure 11–10 shows an example of a registered clock signal.

Figure 11–10. Registered Clock Signal



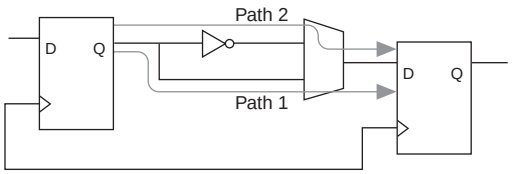
If no clock setting is applied to the register on the clock path (shown as register `reg_1` in Figure 11–10), the Classic Timing Analyzer treats the register in the clock path as a buffer. The delay of the buffer is equal to the CELL delay of the register plus the t_{CO} of the register. The PrimeTime 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 11–11 shows a sample design that contains multiple path pairs between the source register and destination register.

Figure 11–11. Multiple Source and Destination Pairs



The 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 11–11, the 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 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 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 analyzing the dedicated LVDS transceivers in your design, the 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 Classic Timing Analyzer does not factor in the clock latency values when it calculates the clock skew between the two registers. The 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 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 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.

TimeQuest Timing Analyzer and PrimeTime Software

The following sections describe the static timing analysis differences between the TimeQuest 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 TimeQuest analyzer to perform a complete static timing analysis on the design. However, licensed and encrypted IP blocks do not permit output netlists to be generated when using PrimTime as the static timing analysis tool. (The EDA Netlist Writer does not generate .**vho** or .**vo** netlist files.)

Latches

By default, the Quartus II software implements all latches as combinational loops. The TimeQuest analyzer can analyze such latches by treating them as registers with inverted clocks. The TimeQuest analyzer analyzes 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 analyzing the dedicated LVDS transceivers in your design, the TimeQuest 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 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 TimeQuest 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 by the PrimeTime software for compatibility of constraints between the TimeQuest 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 with it), and a data path (a path that feeds into the data-in port of a register), the TimeQuest analyzer reports the data paths, whereas PrimeTime does not.

Inverting and Non-Inverting Propagation

The TimeQuest analyzer 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, the TimeQuest analyzer reports only one timing path for the set of pins that make up the path.

Virtual Generated Clocks

PrimeTime does not support virtual generated clocks. To maintain compatibility between the TimeQuest analyzer and PrimeTime, all generated clocks should have an explicit target specified.

Generated Clocks Derived from Generated Clocks

The 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.

Document Revision History

Table 11–4 shows the revision history for this chapter.

Table 11–4. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Minor corrections throughout, and Quartus II interface changes.
November 2009	9.1.0	■ Updated "Setting the Quartus II Software to Generate the PrimeTime Software Files" figure for changes in the Quartus II software version 9.1
March 2009	9.0.0	■ This was chapter 10 in version 8.1. ■ Updated for the Quartus II software version 9.0 release.
November 2008	8.1.0	■ Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	■ 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.

Take an online survey to provide feedback about this chapter.

Section III. Power Estimation and Analysis

As FPGA designs grow larger and processes continue to shrink, power is 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

  This chapter describes the Altera® Quartus II PowerPlay power analysis tool and how to use the tools to accurately estimate device power consumption.

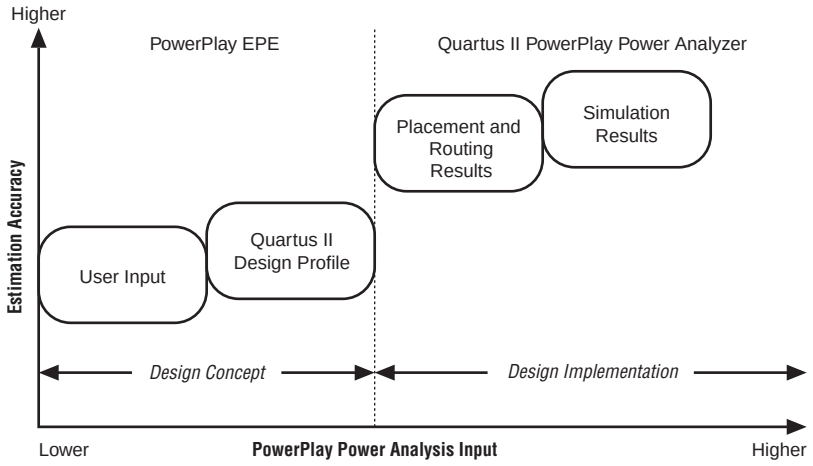
12. PowerPlay Power Analysis

QII53013-10.0.0

This chapter describes how to use the Altera® Quartus® II PowerPlay Power Analysis tools to accurately estimate device power consumption.

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. As shown in Figure 12–1, the PowerPlay Power Analysis tools provide the ability to estimate power consumption from early design concept through design implementation.

Figure 12–1. PowerPlay Power Analysis



For more information about the PowerPlay suite of power analysis and optimizations tools, refer to *About Power Estimation and Analysis* in Quartus II Help.

For more information about acquiring the PowerPlay EPE spreadsheet, refer to PowerPlay Early Power Estimators (EPE) and Power Analyzer on the Altera website.

This chapter discusses the following topics:

- "Creating PowerPlay EPE Spreadsheets"
- "Types of Power Analysis" on page 12–4
- "Factors Affecting Power Consumption" on page 12–4
- "PowerPlay Power Analyzer Flow" on page 12–7
- "Using Simulation Files in Modular Design Flows" on page 12–10
- "Using the PowerPlay Power Analyzer" on page 12–17
- "Conclusion" on page 12–27

Creating PowerPlay EPE Spreadsheets

You can use PowerPlay EPE spreadsheets to perform a preliminary thermal analysis and power consumption estimate for your design. You can either enter the data manually or use the tools in the Quartus II software to assist you with generating the device resources usage information for your design.

For more information about generating a PowerPlay EPE File in the Quartus II software, refer to *Performing an Early Power Estimate Using the PowerPlay Early Power Estimator* in Quartus II Help.

Figure 12–2 shows an example of the contents of a PowerPlay EPE File generated for a design that targets a Stratix III device.

Figure 12–2. Example of a PowerPlay EPE File

	A	B	C	D	E	F	G	H
1	EARLY_POWER_ESTIMATOR_FILE_FORMAT_VERSION	6						
2	QUARTUS_II_VERSION	9.0 Build 235 06/17/2009 SP 2 SJ Full Version						
3	PROJECT	fractal						
4	REVISION	fractal_extra						
5	PROJECT_FILE	C:/Powe_Lab_and_Test_designs/fractal.qpf						
6	TIME	Fri Aug 14 11:45:17 2009						
7	TIME_SECONDS	1250275517						
8	FAMILY	Stratix III						
9	DEVICE	EP3SE260						
10	PACKAGE	FBGA						
11	PART	EP3SE260H780C4						
12	POWER_USE_DEVICE_CHARACTERISTICS	TYPICAL						
13	POWER_AUTO_COMPUTE_TJ	ON						
14	POWER_TJ_VALUE	25						
15	POWER_USE_CUSTOM_COOLING_SOLUTION	OFF						
16	MIN_JUNCTION_TEMPERATURE	0						
17	MAX_JUNCTION_TEMPERATURE	85						
18	POWER_PRESET_COOLING_SOLUTION	23 mm heat sink with 200 LFpM airflow						
19	POWER_BOARD_THERMAL_MODEL	None (Conservative)						
20	POWER_USE_TA_VALUE	25						
21	POWER_BOARD_TEMPERATURE	-1						
22	POWER_OJC_VALUE	0.1						
23	POWER_OCS_VALUE	0.1						
24	POWER_OSA_VALUE	1.8						
25	POWER_OJB_VALUE	-1						
26	VCCIO		1A	2.5	1B	0	1C	2.5
27	VCCPD		1A	2.5	1B	0	1C	2.5
28	RAIL_VOLTAGES		VCC	1.1	VCCPT	2.5	VCCA_PLL	2.5
29	HIGH_SPEED		NUM_HIGH_SPEED_M9K block_TILES	35	NUM_M9K block_TILES_USED	35		
30								
31								
32	BLOCK	M9K block	count	16	ram_mode	Simple Dual Port	ram_read_durir	new
33	BLOCK	M9K block	count	3	ram_mode	Simple Dual Port	ram_read_durir	new
34	BLOCK	M9K block	count	16	ram_mode	Simple Dual Port	ram_read_durir	new
35	BLOCK	Combinational cell	count	28	avg_toggle_rate	181153.408	avg_toggle_rate	0
36	BLOCK	Combinational cell	count	2967	avg_toggle_rate	7292423.461	avg_toggle_rate	0.147695
37	BLOCK	Combinational cell	count	47	avg_toggle_rate	639806.5769	avg_toggle_rate	0.044742
38	BLOCK	Clock enable block	count	1	avg_toggle_rate	0	avg_toggle_rate	0
39	BLOCK	Clock enable block	count	1	avg_toggle_rate	0	avg_toggle_rate	0
40	BLOCK	Clock enable block	count	1	avg_toggle_rate	28500000	avg_toggle_rate	1.993007
41	BLOCK	Clock enable block	count	1	avg_toggle_rate	28550000	avg_toggle_rate	2
42	BLOCK	Clock enable block	count	1	avg_toggle_rate	98750000	avg_toggle_rate	2
43	BLOCK	Register cell	count	2567	avg_toggle_rate	3790981.691	avg_toggle_rate	0.076779
44	BLOCK	Register cell	count	71	avg_toggle_rate	249731.7324	avg_toggle_rate	0.017464
45	BLOCK	MLAB cell	count	1	mlab_width	8	mlab_depth	4
46	BLOCK	I/O pad	count	16	avg_toggle_rate	442715.8125	avg_toggle_rate	0
47	BLOCK	I/O pad	count	26	avg_toggle_rate	982211.5385	avg_toggle_rate	0
48	BLOCK	I/O pad	count	1	avg_toggle_rate	100850000	avg_toggle_rate	2.042532
49	BLOCK	I/O pad	count	4	avg_toggle_rate	12500	avg_toggle_rate	0.000253

fractal_extra_early_pwr

The PowerPlay EPE spreadsheet includes the Import Data macro that parses the information in the PowerPlay EPE File and transfers it into the spreadsheet. If you do not want to use the macro, you can manually transfer the data into the PowerPlay EPE spreadsheet.

For example, after importing the PowerPlay EPE File information into the PowerPlay EPE spreadsheet, you can add additional device resource information at any time. If the existing Quartus II project represents only a portion of your full design, you must enter the additional device resources used in the final design manually.

PowerPlay EPE File Generator Compilation Report

After successfully generating the PowerPlay EPE File, you can locate a PowerPlay EPE File Generator report under the **Compilation Report** section. This report contains different sections, such as Summary, Settings, Generated Files, Confidence Metric Details, and Signal Activities. For more information about the PowerPlay EPE File Generator report, refer to "PowerPlay Power Analyzer Compilation Report" on page 12–23.

Table 12–1 lists the main differences between the PowerPlay EPE and the Quartus II PowerPlay Power Analyzer.

Table 12–1. Comparison of the PowerPlay EPE and Quartus II PowerPlay Power Analyzer

Characteristic	PowerPlay EPE	Quartus II PowerPlay Power Analyzer
Phase in the design cycle	Any time	Post-fit
Tool requirements	Spreadsheet program or the Quartus II software	The Quartus II software
Accuracy	Medium	Medium to very high
Data inputs	■ Resource usage estimates ■ Clock requirements ■ Environmental conditions ■ Toggle rate	■ Post-fit design ■ Clock requirements ■ Signal activity defaults ■ Environmental conditions ■ Register transfer level (RTL) simulation results (optional) ■ Post-fit simulation results (optional) ■ Signal activities per node or entity (optional)
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) The PowerPlay EPE output varies by device family as some features might not be available.

(2) Available only for Arria II GX, Cyclone III, Cyclone III LS, Cyclone IV GX, MAX II, Stratix III, Stratix IV series, and Stratix V device families.

The result of the PowerPlay Power Analyzer is only an estimation of power. Altera does not recommend using the result as a specification. The purpose of the estimation is to help you to establish a guide for the power budget of your design. Altera recommends measuring the actual power on the board. You must measure the total dynamic current of your design 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 PowerPlay Power Analyzer or data sheet must be used because the tested devices might not exhibit worst-case behavior.

Types of Power Analysis

Understanding the uses of power analysis and the factors affecting power consumption helps you to use the PowerPlay Power Analyzer effectively. Power analysis meets two significant planning requirements:

- **Thermal planning—**The cooling solution must be sufficient to dissipate the heat generated by the device. 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, if you are using terminated I/O standards, some of the power drawn from the power supply of the device dissipates in termination resistors rather than in the device.

Power analysis also addresses the activity of your design over time as a factor that impacts the power consumption of the device. Static power is 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 EPE at the early stages of your design cycle, or use the PowerPlay Power Analyzer reports when your design is complete to get an estimate of your design power requirement.

Factors Affecting Power Consumption

This section describes the factors affecting power consumption. Understanding these factors allows you to use the PowerPlay 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 consumes less static power than the high-performance and full-featured Stratix II device family.

Power consumption also varies in a single device family. A larger device consumes more static power than a smaller device in the same family because of its larger transistor count. Dynamic power can also increase with device size in devices that employ global routing architectures, for example, the MAX device family. Cyclone, Max II, and Stratix devices do not exhibit significantly increased dynamic power as device size increases.

The choice of device package also affects the ability of the device to dissipate heat. This choice can impact your cooling solution choice required to meet junction temperature constraints.

Process variation can affect power consumption. Process variation primarily impacts static power because 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 the device. The main environmental parameters affecting junction temperature are the cooling solution and ambient temperature.

Airflow

Airflow is a measure of how quickly heated air is removed from the vicinity of the device and replaced by air at ambient temperature. Airflow 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 airflow 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}.

Junction Temperature

The junction temperature of a device is equal to:

$$T_{Junction} = T_{Ambient} + P_{Thermal} \cdot \theta_{JA}$$

in which 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}), having units of degrees Celsius per watt. It is used in conjunction with the board temperature, as well as the top-of-chip θ_{JA} and ambient temperatures, to compute junction temperature.

Device Resource Usage

The number and types of device resources 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 circuit elements. The operating mode of each circuit element also affects its power consumption. For example, a DSP block performing 18 × 18 multiplications and a DSP block performing multiply-accumulate operations consume different amounts of dynamic power because of different amounts of internal capacitance being charged on each transition.The operating mode of a circuit element also affects static power.

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 tend to consume less power. The location of the logic array blocks (LABs) driven by the clock network can also 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 your 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 software models 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 PowerPlay Power Analyzer.

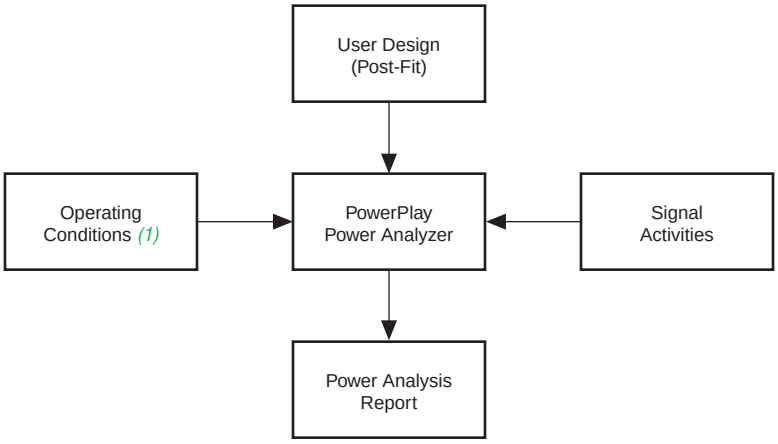
The static power consumed by both routing and logic can sometimes be affected by the static probabilities of their input signals. This effect is due to state-dependent leakage and has a larger effect on smaller process geometries. The Quartus II software models this effect on devices at 90 nm (or smaller) if it is 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 for analysis must represent the actual operating behavior of your design. Inaccurate signal toggle rate data is the largest source of power estimation error.

PowerPlay Power Analyzer Flow

The PowerPlay Power Analyzer supports accurate power estimations by allowing you to specify all the important design factors affecting power consumption. Figure 12–3 shows the high-level PowerPlay Power Analyzer flow.

Figure 12–3. PowerPlay Power Analyzer High-Level Flow



Note to Figure 12–3:

(1) Operating condition specifications are available only for Arria II GX, Arria GX, Cyclone II, Cyclone III series, Cyclone IV series, HardCopy series, MAX II, Stratix II, Stratix II GX, Stratix III, Stratix IV, and Stratix V device families.

The PowerPlay Power Analyzer requires your design to be synthesized and fitted to the target device. You must specify the electrical standard used by each I/O cell and the capacitive load on each I/O standard in your design to obtain accurate I/O power estimates.

Operating Conditions

For Arria II GX, Arria GX, Cyclone II, Cyclone III series, Cyclone IV series, HardCopy series, MAX II, Stratix II, Stratix II GX, Stratix III, Stratix IV, and Stratix V device families, you can specify the operating settings and conditions for power analysis in the Quartus II software.

The following settings are available when you click **Operating Settings and Conditions** in the **Settings** dialog box:

- **Device power characteristics**—You can specify that the device has typical power consumption characteristics or maximum power consumption characteristics.

  For more information, refer to *Operating Setting and Conditions Page (Settings Dialog Box)* in Quartus II Help.

- **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 Stratix III devices with variable voltage support. The power consumption of a device is heavily dependent on 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 logic array blocks (MLABs), DSP functions, memory, and interconnects.

- **Environmental conditions and junction temperature**—The PowerPlay 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. If you are 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 it because the PowerPlay Power Analyzer provides more accurate results by computing the junction temperature.

- **Board Thermal Modeling**—If you want the PowerPlay 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 must 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 modeling** option to **None** (conservative). In this case, the path through the board and the path through power dissipation are not considered, and a more conservative thermal power estimate is obtained.

  The **Board thermal modeling** option is only available if you select the **Auto compute junction temperature using cooling solution** option with the pre-set cooling solution set to a heat sink solution option or custom solution. The **Auto compute junction temperature using cooling solution** 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.

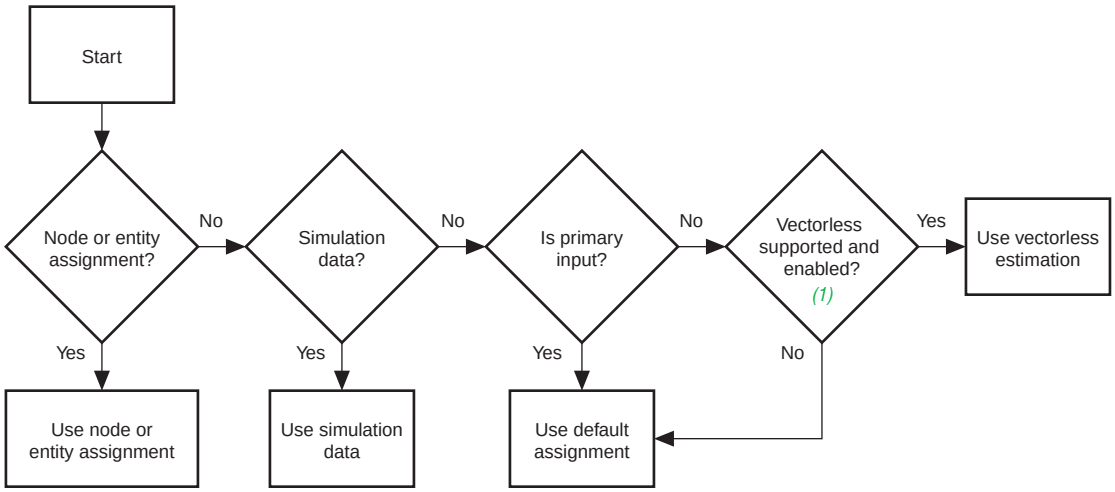
Signal Activities Data Sources

The PowerPlay Power Analyzer provides a flexible framework for specifying signal activities. It 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 allows you to mix and match the signal-activity data sources on a signal-by-signal basis. Figure 12–4 shows the priority scheme. The following sections describe the data sources.

Figure 12–4. Signal-Activity Data Source Priority Scheme



Note to Figure 12–4:

(1) Vectorless estimation is available only for the Arria II GX, Arria GX, Cyclone II, Cyclone III series, Cyclone IV series, MAX II, Stratix II, Stratix II GX, Stratix III, Stratix IV, and Stratix V device families.

Simulation Results

The PowerPlay Power Analyzer directly reads the waveforms generated by a design simulation. The static probability and toggle rate for each signal are calculated from the simulation waveform. Power analysis is most accurate when you use representative input stimuli to generate simulations.

The PowerPlay Power Analyzer reads results generated by the following simulators:

- ModelSim®
- ModelSim-Altera
- QuestaSim
- Active-HDL
- NCSim

- VCS
- VCS MX
- Riviera-PRO

Signal activity and static probability information derive from a Verilog Value Change Dump File (**.vcd**). For more information, refer to "Signal Activities" on page 12–6.

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** that encodes the simulated waveforms. The Quartus II PowerPlay 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, can generate a **.vcd** that can then be used with the PowerPlay Power Analyzer. For those simulators, you must manually create a simulation script to generate the appropriate **.vcd**.

You can use a **.vcd** created for power analysis to optimize your design for power during fitting by utilizing the appropriate settings in the **PowerPlay power optimization** list, available in the **Fitter Settings** page of the **Settings** dialog box.

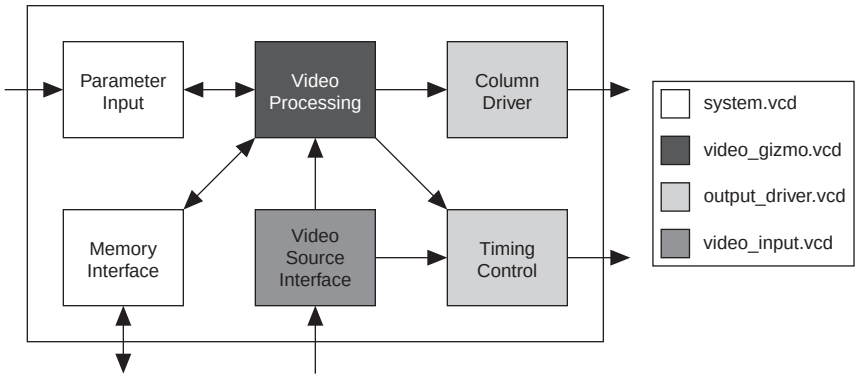
For more information about power optimization, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*.

For more information about how to create a **.vcd** in other third-party EDA simulation tools, refer to *Section I. Simulation* in volume 3 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. You can perform simulation on a complete design or on each modular design for verification. The PowerPlay Power Analyzer supports modular design flows when reading the signal activities generated from these simulation files. An example of a modular design flow is shown in Figure 12–5.

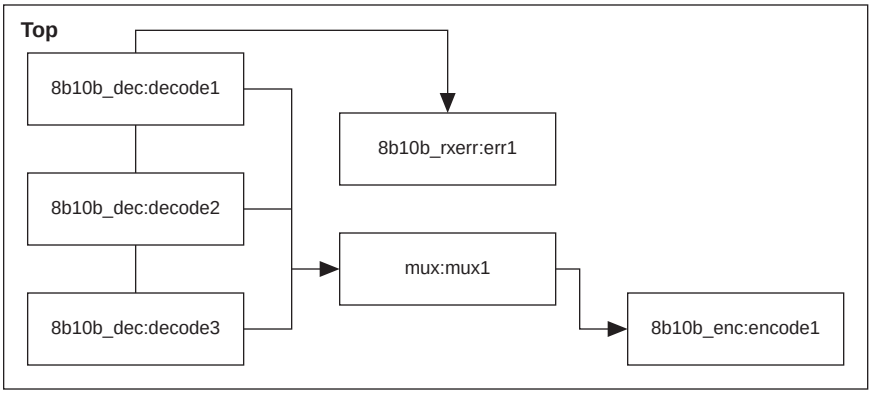
Figure 12–5. Modular Simulation Flow



When specifying a simulation file, an associated design entity name is given, such that the signal activities derived from the simulation file (**.vcd**) are imported into the PowerPlay Power Analyzer for that particular design entity. The PowerPlay Power Analyzer also supports the specification of multiple **.vcd** 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 **.vcd** for your design, it is possible that more than one simulation file contains signal-activity information for the same signal. When you apply multiple **.vcd** to the same design entity, the signal activity used in the power analysis is the equal-weight arithmetic average of each **.vcd**. When you apply multiple simulation files to design entities at different levels in your design hierarchy, the signal activity in the power analysis derives from the simulation file that applies to the most specific design entity.

Figure 12–6 shows an example of a hierarchical design. The top-level module of your design, called **Top**, consists of three 8b/10b decoders, followed by a multiplexer. The output of the multiplexer is then encoded again before being the output from your design. There is also an error-handling module that handles any 8b/10b decoding errors. The top contains the top-level entity of your design 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 **.vcd** into the PowerPlay Power Analyzer.

Figure 12–6. Example Hierarchical Design



Complete Design Simulation

You can simulate the entire design top, generating a **.vcd** from a third-party simulator. The **.vcd** can then be imported (specifying entity top) into the PowerPlay Power Analyzer. The resulting power analysis uses all the signal activities information from the generated **.vcd**, including those that apply to submodules, such as `decode [1-3]`, `err1`, `mux1`, and `encode1`.

Modular Design Simulation

You can independently simulate submodules of the design top, and then import all the resulting **.vcd** into the PowerPlay 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** 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.vcd**, the import specifications in Table 12–2 are used.

Table 12–2. Import Specifications

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`
8b10b_enc.vcd	`Top|8b10b_enc:encode1`
mux.vcd	`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 design top, you can have three different simulation testbenches: one for normal operation and two for corner cases. Each of these simulations produces a separate **.vcd**. In this case, apply the different **.vcd** 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.vcd	`Top`
corner1.vcd	`Top`
corner2.vcd	`Top`

The resulting power analysis uses an arithmetic average that the signal activities calculated from each simulation file to obtain the final signal activities used. If a signal `err_out` has a toggle rate of zero toggles per second in **normal.vcd**, 50 toggles per second in **corner1.vcd**, and 70 toggles per second in **corner2.vcd**, the final toggle rate 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 submodule, 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 **.vcd** 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) derives signal activities for overlapping nodes.

Partial Simulations

You can perform a simulation in which the entire simulation time is not applicable to signal-activity calculation. For example, 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 only 80% of their steady state value (because the chip is in reset for the first 20% of the simulation). In this case, you must specify the useful parts of the **.vcd** 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 **.vcd** applied to entities other than the top-level entity. In a modular design flow, the gate-level simulation files created in different Quartus II 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** into another project called **Top**, you might encounter name mismatches when applying the **.vcd** to the `8b10b_enc` module in the **Top** project. This mismatch happens because all the combinational nodes in the **8b10b_enc.vcd** might be named differently in the **Top** project.

You can avoid name mismatching with only RTL simulation data, in which register names do not change, or with an incremental compilation 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 the node names of your design.

For more information about the incremental compilation 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 PowerPlay Power Analyzer defines a glitch as two signal transitions 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 contains glitches for some signals. The logic and routing structures of the device 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 model. Different models cause differences in signal activity and power estimation. The inertial delay model, which is the ModelSim default model, filters out more glitches than the transport delay model and usually yields a lower power estimate.

Altera recommends using the transport simulation model when using the Quartus II software glitch filtering support with third-party simulators. Simulation glitch filtering has little effect if you use the inertial simulation model.

For more information about how to set the simulation model type for your specific simulator, refer to 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 to ensure that the glitch does not affect simulated results. It prevents a glitch on one signal from producing non-physical glitches on all downstream logic, which can result in a signal toggle rate and a power estimate that are 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 such functions can have power estimates that are too high when you do not use glitch filtering.

Altera recommends using the glitch filtering feature to obtain the most accurate power estimates. For **.vcd**, the PowerPlay Power Analyzer flows support two levels of glitch filtering, both of which are recommended for power estimation.

In the first level of glitch filtering, glitches are filtered during simulation. To enable this level of glitch filtering in the Quartus II software for supported third-party simulators, follow these steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Simulation** under **EDA Tool Settings**. The **Simulation** page appears.
3. Select the **Tool name** to use for the simulation.
4. Turn on **Enable glitch filtering**.

The second level of glitch filtering occurs while the PowerPlay Power Analyzer is reading the **.vcd** generated by a third-party simulator. To enable this level of glitch filtering, follow these steps:

On the Assignments menu, click **Settings**. The **Settings** dialog box appears.

1. In the **Category** list, select **PowerPlay Power Analyzer Settings**. The **PowerPlay Power Analyzer Settings** page appears.

2. Under **Input File(s)**, turn on **Perform glitch filtering on VCD files**.

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 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 PowerPlay Power Analyzer), Altera recommends turning off the glitch filtering option to produce the most rigorous and conservative simulation from a functionality viewpoint. When performing simulation to produce input for the PowerPlay Power Analyzer, Altera recommends turning on the 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.

You must use the Assignment Editor or Tcl commands to create 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 a 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*.

Assigning specific toggle rates and static probabilities to individual nodes and entities is appropriate for signals in which 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 PowerPlay 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 creating 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 create the logic assignment, you must manually enter the two differentiating node names to create the specific assignment for the input and output port of the bidirectional pin.

Timing Assignments to Clock Nodes

For clock nodes, the PowerPlay 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 is 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 derives 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 PowerPlay 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 PowerPlay Power Analyzer substitutes and reports a toggle rate of zero.

Vectorless Estimation

For some device families, the PowerPlay 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 II GX, Arria GX, Cyclone II, Cyclone III series, Cyclone IV series, MAX II, Stratix II, Stratix II GX, Stratix III, Stratix IV, and Stratix V 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 the 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 the 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.

For more information about using the PowerPlay Power Analyzer, refer to *Performing Power Analysis with the PowerPlay Power Analyzer* in Quartus II Help.

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 most 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 of this method is that the simulation time is long.

Signal Activities from Full Post-Fit Netlist (Zero Delay) Simulation

The zero delay simulation flow is used with designs for which signal activities from a full post-fit netlist (timing) simulation are not available. Zero delay simulation is as accurate as timing simulation in 95% of designs with no glitches.

If your design has glitches, power may be underestimated. Altera recommends using the signal activities from a full post-fit netlist (timing) simulation to achieve accurate power estimation of your design.

The following designs might exhibit glitches:

- Designs with many XOR gates (for example, an encryption core)
- Designs with arithmetic blocks without input and output registers (DSPs and carry chains)

  For more information about creating zero delay simulation signal activities, refer to "Generating a .vcd from Full Post-Fit Netlist (Zero Delay) Simulation" on page 12–20.

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, giving good results. 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 and User-Supplied Input Pin Activities

This flow provides a low level of accuracy, because vectorless estimation for registers is not entirely accurate.

Signal Activities from User Defaults Only

This flow provides the lowest degree of accuracy.

Generating a .vcd

In previous versions of the Quartus II software, you could use either the Quartus II simulator or an EDA simulator to perform your simulation. The Quartus II software no longer supports a built-in simulator, and you must use EDA simulators to perform simulation. Use the **.vcd** as the input to the PowerPlay Power Analyzer to estimate power for your design.

For more information about the supported third-party simulators, refer to "Simulation Results" on page 12–9.

To create a **.vcd** for your design, follow these steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, under **EDA Tool Settings**, click **Simulation**. The **Simulation** page appears.
3. In the **Tool name** list, select your preferred EDA simulator.
4. In the **Format for output netlist** list, select **Verilog HDL**, or **SystemVerilog HDL**, or **VHDL**.
5. Turn on **Generate Value Change Dump (VCD) file script**.

   This turns on the **Map illegal HDL characters** and **Enable glitch filtering** options. The **Map illegal HDL characters** option ensures that all signals have legal names and that signal toggle rates are available later in the PowerPlay Power Analyzer.

6. By turning on **Enable glitch filtering**, glitch filtering logic is the output when you generate an EDA netlist for simulation. This option is available regardless of whether or not you want to generate the **.vcd** scripts. For more information about glitch filtering, refer to "Glitch Filtering" on page 12–14.

When performing simulation using ModelSim, the **+nospecify** option for the `vsim` command disables the **specify path delays and timing checks** option in ModelSim. By enabling glitch filtering on the **Simulation** page, the simulation models include specified path delays. Thus, ModelSim might fail to simulate a design if glitch filtering is enabled, and the **+nospecify** option is specified. Altera recommends removing the **+nospecify** option from the ModelSim `vsim` command to ensure accurate simulation for power estimation.

7. Click **Script Settings**. The **Script Settings** dialog box appears.

   Select which signals must be output to the **.vcd**. With **All signals** selected, the generated script instructs the third-party simulator to write all connected output signals to the **.vcd**. 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**, except logic cell combinational outputs.

The file can become extremely large if you write all output signals to the file because 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 instructs the third-party simulator to generate a **.vcd**.

For more information about the NativeLink feature, 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** and places it in the project directory.

Generating a .vcd from ModelSim Software

To successfully produce a **.vcd** with the ModelSim software, follow these 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**, click **Simulation**. The **Simulation** page appears.
3. In the **Tool name** list, select your preferred EDA simulator.
4. In the **Format for output netlist** list, select **Verilog HDL**, or **SystemVerilog HDL**, or **VHDL**.
5. Turn on **Generate Value Change Dump (VCD) file script**.
6. To generate the **.vcd**, perform a full compilation.
7. In the ModelSim software, compile the files necessary for simulation.
8. Load your design by clicking **Start Simulation** on the Tools menu, or use the `vsim` command.

9. Use the **.vcd** script created in step 6 using the following command:
`source <design>_dump_all_vcd_nodes.tcl`
10. Run the simulation (for example, `run 2000ns` or `run -all`).
11. Quit the simulation using the `quit -sim` command, if required.
12. Exit the ModelSim software. If you do not exit the software, the ModelSim software might end the writing process of the **.vcd** improperly, resulting in a corrupt **.vcd**.

Generating a .vcd from Full Post-Fit Netlist (Zero Delay) Simulation

To successfully generate a **.vcd** from the full post-fit Netlist (zero delay) simulation, follow these steps:

1. Compile your deisgn in the Quartus II software to generate the Netlist <*project_name*>**.vo**.
2. In <*project_name*>**.vo**, search for the include statement for <*project_name*>**.sdo**, comment the statement out, and save the file.
3. Generate a **.vcd** for power estimation by performing the steps in "Generating a .vcd" on page 12–18.

Altera recommends using the Standard Delay Format Output File (**.sdo**) for gate-level timing simulation. The **.sdo** contains the delay information of each architecture primitive and routing element specific to your design; however, you must exclude the **.sdo** for zero delay simulation.

For more information about how to create a **.vcd** in other third-party EDA simulation tools, refer to *Section I. Simulation* in volume 3 of the *Quartus II Handbook*.

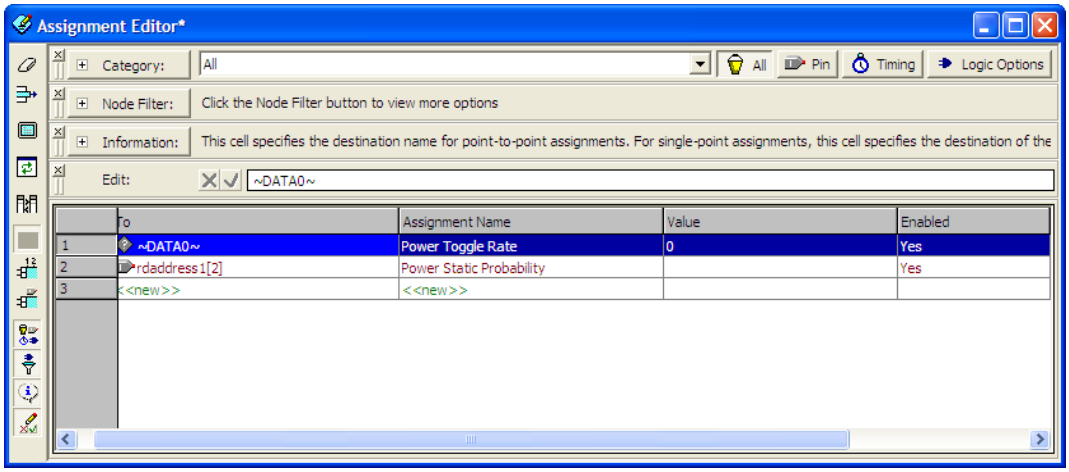
Running the PowerPlay Power Analyzer Using the Quartus II GUI

To run the PowerPlay Power Analyzer using the Quartus II GUI, follow these steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click **PowerPlay Power Analyzer Settings**.
3. If you want to use **.vcd** as an input to the PowerPlay Power Analyzer, turn on **Use input file(s) to initialize toggle rates and static probabilities during power analysis**.
4. Click **Add**. The **Add Power Input File** dialog box appears.
5. Browse to your **.vcd**.
6. Type the entity name into the **Entity** box, or browse to your entity. The **Entity** box enables you to specify the design entity (hierarchy) to which the entered power input file applies. In the **Select Hierarchy** dialog box, you can specify multiple entities in the entity text box with comma delimiters. Under **Simulation period**, you can specify either a **Signal Activity File** or a **VCD file**. The **Limit VCD period** option is available only when **VCD file** is selected. Turning on the **Limit VCD period** option enables you to specify the simulation period when calculating the signal activities.

7. Click **OK**.
8. Turn on the **Write out signal activities used during power analysis** option. In the **Output file name** box, browse to the output file name. The output file contains all the signal activities information used during the power estimation of your design. Turning on the **Write out signal activities used during power analysis** option reduces the run time of any subsequent power estimation.
9. Turn on the **Write signal activities to report file** option (optional).
10. Turn on the **Write power dissipation by block to report file** option to enable the output of detailed thermal power dissipation by block to be included in the PowerPlay Power Analyzer report (optional).
11. Use the Assignment Editor to enter the **Power Toggle Rate** or **Power Toggle Rate Percentage**, and the **Power Static Probability** assignments for a node or entity in your design, shown in Figure 12–7 (optional).

Figure 12–7. Assignment Editor *(Notes 1), (2)*



Notes to Figure 12–7:

(1) The assignments made with the Assignment Editor override the existing values in the **vcd**.
(2) You can also use Tcl script commands to create these assignments.

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*. For information about scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*.

12. 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.

13. Select either **Use default value** or **Use vectorless estimation** for Arria II GX, Arria GX, Cyclone II, Cyclone III series, Cyclone IV series, MAX II, Stratix II, Stratix II GX, Stratix III, Stratix IV, and Stratix V 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 12–16 and "Default Toggle Rate Assignment" on page 12–16.
14. In the **Category** list, click **Operating Settings and Conditions**. This page is only available for the Arria II GX, Arria GX, Cyclone II, Cyclone III series, Cyclone IV series, HardCopy series, MAX II, Stratix II, Stratix II GX, Stratix III, Stratix IV, and Stratix V device families.
15. In the **Device power characteristics** list, select **Typical** or **Maximum**, depending on the options available.
16. In the **Category** list, click **Voltage** under **Operating Settings and Conditions**. The **Voltage** page appears.
17. 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.
18. In the **Category** list, click **Temperature** under **Operating Settings and Conditions**. The **Temperature** page appears.
19. Under **Junction temperature range**, specify the **Low temperature** and **High temperature** range for your selected device.
20. Under **Junction temperature and cooling solution settings for PowerPlay power pnalysis**, select **Specify junction temperature** or **Auto compute junction temperature using cooling solution**, and then specify the junction temperature of **Ambient temperature**.
21. Under **Board thermal modeling**, select the **Board thermal model** and type the **Board temperature** (optional). This feature is available only when you select **Auto compute junction temperature using cooling solution**.

    For more information about how to use the operating condition settings, refer to "Operating Conditions" on page 12–7.

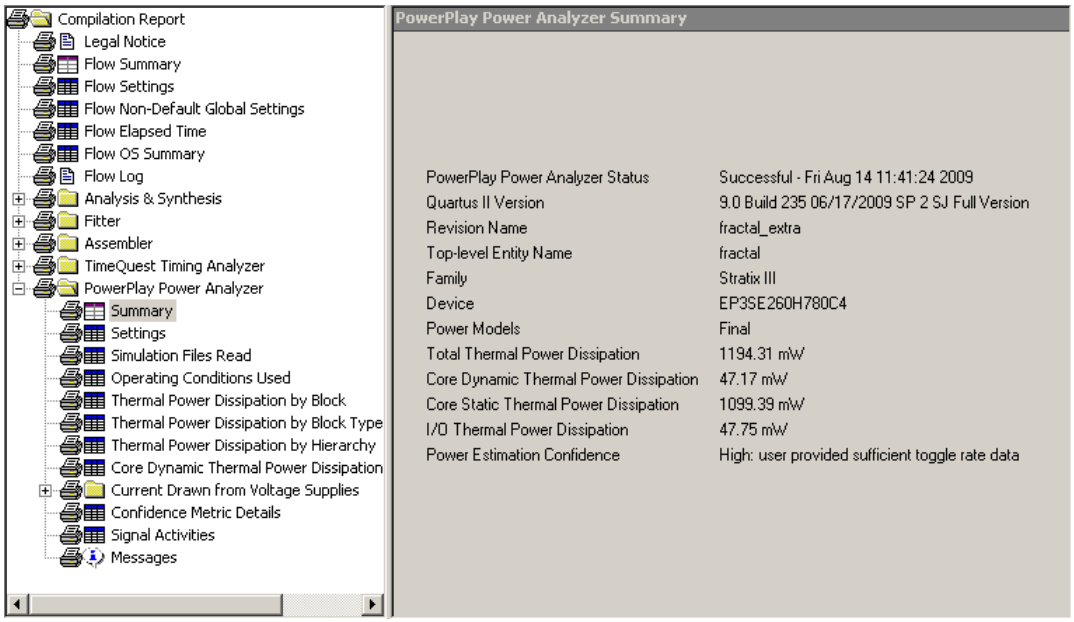
26. Click **OK**.
27. On the Processing menu, click **PowerPlay Power Analyzer Tool**. The **PowerPlay Power Analyzer Tool** dialog box appears.
28. Click **Start** to run the PowerPlay Power Analyzer. Verify all the settings.

    You can also change some of your settings in the **PowerPlay Power Analyzer Tool** dialog box. For example, click **Add Power Input File(s)** to make changes to your input files, or click **Cooling Solution and Temperature** to make changes to your design temperature and cooling solution selection.

29. Click **OK**.

30. 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 12–8).

Figure 12–8. PowerPlay Power Analyzer Summary



PowerPlay Power Analyzer Compilation Report

The PowerPlay Power Analyzer section of the Compilation Report consists of the following sections.

Summary

This section of the report shows the estimated total thermal power consumption of your design. 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 you have provided insufficient toggle rate data, or most of the signal-activity information used for power estimation is from default or vectorless estimation settings. For more information about the input data, refer to the PowerPlay Power Analyzer Confidence Metric report.

Settings

This section of the report shows the PowerPlay Power Analyzer settings information of your design, including the default input toggle rates, operating conditions, and other relevant setting information.

Simulation Files Read

This section of the report lists the simulation output file (**.vcd**) used for power estimation. This section also includes the file ID, file type, entity, VCD start time, VCD end time, the unknown percentage, and the toggle percentage. The unknown percentage 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. This section also shows the entered junction temperature or auto-computed junction temperature that was used during the power analysis. This report is generated only for Arria II GX, Arria GX, Cyclone II, Cyclone III series, Cyclone IV series, HardCopy series, MAX II, Stratix II, Stratix II GX, Stratix III, Stratix IV, and Stratix V 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 you with estimated power consumption for each atom in your 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 estimated thermal dynamic power and thermal static power consumption categorized by design hierarchy. This information is further categorized by the dynamic and static power that was used by the blocks and routing in that hierarchy. This information is very useful when 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, which provides designs with estimated power consumption for each clock domain in the 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 zero MHz.

Current Drawn from Voltage Supplies

This section of the report lists the current that was drawn from each voltage supply. The V_{CCIO} 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 report is generated only for Arria II GX, Cyclone III, Cyclone III LS, Cyclone IV series, MAX II, Stratix III, Stratix IV series, and Stratix V device families.

Transceiver-based devices have multiple voltage supplies, which are V_{CCH}, V_{CCT}, V_{CCR}, V_{CCA}, and V_{CCP}. 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. This section 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 helps you understand how to increase the confidence metric, letting you determine your own confidence in the toggle rate data.

Signal Activities

This section lists toggle rates 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, but can be turned off by turning off the **Write signal activities to report file** option on the P**owerPlay Power Analyzer Settings** page.

Altera recommends turning off the **Write signal activities to report file** option 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 the 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 zero Watts.

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 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 about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook* and *API Functions for Tcl* in Quartus II Help.. 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*.

Running the PowerPlay Power Analyzer from the Command–Line

The separate executable 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 command 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 EPE File, type the following command at a system command prompt:

```
quartus_pow sample --output_epe=sample.csv ↵
```

- To instruct the PowerPlay Power Analyzer to generate a PowerPlay EPE File without performing 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 **.vcd** as input (**sample.vcd**), type the following command at a system command prompt:

```
quartus_pow sample --input_vcd=sample.vcd ↵
```

- To instruct the PowerPlay Power Analyzer to use two **.vcd** files as input files (**sample1.vcd** and **sample2.vcd**), perform glitch filtering on the **.vcd** and use a default input I/O toggle rate of 10,000 transitions per second, type the following command 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 command 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. You can use the Quartus II GUI to specify these options.

The `quartus_pow` executable creates a report file, *<revision name>***.pow.rpt**. You can locate the report file in the main project directory. The report file contains the same information in "PowerPlay Power Analyzer Compilation Report" on page 12–23.

Conclusion

PowerPlay power analysis tools are designed for accurate estimation of power consumption from early design concept through design implementation. You can use the PowerPlay EPE to estimate power consumption during the design concept stage. You can use the PowerPlay Power Analyzer tool to refine power estimations during design implementation. The PowerPlay Power Analyzer produces detailed reports that you can use to optimize designs for lower power consumption and verify that the design is in your power budget.

Document Revision History

Table 12–5 shows the revision history for this chapter.

Table 12–5. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Removed references to the Quartus II Simulator. ■ Updated Table 12–1 on page 12–3, Table 12–2 on page 12–12, and Table 12–3 on page 12–12. ■ Updated Figure 12–3 on page 12–7, Figure 12–4 on page 12–9, and Figure 12–5 on page 12–10.
November 2009	9.1.0	■ Updated "Creating PowerPlay EPE Spreadsheets" on page 12–2 and "Simulation Results" on page 12–9. ■ Added "Signal Activities from Full Post-Fit Netlist (Zero Delay) Simulation" on page 12–17 and "Generating a .vcd from Full Post-Fit Netlist (Zero Delay) Simulation" on page 12–20. ■ Minor changes to "Generating a .vcd from ModelSim Software" on page 12–19. ■ Updated Figure 12–2 on page 12–2 and Figure 12–8 on page 12–23.
March 2009	9.0.0	■ 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.
November 2008	8.1.0	■ Updated for the Quartus II software version 8.1. ■ Replaced Figure 11-3. ■ Replaced Figure 11-14.
May 2008	8.0.0	■ 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.

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

Take an online survey to provide feedback about this handbook chapter.

Section IV. System Debugging Tools

The Altera® Quartus® II design software provides a complete design debugging environment that easily adapts to your specific design requirements. This handbook is arranged in chapters, sections, and volumes that correspond to the major tools available for debugging your designs. 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 is an introduction to System Debugging Tools and includes the following chapters:

- Chapter 13, System Debugging Tools Overview

  This chapter compares the various system debugging tools and explains when to use each of them.

- Chapter 14, Analyzing and Debugging Designs with the System Console

  This chapter describes the System Console Toolkit and compares the different capabilities within the toolkit.

  Chapter 15, Transceiver Link Debugging Using the Quartus II Software

  This chapter explains what functions are available within the Transceiver Toolkit and helps you decide which tool best meets your debugging needs.

- Chapter 16, Quick Design Debugging Using SignalProbe

  This chapter provides detailed instructions about how to use SignalProbe to quickly debug your design.

  Use this chapter to verify your design more efficiently by routing internal signals to I/O pins quickly without affecting the design.

- Chapter 17, Design Debugging Using the SignalTap II Logic Analyzer

  This chapter describes how to debug your FPGA design during normal device operation without the need for external lab equipment. Use this chapter to learn how 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.

- Chapter 18, In-System Debugging Using External Logic Analyzers

  This chapter explains how to use external logic analyzers to debug designs on Altera devices.

- Chapter 19, In-System Modification of Memory and Constants

  This chapter explains how to use the Quartus II In-System Memory Content Editor as part of your FPGA design and verification flow to easily view and debug your design in the hardware lab.

- Chapter 20, Design Debugging Using In-System Sources and Probes

  This chapter provides detailed instructions about how to use the In-System Sources and Probes Editor and Tcl scripting in the Quartus® II software to debug your design.

13. System Debugging Tools Overview

QII53027-10.0.0

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. Altera provides a set of verification tools that are powerful and easy to use, allowing you to get your product to market quickly by minimizing design verification time.

The Quartus® II software provides a portfolio of system design debugging tools for real-time verification of your design. Each tool in the system 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 of the tools available in the system debugging suite and discusses the criteria for selecting the best tool for your design.

System Debugging Tools

Table 12–1 summarizes the tools in the system debugging tool suite that are covered in this section of the *Quartus II Handbook*.

Table 12–1. Available Tools in the In-System Verification Tools Suite (Part 1 of 2)

Tool	Description	Typical Usage
System Console	This is a Tcl console that communicates to hardware modules instantiated into your design. You can use it with the Transceiver Toolkit to monitor or debug your design.	You need to perform system-level debugging. For example, if you have an Avalon-MM slave or Avalon-ST interfaces, you can debug your design at a transaction level. The tool supports JTAG connectivity, but also supports PLI connectivity to a simulation model, as well as TCP/IP connectivity to the target FPGA you wish to debug.
Transceiver Toolkit	This toolkit allows you to calibrate PMA settings so that you can optimize settings for your board.	You need to debug or optimize transceiver link channels in your design.
SignalTap® II Logic Analyzer	This 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.

Table 12–1. Available Tools in the In-System Verification Tools Suite (Part 2 of 2)

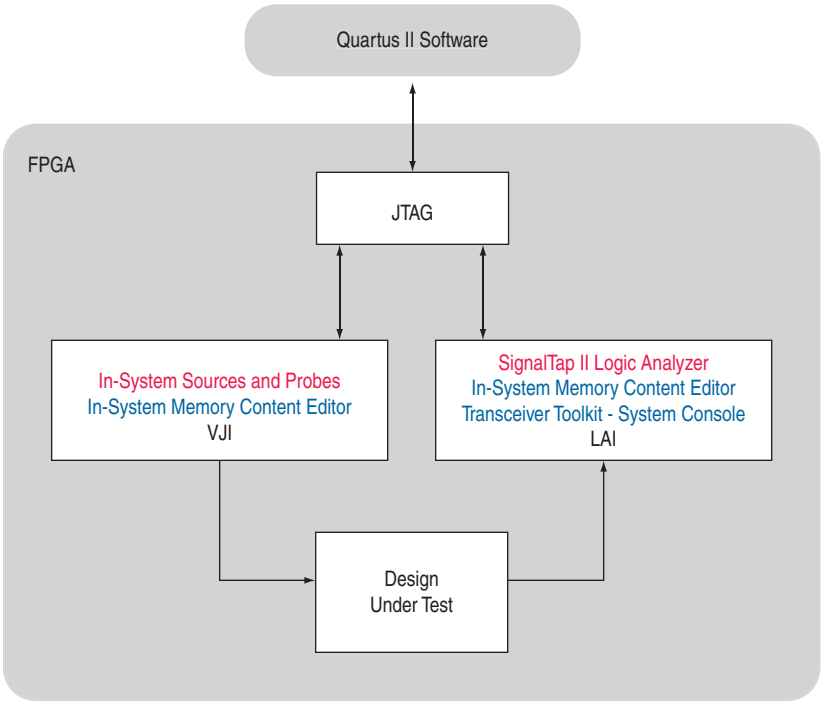
Tool	Description	Typical Usage
Logic Analyzer Interface (LAI)	This tool multiplexes a larger set of signals to a smaller number of spare I/O pins. LAI allows you to select which signals are switched onto the I/O pins over a JTAG connection.	You have limited on-chip memory, and have a large set of internal 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 of the tool.
In-System Sources and Probes	This tool 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.
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 on-chip memory that is not connected to a Nios II processor. You can also use the tool when you do not want to have a Nios II debug core in your system.
Virtual JTAG Interface	This megafunction allows you to communicate with the JTAG interface so that you can develop your own custom applications.	You have custom signals in your design that you want to be able to communicate with.

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. System Console uses JTAG and other interfaces as well. The JTAG resource is shared among all of the on-chip debugging tools.

For all system debugging tools except System Console, 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. For System Console, you must explicitly insert IP cores into your design to enable debugging.

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 12–1 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, System Console, Transceiver Toolkit, 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 12–1).

Figure 12–1. Quartus II Debugging Ecosystem *(Note 1)*



Note to Figure 12–1:

(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 second section, "Stimulus-Capable Tools" on page 13–8, compares System Console, 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 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. 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.

Conversely, 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. You should 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 logic elements (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.

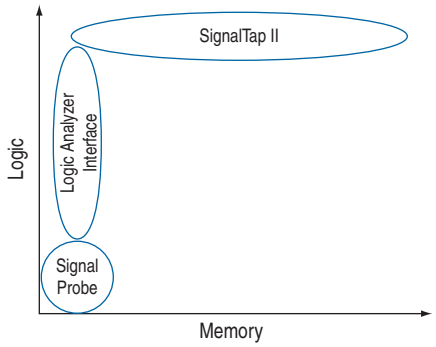
SignalProbe requires very few on-chip resources. Because it requires no JTAG connection, SignalProbe uses no logic or memory resources. SignalProbe uses only routing resources to route an internal signal to a debugging test point.

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 to 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 12–2 shows a conceptual graph of the resource usage of the three analysis tools relative to each other.

Figure 12–2. Resource Usage per Debugging Tool *(Note 1)*

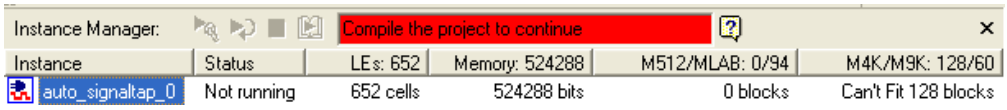


Note to Figure 12–2:

(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 12–3 shows the resource estimation feature for the SignalTap II Logic Analyzer and the LAI.

Figure 12–3. 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 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 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 Logic Analyzer GUI via the JTAG test port.

Usability Enhancements

The SignalTap II 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 files. 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 12–2 compares common debugging features between these tools and provides suggestions about which is the best tool to use for a given feature.

Table 12–2. 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. When used with the LAI to provide you with access to timing mode, you can debug combined streams of data.

Table 12–2. 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, because 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 may be limited by signal integrity issues.
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 Logic Analyzer software. SignalProbe and the LAI require the use of external debugging equipment, such as multimeters, oscilloscopes, or logic analyzers.

Notes to Table 12–2:

(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 interface is well suited for more advanced users who want to develop their own customized JTAG solution.

System Console provides system-level debugging at a transaction level, such as with Avalon-MM slave or Avalon-ST interfaces. You can communicate to a chip through JTAG, PLI connectivity for simulation models, and TCP/IP protocols. System Console is a Tcl console that you use to communicate with hardware modules that you have instantiated into your design.

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 code. 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. This feature 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 content 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, or with System Console. 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.

System Console

System Console is a framework that you can launch from the Quartus II software to start services for performing different kinds of tasks. System Console provides you with Tcl scripts and a GUI to access SOPC Builder modules to perform low-level hardware debugging of your design, as well as identify a module by its path, and open and close a connection to an SOPC module. You can access your design at a system level for purposes of loading, unloading, and transferring designs to multiple devices.

System Console also allows you to access commands that allow you to control how you generate test patterns, as well as verify the accuracy of data generated by test patterns. You can use JTAG debug commands in System Console to verify the functionality and signal integrity of your JTAG chain. You can test clock and reset signals.

You can use System Console to access programmable logic devices on your development board, as well as bring up a board and verify stages of setup. You can also access software running on a Nios II processor, as well as access modules that produce or consume a stream of bytes.

Transceiver Toolkit runs from the System Console framework, and allows you to run automatic tests of your transceiver links for debugging and optimizing your transceiver designs. You can use the Transceiver Toolkit GUI to set up channel links in your transceiver devices, and then automatically run Eye-Q and Auto Sweep testing to view a graphical representation of your test data.

Conclusion

The Quartus II on-chip debugging tool suite allows you to reach debugging closure quickly by providing you a with 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 can increase the level of automation for all of the analysis tools. You can also build virtual front panel applications quickly during the early 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 compilation 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.

Document Revision History

Table 12–3shows the revision history for this application note.

Table 12–3. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	Initial release

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

Take an online survey to provide feedback about this handbook chapter.

14. Analyzing and Debugging Designs with the System Console

QII53028-10.0.0

The System Console performs low-level hardware debugging of SOPC Builder systems. You can use the System Console to access IP cores instantiated in your SOPC Builder system, and for initial bring-up of your printed circuit board and low-level testing. You access the System Console by clicking **Tools** on the main menu of SOPC Builder, and then clicking **System Console**.You can use the System Console in command-line mode or with the GUI.

Command-line mode allows you to run Tcl scripts from the Tcl Console pane, located in the System Console window. You can also run Tcl scripts from the System Console GUI.

The System Console GUI allows you to view a main window with separate panes, including System Explorer, TCL Console, Messages, and Tools panes.

For further details on how to use the System Console GUI, refer to *About Console* in Quartus II Help.

This chapter contains the following sections:

- "Setting Up the System Console"
- "Interactive Help"
- "Using the System Console" on page 14–2
- "System Console Examples" on page 14–17
- "Device Support" on page 14–26

Setting Up the System Console

You set up the System Console according to the hardware you have available in your system. You can access available debugging IP on your system with the System Console. The debug IP allows you to access the running state of your system. The following sections discuss setting up and using debug IP in further detail.

"System Console Examples" on page 14–17 provides you with detailed examples of using the System Console in conjunction with SOPC Builder components. You can download the design examples described in this section from the Altera® website. These design examples demonstrate how to add debug IP blocks to your design and how to connect them before you can use the host application.

Interactive Help

Typing `help help` into the System Console lists all available commands. Typing `help` *<command name>* provides the syntax of individual commands. The System Console provides command completion if you type the beginning letters of a command and then press the Tab key.

The System Console interactive help commands only provide help for enabled services; consequently, typing `help help` does not display help for a plug-in unless you have enabled it.

Using the System Console

The Quartus® II software expands the framework of the System Console by allowing you to start services for performing different types of tasks, as described in the following paragraphs of this chapter. These paragraphs provide Tcl scripting commands, arguments, and a brief description of the command functions.

Many of the System Console commands do not work unless you connect to a system with a programming cable and with the proper debug IP.

SOPC Builder Communications

Table 14–1 describes four SOPC Builder components you can use to facilitate debugging with the System Console. When connected to the System Console, these components enable you to send commands and receive data..

Table 14–1. SOPC Builder Components for Communication with the System Console *(Note 1)*

Component Name	Debugs Components with the Following Interface Types
The Nios® II processor with JTAG debug enabled	Components that include an Avalon® Memory-Mapped (Avalon-MM) slave interface. The JTAG debug module can also control the Nios II processor for debug functionality, including starting, stopping, and stepping the processor.
JTAG to Avalon master bridge	Components that include an Avalon-MM slave interface
Avalon Streaming (Avalon-ST) JTAG Interface	Components that include an Avalon-ST interface
JTAG UART	The JTAG UART is an Avalon-MM slave device that can be used in conjunction with the System Console to send and receive byte streams.

Note to Table 14–1:

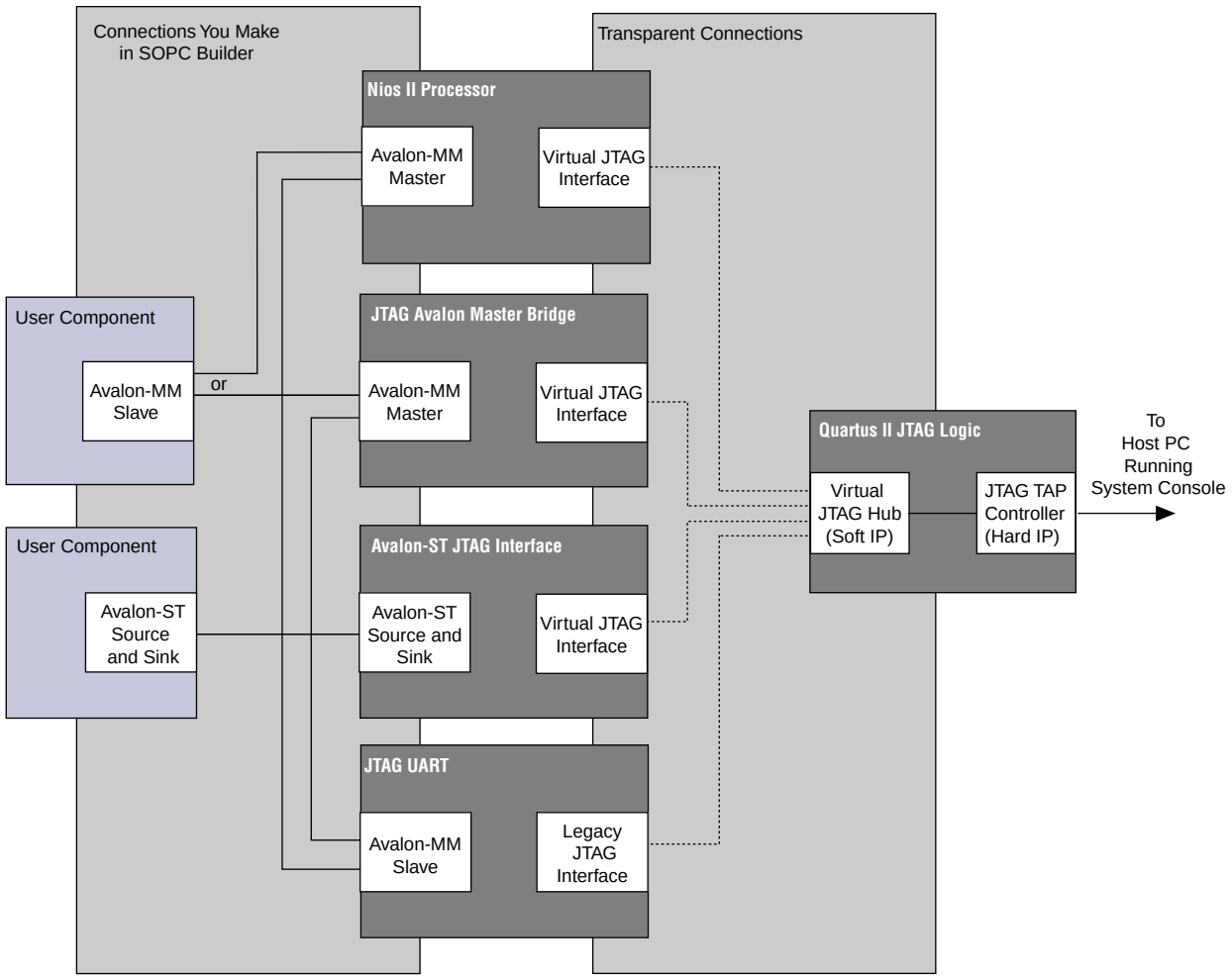
(1) The System Console can also send and receive byte streams from any SLD node whether it is instantiated in SOPC Builder component provided by Altera®, a custom component, or part of your Quartus II project; however, this approach requires detailed knowledge of the JTAG commands.

For further information about SOPC Builder components, refer to the following web pages and documents:

- The Nios II Processor product web page
- *SPI Slave/JTAG to Avalon Master Bridge Cores* chapter in volume 5 of the *Quartus II Handbook*
- *Avalon-ST JTAG Interface Core* chapter in volume 5 of the *Quartus II Handbook*
- *SLD Virtual JTAG MegaFunction User Guide*
- *JTAG UART Core* chapter in volume 5 of the *Quartus II Handbook*

Figure 14–1 illustrates examples of interfaces of the components that the System Console can use.

Figure 14–1. Example Interfaces (Paths) the System Console Can Use to Send Commands



Altera recommends that you also include the following components in your system:

- On-chip memory
- JTAG UART
- System ID core

The System Console provides many different types of services. Different modules can provide the same type of service. For example, both the Nios II processor and the JTAG to Avalon Bridge master provide the master service; consequently, you can use the master commands to access both of these modules.

If your system includes a Nios II/f core with a data cache it may complicate the debugging process. If you suspect that writes to memory from the data cache at nondeterministic intervals are overwriting data written by the System Console, you can disable the cache of the Nios II/f core while debugging.

You can start the System Console from a Nios II command shell.

1. Choose **All Programs > Altera > Nios II EDS** <*version*> **Command Shell** (Windows Start menu) to run a Nios II command shell.
2. To start the System Console, type the following command:

```
system-console ↵
```

You can customize your System Console environment by adding commands to the configuration file called **system_console_rc.tcl**. This file can be located in either of two places:

- <*quartus_install_dir*>**/sopc_builder/system_console_macros/system_console_rc.tcl**, known as the global configuration file, which affects all users of the system
- <$HOME>**/system_console/system_console_rc.tcl**, known as the user configuration file, which only affects the owner of that home directory

On startup, the System Console automatically runs any Tcl commands in these files. The commands in the global configuration file runs first, followed by the commands in the user configuration file.

Console Commands

The console commands enable testing. You can use console commands to identify a module by its path, and to open and close a connection to it. The `path` that identifies a module is the first argument to most of the other System Console commands. To exercise a module, follow these steps:

1. Identify a module by specifying the `path` to it, using the `get_service_paths` command.
2. Open a connection to the module using the `open_service` command.
3. Run Tcl and System Console commands to use a debug module that you insert to test another module of interest.
4. Close a connection to a module using the `close_service` command.

Table 14–2 describes the syntax of the console commands.

Table 14–2. Console Commands (Part 1 of 2)

Command	Arguments	Function
`get_service_types`	-	Returns a list of service types that the System Console manages. Examples of service types include master, bytestream, processor, sld, jtag_debug, device, and plug-in.
`get_service_paths`	*<service_type_name>*	Returns a list of paths to nodes that implement the requested service type.
`open_service`	*<service_type_name>* *<service_path>*	Opens the service type specified.
`close_service`	*<service_type_name>* *<service_path>*	Closes the service type specified.
`is_service_open`	*<service_type_name>* *<service_path>*	Returns 1 if the service type provided by the path is open, 0 if the service type is closed.
`get_services_to_add`	-	Returns a list of all services that are instantiable with the add_service command.

Table 14–2. Console Commands (Part 2 of 2)

Command	Arguments	Function
add_service	<service-type> <instance-name> <optional-parameters>	Add a service of the specified service type with the given instance name. Run get_services_to_add to retrieve a list of instantiable services. This command returns the path where the service was added. Run help add_service <service-type> to get specific help about that service-type, including any arguments that might be required for that service.
get_version	-	Returns the current System Console version and build number.
add_help	<command> <help-text>	Adds help text for a given command. Use this when you write a Tcl script and then want to provide help for others to use the script.

Plug-ins

Plug-in commands allow you to customize how you use the System Console services, and are enabled by default. Table 14–3 lists Plug-in commands.

Table 14–3. Plug-in Commands

Command	Arguments	Function
Plug-in Service Type Commands		
plugin_enable	<plugin-path>	Enables the plug-in specified by the path. After a plug-in is enabled, you can retrieve the <service-path> and <service_type_name> for additional services using the get_service_paths and get_service_types commands, respectively.
plugin_disable	<plugin-path>	Disables the plug-in specified by the path.
is_plugin_enabled	<plugin-path>	Returns a non-zero value when the plug-in at the specified path is enabled.

Design Service Commands

Design Service commands allow you to use the System Console services to load and work with your design at a system level. Table 14–4 lists Design Service commands.

Table 14–4. Design Service Commands

Command	Arguments	Function
`design_load`	*<quartus-project-path>* or *<qpf-file-path>*	Loads a model of a Quartus II design into the System Console. For example, if your **.qpf** file is in **c:/projects/loopback**, type the following command: `design_load {c:\projects\loopback\}`
`design_instantiate`	*<design-path>* *<instance-name>*	Allows you to apply the same design to multiple devices.
`design_link`	*<design-instance-path>* *<device-service-path>*	Links a Quartus II logical design with a physical device. For example, you can link a Quartus II design called **2c35_quartus_design** to a 2c35 device. After you create this link, the System Console creates the appropriate correspondences between the logical and physical submodules of the Quartus II project. Example 14–1 on page 14–17 shows a transcript illustrating the `design_load` and `design_link` commands. Note that the System Console does not verify that the link is valid, so that if you create an incorrect link, the System Console does not report an error.

Data Pattern Generator Commands

Data Pattern Generator commands allow you control data patterns that you generate for testing, debugging, and optimizing your design. You must first insert debug IP to enable these commands.Table 14–5 lists Data Pattern Generator commands.

Table 14–5. Data Pattern Generator Commands (Part 1 of 3)

Command	Arguments	Function
`data_pattern_generator_start`	*<service-path>*	Starts the data pattern generator.
`data_pattern_generator_stop`	*<service-path>*	Stops the data pattern generator.
`data_pattern_generator_is_generating`	*<service-path>*	Returns non-zero if the generator is running.
`data_pattern_generator_inject_error`	*<service-path>*	Injects a 1-bit error into the generator's output.

Table 14–5. Data Pattern Generator Commands (Part 2 of 3)

Command	Arguments	Function
`data_pattern_generator_set_pattern`	*<service-path>* *<pattern-name>*	Sets the output pattern specified by the *<pattern-name>*. In all, 6 patterns are available, 4 are pseudo-random binary sequences (PRBS) and 2 are high and low frequency. The following pattern names are defined: ■ PRBS7 ■ PRBS15 ■ PRBS23 ■ PRBS31 ■ HF–outputs a high frequency, constant pattern of alternating 0s and 1s ■ LF–outputs a low frequency, constant pattern of 10b'1111100000 for 10-bit symbols and 8b'11110000 for 8-bit symbols
`data_pattern_generator_get_pattern`	*<service-path>*	Returns currently selected output pattern.
`data_pattern_generator_get_available_patterns`	*<service-path>*	Returns a list of available data patterns by name.
`data_pattern_generator_enable_preamble`	*<service-path>*	Enables the preamble mode at the beginning of generation.
`data_pattern_generator_disable_preamble`	*<service-path>*	Disables the preamble mode at the beginning of generation.
`data_pattern_generator_is_preamble_enabled`	*<service-path>*	Returns a non-zero value if preamble mode is enabled.
`data_pattern_generator_set_preamble_word`	*<service-path>* *<preamble-word>*	Sets the preamble word. (Could be 32-bit or 40-bit).
`data_pattern_generator_get_preamble_word`	*<service-path>*	Gets the preamble word.

Table 14–5. Data Pattern Generator Commands (Part 3 of 3)

Command	Arguments	Function
data_pattern_generator_set_preamble_beats	<service-path> <number-of-preamble-beats>	Sets the number of beats to send out the in the preamble word.
data_pattern_generator_get_preamble_beats	<service-path>	Returns the currently set number of beats to send out in the preamble word.

Data Pattern Checker Commands

Data Pattern Checker commands allow you verify the data patterns that you generate. You must first insert debug IP to enable these commands. Table 14–6 lists Data Pattern Checker commands.

Table 14–6. Data Pattern Checker Commands

Command	Arguments	Function
data_pattern_checker_start	<service-path>	Starts the data pattern checker.
data_pattern_checker_stop	<service-path>	Stops the data pattern checker.
data_pattern_checker_is_checking	<service-path>	Returns a non-zero value if the checker is running.
data_pattern_checker_is_locked	<service-path>	Returns non-zero if the checker is locked onto the incoming data.
data_pattern_checker_set_pattern	<service-path> <pattern-name>	Sets the expected pattern to the one specified by the <pattern-name>.
data_pattern_checker_get_pattern	<service-path>	Returns the currently selected expected pattern by name.
data_pattern_checker_get_available_patterns	<service-path>	Returns a list of available data patterns by name.
data_pattern_checker_get_data	<service-path>	Returns the correct and incorrect_counts _of_data received, providing the number of bits and the number of errors.
data_pattern_checker_reset_counters	<service-path>	Resets the bit and error counters inside the checker.

Programmable Logic Device (PLD) Commands

The PLD commands provide access to programmable logic devices on your board. Before using these commands, you must identify the path to the programmable logic device on your board using the `get_service_paths` command described in Table 14–2.

Table 14–7 describes the PLD commands.

Table 14–7. PLD Commands

Command	Arguments	Function
`device_download_sof`	*<device_path>* *<sof_file>*	This command loads the specified SRAM object file (**.sof)** file to the device specified by the path.
`device_load_jdi`	*<device_path>* *<jdi_file>*	This command renames the Tcl interface layer's nodes to the names specified in the JTAG debug interface (**.jdi)** file, making your design easier to understand.

Board Bring-Up Commands

The board bring-up commands allow you to test your system. These commands are presented in the order that you would use them during board bring-up, including the following setup work flow:

1. Verify JTAG connectivity
2. Verify the clock and reset signals
3. Verify memory and other peripheral interfaces
4. Verify basic Nios II processor functionality

The System Console is intended for debugging the basic hardware functionality of your Nios II processor, including its memories and pinout. Once the hardware is functioning correctly, you can refer to the *Nios II Software Build Tool Reference* in the *Nios II Software Developer's Handbook* for further software debugging. If you are writing device drivers, you may want to use the System Console and the Nios II software build tools together to debug your code.

JTAG Debug Commands

You can use these commands to verify the functionality and signal integrity of your JTAG chain. Your JTAG chain must be functioning correctly to debug the rest of your system. To verify signal integrity of your JTAG chain, Altera recommends that you provide an extensive list of byte values. Table 14–8 lists these commands.

Table 14–8. JTAG Commands

Command	Arguments	Function
`jtag_debug_loop`	*<path> <list_of_byte_values>*	Loops the specified list of bytes through a loopback of `tdi` and `tdo` of a system-level debug (SLD) node. Returns the list of byte values in the order that they were received. Blocks until all bytes are received. Byte values are given with the 0x (hexadecimal) prefix and delineated by spaces.
`jtag_debug_reset_system`	`<service-path>`	Issues a reset request to the system.

Clock and Reset Signal Commands

The next stage of board bring-up tests the clock and reset signals. Table 14–9 lists the three commands to verify these signals. You can use these commands to verify that your clock is toggling and that the reset signal has the expected value.

Table 14–9. Clock and Reset Commands

Command	Argument	Function
`jtag_debug_sample_clock`	*<service-path>*	Returns the value of the clock signal of the system clock that drives the module's system interface. The clock value is sampled asynchronously; consequently, you may need to sample the clock several times to guarantee that it is toggling.
`jtag_debug_sample_reset`	*<service-path>*	Returns the value of the reset signal of the system reset that drives the module's system interface.
`jtag_debug_sense_clock`	*<service-path>*	Returns the result of a sticky bit that monitors for system clock activity. If the clock has ever toggled, the bit is 1. Returns `true` if the bit has ever toggled and otherwise returns `false`. The sticky bit is reset to 0 on read.

Avalon-MM Commands

These commands allow you to access the memory-mapped or SLD modules that include Avalon-ST slave interfaces. You can read or write the Avalon-MM interfaces using the master read and write commands, but the Avalon-MM Slave must be connected to an Avalon MM-Master that you can control from the host.

PLI Master commands listed in Table 14–12 provide bytestream and master services over a PLI connection to a simulator. TCL Master commands provide bytestream and master services over TCP IP.

Additionally, you can use the SLD commands to shift values into the instruction and data registers of SLD nodes and read the previous value. Table 14–10 lists these commands.

Table 14–10. Module Commands *(Note 1)*

Command	Arguments	Function
Avalon-MM Master Commands		
`master_write_memory`	`<service-path> <address> <list_of_byte_values>`	Writes the specified data to the specified path and address. Values are given in hexadecimal format with the 0x prefix and delineated by spaces.
`master_write_8`	`<service-path> <address> <list_of_byte_values>`	
`master_write_16`	`<service-path> <address> <list_of_16_bit_words>`	
`master_write_32`	`<service-path> <address> <list_of_32_bit_words>`	
`master_read_memory`	`<service-path> <base_address> <size_in_bytes>`	Returns a list of read values.
`master_read_8`	`<service-path> <base_address> <size_in_bytes>`	
`master_read_16`	`<service-path> <base_address> <size_in_multiples_of_16_bits>`	
`master_read_32`	`<service-path> <base_address> <size_in_multiples_of_32_bits>`	
SLD Commands		
`sld_access_ir`	`<service-path> <value> <timeout>` (in µseconds)	Shifts the instruction value into the instruction register of the specified node. Returns the previous value of the instruction. If the timeout value is set to 0, the operation never times out. A suggested starting value for `timeout` is 1000 µseconds.
`sld_access_dr`	`<service-path> <size_in_bits> <timeout>` (in µseconds), `<list_of_byte_values>`	Shifts the byte values into the data register of the SLD node up to the size in bits specified. If the `timeout` value is set to 0, the operation never times out. Returns the previous contents of the data register. A suggested starting value for `timeout` is 1000 µseconds.
`sld_lock`	`sld_lock <service-path> <timeout>` (in mseconds)	Locks the SLD chain to guarantee exclusive access. If the SLD chain is already locked, tries for `<timeout>` mseconds before returning -1, indicating an error. Returns 0 if successful.
`sld_unlock`	`sld_unlock <service-path>`	Unlocks the SLD chain.

Note to Table 14–10:

(1) Transfers performed in 16- and 32-bit sizes are packed in little endian format.

Processor Commands

These commands allow you to start, stop, and step through software running on a Nios II processor. They also allow you to read and write the processor's registers. Table 14–11 lists the commands.

Table 14–11. Processor Commands

Command	Arguments	Function
`elf_download`	*<processor-service-path>* *<master-service-path>* *<elf-file-path>*	Downloads the given executable and loadable format (ELF) file to memory using the specified master service. Sets the processor's program counter to the **.elf** entry point.
`processor_in_debug_mode`	*<service-path>*	Returns a non-zero value if the processor is in debug mode.
`processor_reset`	*<service-path>*	Resets the processor and places it in debug mode.
`processor_run`	*<service-path>*	Puts the processor into existing debug mode.
`processor_stop`	*<service-path>*	Puts the processor into debug mode.
`processor_step`	*<service-path>*	Executes one assembly instruction.
`processor_get_register_names`	*<service-path>*	Returns a list with the names of all of the processor's accessible registers.
`processor_get_register`	*<service-path>* *<register_name>*	Returns the value of the specified register.
`processor_set_register`	*<service-path>* *<register_name>*	Sets the value of the specified register.

Bytestream Commands

These commands provide access to modules that produce or consume a stream of bytes. You can use the bytestream service to communicate directly to IP that provides bytestream interfaces, such as the Altera JTAG UART. Table 14–12 lists the commands.

Table 14–12. Bytestream Commands

Command	Arguments	Function
`bytestream_send`	*<service-path>* *<list_of_byte_values>*	Sends the list of byte values on the specified path. Values are in hexadecimal format and delineated by spaces.
`bytestream_receive`	*<service-path>* *<number_of_bytes>*	Limits the maximum number of bytes to return.

Transceiver Toolkit Commands

You run the Transceiver Toolkit from the System Console window. Alternatively, you can open the Transceiver Toolkit from the Tools menu in the Quartus II software. You can debug, monitor, and optimize the transceiver channel links in your design with either Tcl scripts, or you can launch the Transceiver Toolkit GUI from the System Console Tools menu. The GUI for the Transceiver Toolkit provides a graphical representation of automatic tests that you run. It also presents graphical control panels to change PMA settings of channels, start and stop generators, and checkers.

Table 14–13 through Table 14–17 lists the Transceiver Toolkit commands..

Table 14–13. Transceiver Toolkit Channel_rx Commands

Command	Arguments	Function
`transceiver_channel_rx_get_data`	<*service-path*>	Returns a list of the current checker data.The results are in the order: number of bits, number of errors, bit error rate.
`transceiver_channel_rx_get_dcgain`	<*service-path*>	Gets the DC gain value on the receiver channel.
`transceiver_channel_rx_get_eqctrl`	<*service-path*>	Gets the equalization control value on the receiver channel.
`transceiver_channel_rx_get_pattern`	<*service-path*>	Returns the current data checker pattern by name.
`transceiver_channel_rx_is_checking`	<*service-path*>	Returns non-zero if the checker is running.
`transceiver_channel_rx_is_locked`	<*service-path*>	Returns non-zero if the checker is locked onto the incoming data.
`transceiver_channel_rx_reset_counters`	<*service-path*>	Resets the bit and error counters inside the checker.
`transceiver_channel_rx_set_dcgain`	<*service-path*> `<dc_gain setting>`	Sets the DC gain value on the receiver channel.
`transceiver_channel_rx_set_eqctrl`	<*service-path*> `<eqctrl setting>`	Sets the equalization control value on the receiver channel.
`transceiver_channel_rx_start_checking`	<*service-path*>	Starts the checker.
`transceiver_channel_rx_stop_checking`	<*service-path*>	Stops the checker.
`transceiver_channel_rx_get_eyeq_phase_step`	<*service-path*>	Get the current phase step of the specified channel.
`transceiver_channel_rx_is_eyeq_enabled`	<*service-path*>	Get whether the EyeQ feature is enabled on the specified channel.
`transceiver_channel_rx_set_eyeq_enabled`	<*service-path*> `<disable(0)/enable(1)>`	Enable/disable the EyeQ feature on the specified channel.
`transceiver_channel_rx_set_eyeq_phase_step`	<*service-path*> `<phase step>`	Set the phase step of the specified channel.
`transceiver_channel_rx_set_word_aligner_enabled`	<*service-path*> `<disable(0)/enable(1)>`	Enable/disable the word aligner of the specified channel.
`transceiver_channel_rx_is_word_aligner_enabled`	<*service-path*> `<disable(0)/enable(1)>`	Enable/disable the word aligner of the specified channel.

Table 14–14. Transceiver Toolkit Channel _tx Commands (Part 1 of 2)

Command	Arguments	Function
transceiver_channel_tx_disable_preamble	<service-path>	Disables the preamble mode at the beginning of generation.
transceiver_channel_tx_enable_preamble	<service-path>	Enables the preamble mode at the beginning of generation.
transceiver_channel_tx_get_number_of_preamble_beats	<service-path>	Returns the number of preamble beats.
transceiver_channel_tx_get_pattern	<service-path>	Returns the currently set pattern.
transceiver_channel_tx_get_preamble_word	<service-path>	Returns the currently set preamble word.
transceiver_channel_tx_get_preemph0t	<service-path>	Gets the pre-emphasis pre-tap value on the transmitter channel.
transceiver_channel_tx_get_preemph1t	<service-path>	Gets the pre-emphasis first post-tap value on the transmitter channel.
transceiver_channel_tx_get_preemph2t	<service-path>	Gets the pre-emphasis second post-tap value on the transmitter channel.
transceiver_channel_tx_get_vodctrl	<service-path>	Gets the VOD control value on the transmitter channel.
transceiver_channel_tx_inject_error	<service-path>	Injects a 1-bit error into the generator's output.
transceiver_channel_tx_is_generating	<service-path>	Returns non-zero if the generator is running.
transceiver_channel_tx_is_preamble_enabled	<service-path>	Returns non-zero if preamble mode is enabled.
transceiver_channel_tx_set_number_of_preamble_beats	<service-path> <number-of-preamble-beats>	Sets the number of beats to send out the preamble word.
transceiver_channel_tx_set_pattern	<service-path> <pattern-name>	Sets the output pattern to the one specified by the pattern name.
transceiver_channel_tx_set_preamble_word	<service-path> <preamble-word>	Sets the preamble word to be sent out.
transceiver_channel_tx_set_preemph0t	<service-path> <preemph0t value>	Sets the pre-emphasis pre-tap value on the transmitter channel.
transceiver_channel_tx_set_preemph1t	<service-path> <preemph1t value>	Sets the pre-emphasis first post-tap value on the transmitter channel.
transceiver_channel_tx_set_preemph2t	<service-path> <preemph2t value>	Sets the pre-emphasis second post-tap value on the transmitter channel.

Table 14–14. Transceiver Toolkit Channel _tx Commands (Part 2 of 2)

Command	Arguments	Function
`transceiver_channel_tx_set_vodctrl`	<*service-path*> <vodctrl value>	Sets the VOD control value on the transmitter channel.
`transceiver_channel_tx_start_generation`	<*service-path*>	Starts the generator.
`transceiver_channel_tx_stop_generation`	<*service-path*>	Stops the generator.

Table 14–15. Transceiver Toolkit Debug_Link Commands

Command	Arguments	Function
`transceiver_debug_link_get_pattern`	<*service-path*>	Gets the currently set pattern the link uses to run the test.
`transceiver_debug_link_is_running`	<*service-path*>	Returns non-zero if the test is running on the link.
`transceiver_debug_link_set_pattern`	<*service-path*><data pattern>	Sets the pattern the link uses to run the test.
`transceiver_debug_link_start_running`	<*service-path*>	Starts running a test with the currently selected test pattern.
`transceiver_debugf_link_stop_running`	<*service-path*>	Stops running the test.

Table 14–16. Transceiver Toolkit Reconfig_analog Commands (Part 1 of 2)

Command	Arguments	Function
`transceiver_reconfig_analog_get_logical_channel_address`	<*service-path*>	Gets the transceiver logical channel address currently set.
`transceiver_reconfig_analog_get_rx_dcgain`	<*service-path*>	Gets the DC gain value on the receiver channel specified by the current logical channel address.
`transceiver_reconfig_analog_get_rx_eqctrl`	<*service-path*>	Gets the equalization control value on the receiver channel specified by the current logical channel address.
`transceiver_reconfig_analog_get_tx_preemph0t`	<*service-path*>	Gets the pre-emphasis pre-tap value on the transmitter channel specified by the current logical channel address.
`transceiver_reconfig_analog_get_tx_preemph1t`	<*service-path*>	Gets the pre-emphasis first post-tap value on the transmitter channel specified by the current logical channel address.
`transceiver_reconfig_analog_get_tx_preemph2t`	<*service-path*>	Gets the pre-emphasis second post-tap value on the transmitter channel specified by the current logical channel address.

Table 14–16. Transceiver Toolkit Reconfig_analog Commands (Part 2 of 2)

Command	Arguments	Function
`transceiver_reconfig_analog_get_tx_vodctrl`	*<service-path>*	Gets the VOD control value on the transmitter channel specified by the current logical channel address.
`transceiver_reconfig_analog_set_logical_channel_address`	*<service-path>* <logical channel address>	Sets the transceiver logical channel address.
`transceiver_reconfig_analog_set_rx_dcgain`	*<service-path>* <dc_gain value>	Sets the transceiver logical channel address.
`transceiver_reconfig_analog_set_rx_eqctrl`	*<service-path>* <eqctrl value>	Sets the equalization control value on the receiver channel specified by the current logical channel address.
`transceiver_reconfig_analog_set_tx_preemph0t`	*<service-path>* <preemph0t value>	Sets the pre-emphasis pre-tap value on the transmitter channel specified by the current logical channel address.
`transceiver_reconfig_analog_set_tx_preemph1t`	*<service-path>* <preemph1t value>	Sets the pre-emphasis first post-tap value on the transmitter channel specified by the current logical channel address.
`transceiver_reconfig_analog_set_tx_preemph2t`	*<service-path>* <preemph2t value>	Sets the pre-emphasis second post-tap value on the transmitter channel specified by the current logical channel address.
`transceiver_reconfig_analog_set_tx_vodctrl`	*<service-path>* <vodctrl value>	Sets the VOD control value on the transmitter channel specified by the current logical channel address.

Table 14–17. Eye Monitor Tcl Commands (Part 1 of 2)

Command	Arguments	Function
`alt_xcvr_custom_is_word_aligner_enabled`	*<service-path>* <disable(0)/enable(1)>	Enable/disable the word aligner of the previously specified channel.
`alt_xcvr_custom_set_word_aligner_enabled`	*<service-path>* <disable(0)/enable(1)>	Enable/disable the word aligner of the previously specified channel.
`alt_xcvr_reconfig_eye_viewer_get_logical_channel_address`	*<service-path>*	Gets the logical channel address on which other alt_reconfig_eye_viewer commands will use to apply.

Table 14–17. Eye Monitor Tcl Commands (Part 2 of 2)

Command	Arguments	Function
`alt_xcvr_reconfig_eye_viewer_get_phase_step`	*<service-path>*	Get the current phase step of the previously specified channel.
`alt_xcvr_reconfig_eye_viewer_is_enabled`	*<service-path>*	Get whether the EyeQ feature is enabled on the previously specified channel.
`alt_xcvr_reconfig_eye_viewer_set_enabled`	*<service-path>* `<disable(0)/enable(1)>`	Enable/disable the EyeQ feature on the previously specified channel.
`alt_xcvr_reconfig_eye_viewer_set_logical_channel_address`	*<service-path>*	—
`alt_xcvr_reconfig_eye_viewer_set_phase_step`	*<service-path>* `<phase step>`	Set the phase step of the previously specified channel.

Example 14–1 shows how to load and link a Quartus II design set.

Example 14–1. Loading and Linking a Design

```
% get_service_paths device
{/connections/USB-Blaster [USB-0]/EP2C35}

% set device_path [lindex [get_service_paths device] 0]
/connections/USB-Blaster [USB-0]/EP2C35

% design_load /projects/9.1/standard
QuartusDesignFactory elaborating \projects\9.1\standard
QuartusDesignFactory found SOF File at NiosII_cycloneII_2c35_standard.sof
QuartusDesignFactory found JDI File at NiosII_cycloneII_2c35_standard.jdi
QuartusDesignFactory found SOPC Info File at
\projects\9.1\standard\NiosII_cycloneII_2c35_standard_sopc.sopcinfo

% set design_path [lindex [get_service_paths design] 0]
/designs/standard

% design_link $design_path $device_path
Created a link from /designs/standard to /connections/USB-Blaster [USB-0]/EP2C35.
Created a link from /designs/standard/NiosII
cycloneII_2c35_standard_sopc.sopcinfo/cpu.data_master to /connections/USB-Blaster
[USB-0]/EP2C35/cpu.
Created a link from
/designs/standard/NiosII_cycloneII_2c35_standard_sopc.sopcinfo/cpu.data_master/jtag_
uart.avalon_jtag_slave to /connections/USB-Blaster [USB-0]/EP2C35/jtag_uart
```

System Console Examples

This section provides three SOPC Builder system examples to show you how to use the System Console. The **System-Console.zip** file contains design files for the first two example systems. This zip file includes files for both the Nios II Development Kit Cyclone® II Edition and the Nios II Development Kit Stratix® II Edition.

You can download the design files for the example designs from the Altera website. A link to the design files appears next to this document on the User Guide web page.

The first example Tcl script creates a LED light show on your board. The SOPC Builder system for this example includes two modules: a JTAG to Avalon master bridge and a PIO core. The JTAG to Avalon master bridge provides a connection between your development board and SOPC Builder system via serial peripheral interface (SPI). The PIO module provides a memory-mapped interface between an Avalon-MM slave port and general-purpose IO ports.

For more information about these components refer to the *Embedded Peripherals IP User Guide*.

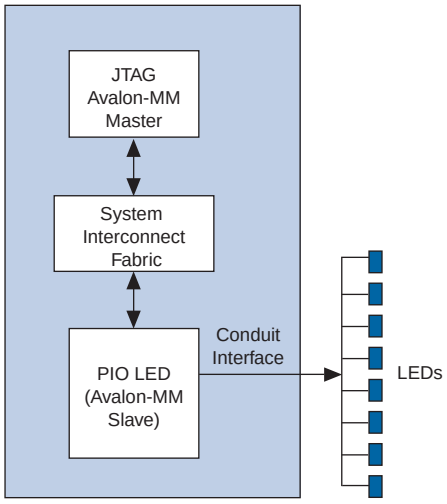
The first example program sends a series of `master_write_8` commands to the JTAG Avalon master bridge. The JTAG Avalon master sends these commands to the Avalon-MM slave port of the PIO module. The PIO I/O ports connect to FPGA pins that are, in turn, connected to the LEDs on your development board. The write commands to the PIO Avalon-MM slave port result in the light show.

The instructions for these examples assume some familiarity with the Quartus II and SOPC Builder software.

LED Light Show Example

Figure 14–2 illustrates the SOPC Builder system for the first example.

Figure 14–2. SOPC Builder System for Light Show Example



To build this example system, complete the following steps:

1. On your host computer file system, locate the following directory: <*Nios II EDS install path*>**\examples**<*verilog or vhdl*>\<*board version*>**\standard**. Each development board has a VHDL and Verilog HDL version of the design. You can use either of these design examples.
2. Copy the **standard** directory to a new location. By copying the design files, you avoid corrupting the original design and avoid issues with file permissions. This document refers to the newly created directory as the c:\<*projects*>**\standard** directory.

Refer to the Altera website for information on different board kits available from Altera.

3. Copy the **System_Console.zip** file to the c:\< *projects*>**standard** directory and unzip it. Specific directories may be created for specific Altera development boards.
4. Choose **All Programs > Altera > Nios II EDS** <*version*> **Command Shell** (Windows Start menu) to run a Nios II command shell.
5. Change to the directory for your board.
6. To program your board with the **.sof** file, type the following command in the Nios II command shell:

   `nios2-configure-sof` <*sof_name*>`.sof` ↵

   If your development board includes more than one JTAG cable, you must specify which cable you are communicating with as an argument to the `nios2-configure-sof` <*sof_name*>`.sof` command. To do so, type the following commands:

   a. `jtagconfig` ↵

Figure 14–3. jtagconfig Output

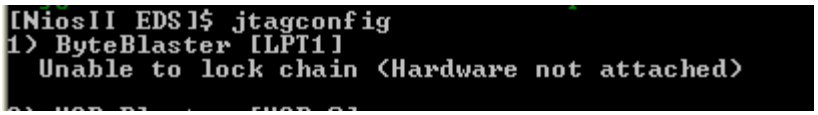


Figure 14–3 gives sample output from the `jtagconfig` command. This output shows that the active JTAG cable is number 2. Substitute the number of your JTAG for the <*cable_number*> variable in the following command:

   b. `nios2-configure-sof -c` <*cable_number*> <*sof_name*>`.sof` ↵

7. You can then run the LED light show example by typing the following command:

   `system-console --script=led_lightshow.tcl` ↵

8. You can see the LEDs performing a running light demonstration. Press Ctrl+C to stop the LED light show.
9. To see the commands that this script runs, open the **led_lightshow.tcl** file in your **jtag_pio**_<*cii_or_sii*> directory.

JTAG Examples

Two JTAG examples are described below. The first JTAG example gives you some practice working with the System Console as an interactive tool. The second example verifies that the clock is toggling.

Verify JTAG Chain

In this example, you verify the JTAG chain on you board. To run this example, complete the following steps:

1. Choose **All Programs > Altera > Quartus II** <*version*> (Windows Start menu) to run the Quartus II software. Open the Quartus II project file, **jtag_pio.qpf** or **jtag_pio_sii.qpf**.
2. On the Tools menu, click **SOPC Builder**.
3. On the SOPC Builder Tools menu, click **System Console**.
4. Set the path to the `jtag_debug` service by typing the following command:

   ```
   set jd_path [lindex [get_service_paths jtag_debug] 0] ↵
   ```

   The `get_service_paths` command always returns a list, even if the list has a single item; consequently, you must index into the list using the `lindex` command. In this case, the variable *jd_path* is assigned the string that is the 0th element of the list.
5. Open the `jtag_debug` service by typing the following command:

   ```
   open_service jtag_debug $jd_path ↵
   ```
6. Set up a list of byte values to test the chain by typing the following command:

   ```
   set values [list 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55
   0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55]↵
   ```
7. Loop the values by typing the following command:

   ```
   jtag_debug_loop $jd_path $values ↵
   ```

   If the `jtag_debug_loop` command is successful, you should see the values that you sent reflected in the System Console. Figure 14–4 shows the transcript from this interactive session.

Figure 14–4. The jtag_debug_loop Command

```
% set jd_path [lindex [get_service_paths jtag_debug] 0]
/root/CONNECTIONS/USB-Blaster [USB-0]/EP2C35/[MFG:110 ID:13

% open_service jtag_debug $jd_path


% set values [list 0xaa 0x55 0xaa 0x55 0xaa 0x55 0xaa 0x55 (
0xaa 0x55 0xaa 0x55]
```

8. Close the `jtag_debug` service by typing the following command:

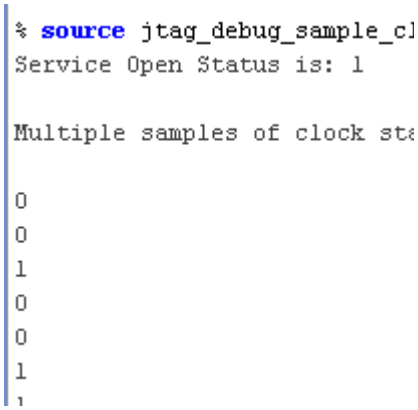
   ```
   close_service jtag_debug $jd_path↵
   ```

This example provides the beginnings of a JTAG chain validation workflow. Depending on the number of devices and FPGAs in your JTAG chain, you can expand upon this test by performing more operations, in which you can interleave access to JTAG chains with larger data sets, and potentially multiple devices.

Verify Clock

The command to verify that your clock is toggling samples the clock asynchronously. Consequently, you may need to use this command several times to determine if the clock is toggling. The **jtag_debug_sample_clock.tcl** script samples the clock 10 times. To run this script, type `source jtag_debug_sample_clock.tcl` at the System Console prompt. You should see 10 values for the JTAG clock printed to the System Console as Figure 14–5 illustrates.

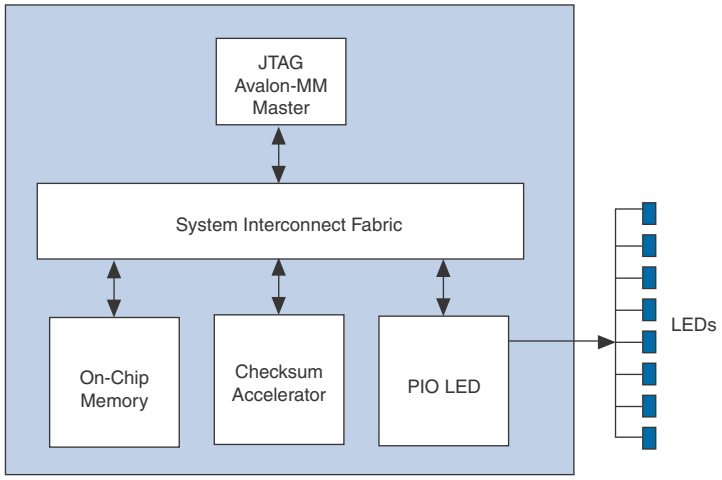
Figure 14–5. The jtag_debug_sample_clock Command



Checksum Example

In this example, you add an on-chip memory and hardware accelerator to the previous SOPC Builder system. The hardware accelerator calculates a checksum. Figure 14–6 illustrates this system.

Figure 14–6. SOPC Builder System for Checksum Accelerator Example



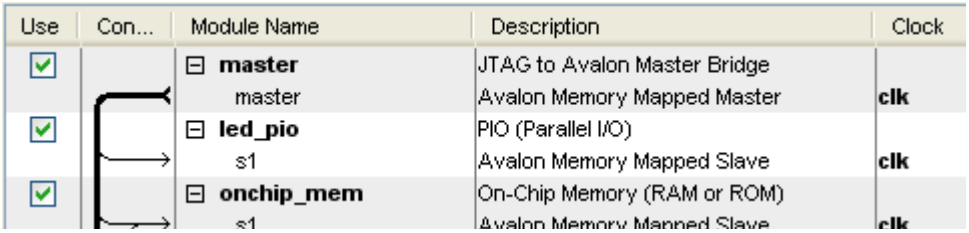
To build this example system, complete the following steps:

1. In the **System Contents** tab in SOPC Builder, double-click **On-Chip Memory (RAM or ROM)** in the **On-Chip** subfolder of the **Memories and Memory Controllers** folder to add this component to your system.
2. In the **On-Chip Memory (RAM or ROM)** wizard, for **Total memory size** type `128` to change the memory size to 128 bytes. Click **Finish** to accept the other default values.
3. To connect the on-chip memory to the master, click the open dot at the intersection of the onchip_mem `s1` Avalon slave port and the JTAG to Avalon Master Bridge `master` port.
4. In the **System Contents** tab, double-click **Checksum Accelerator** in the **Custom Component** folder to add this component to your system.
5. To connect the checksum accelerator `Slave` port, click on the open dot at the intersection of the accelerator `Slave` and the master `master` port.
6. To connect the checksum accelerator `Master` port, click on the open dot at the intersection of the accelerator `Master` and the onchip_mem `s1` port.
7. In the **Base** column, enter the base addresses for the slaves in your system.
   - Onchip_mem `s1` port—0x00000080
   - Accelerator `Slave` port—0x00000020

   Click on the **lock** icon next to each address to lock these values.

Figure 14–7 illustrates the completed system.

Figure 14–7. Checksum Accelerator Module Connections



8. Save your system.
9. In the **System Contents** tab, click **Next**.
10. In the **System Generation** tab, click **Generate**.
11. On the Quartus II Processing menu, click **Start Compilation**.
12. When compilation completes, re-program your board by typing the following command in the Nios II command shell:
    `nios2-configure-sof jtag_pio.sof` ↵
13. Type `system-console` ↵ in the Nios II command shell to start the System Console.

    ☞ If you reprogram your board, you must start a new System Console to receive the changes.

14. To run the checksum example, in the System Console, type:

    `source set_memory_and_run_checksum.tcl` ↵

Figure 14–8. System Console Output

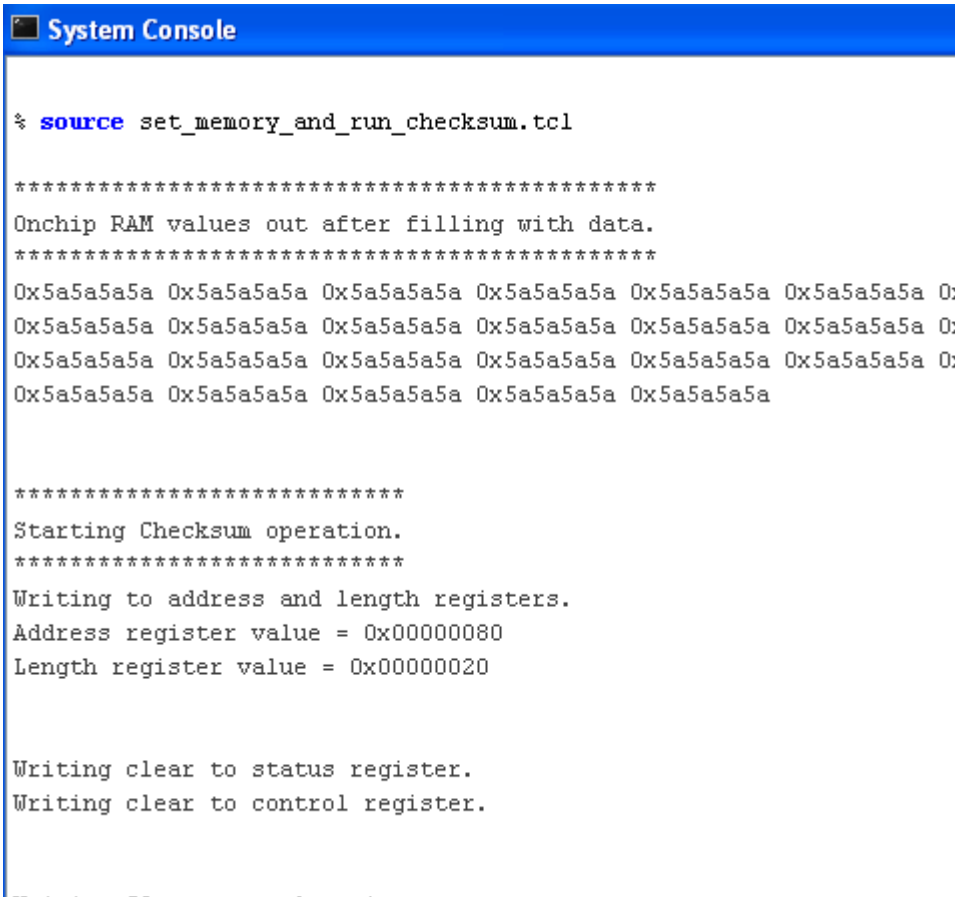


Figure 14–8 shows the output from a successful run.

1. You can change the value written into the RAM by changing the value given in the `fill_memory` routine in the **set_memory_and_run_checksum.tcl** file. Save the Tcl file after editing and rerun the command. (Because the system command uses `master_write_32`, if you use values that are less than 32 bits, they are filled with leading 0s.)

Nios II Processor Example

In this example you program the Nios II processor on your board to run the count binary software example that is included in the Nios II installation. This is a simple program that, using an 8-bit variable, repeatedly counts from 0 to 0xFF. The output of this variable is displayed on the LEDs on your board. After programming the Nios II processor, you use the System Console processor commands to start and stop the processor.

To run this example, complete the following steps:

1. Download the *Nios II Ethernet Standard Design Example* for your board from the Altera website.
2. Create a folder to extract the design. For this example, use **C:\Count_binary**.

3. Unzip the Nios II Ethernet Standard Design Example into **C:\Count_binary**.
4. In a Nios II command shell, change to the directory of your new project.
5. To program your board, type the following command in a Nios II command shell:

   ```
   nios2-configure-sof niosii_ethernet_standard_<board_version>.sof ↵
   ```

6. Using Nios II Software Build Tools for Eclipse, create a new Nios II Application and BSP from Template using the **Count Binary** template and targeting the Nios II Ethernet Standard Design Example.
7. To build the executable and linkable format (ELF) file (.**elf**) for this application, right-click the **Count Binary** project and select **Build Project**.

For more information about creating Nios II applications, refer to the *Using the Nios II Software Build Tools* chapter in the *Nios II Software Developer's Handbook*.

8. Download the .**elf** file to your board by right-clicking **Count Binary** project and selecting **Run As, Nios II Hardware**.

   The LEDs on your board provide a new light show.
9. Start the System Console by typing `system-console` in your Nios II command shell.
10. Set the processor service path to the Nios II processor by typing the following command:

    ```
    set niosii_proc [lindex [get_service_paths processor] 0] ↵
    ```

11. Open both services by typing the following commands:

    ```
    open_service processor $niosii_proc ↵
    ```

12. Stop the processor by typing the following command:

    ```
    processor_stop $niosii_proc ↵
    ```

    The LEDs on your board freeze.
13. Start the processor by typing the following command:

    ```
    processor_run $niosii_proc ↵
    ```

    The LEDs on your board resume their previous activity.
14. Stop the processor by typing the following command:

    ```
    processor_stop $niosii_proc ↵
    ```

15. Close the services by typing the following command:

```
close_service processor $niosii_proc ↵
```

The `processor_step`, `processor_set_register` and `processor_get_register` provide additional control over the Nios II processor.

Device Support

You can target all Altera device families with the System Console. Transceiver Toolkit commands, however, can only be targeted for Arria II GX and Stratix IV GX devices.

Conclusion

The System Console offers you a wide variety of options for communicating with modules in your design at a low level. You can use either Tcl scripting commands or GUI to access and run services for setting up, running tests, optimizing design parameters, and debugging designs you have programmed into Altera supported device families without having to recompile you designs.

Document Revision History

Table 14–18 shows the revision history for this chapter.

Table 14–18. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	Initial release. Previously released as the System Console User Guide, which is being obsoleted. This new chapter adds new commands.

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

Take an online survey to provide feedback about this handbook chapter.

15. Transceiver Link Debugging Using the Quartus II Software

QII53029-10.0.1

This chapter describes how to use the transceiver link debugging examples with the Transceiver Toolkit in the Quartus®II software. The Transceiver Toolkit in the Quartus II software allows you to quickly test the functionality of transceiver channels and help you improve the signal integrity of transceiver links in your design. You can use the Transceiver Toolkit before your design is complete to verify that your transceiver link can perform at the required speed within your error tolerances.

You can use an example design available on the Altera® website if you want to quickly start using the Transceiver Toolkit. You also can create a custom design to use the Transceiver Toolkit, or you can port one of the the design examples. In today's high speed interfaces, stringent Bit Error Ratio (BER) requirements are not easy to meet and debug. You can use the Transceiver Toolkit in the Quartus II software with design examples that you download from the Altera website that allow you to check and improve the signal integrity of transceiver links on your board before you complete the final design, saving you time and helping you find the best PMA settings for your high speed interfaces.

Transceiver Toolkit Overview

To launch the Transceiver Toolkit, in the main Quartus II window, on the Tools menu, select **Transceiver Toolkit**. The underlying framework for the Transceiver Toolkit is the System Console. The System Console performs low-level hardware debugging of your design. The System Console provides read and write access to the IP cores instantiated in your design. You can use the System Console for the initial bring-up of your PCB and low-level testing.

For information about the System Console, refer to the *Analyzing and Debugging Designs with the System Console* chapter in volume 3 of the *Quartus II Handbook*.

The Transceiver Toolkit allows you to perform run-time tasks. You can use the Transceiver Toolkit to perform high speed link tests for the transceivers in your devices. The Transceiver Toolkit allows you to easily test your high-speed interfaces in real-time. Various features and controls in the Transceiver Toolkit are described in the following paragraphs.

Auto Sweep

Auto Sweep, a feature in the Transceiver Toolkit, allows you to input sweep ranges for your transceiver physical medium attachment (PMA) settings and run tests in automatic mode. Auto Sweep can store a history of the test runs and keep a record of the best PMA settings. These settings can then be used in your final design.

For more information, refer to *Transceiver Toolkit Auto Sweep Panel* in Quartus II Help.

EyeQ

The EyeQ feature helps you to determine signal integrity. The EyeQ feature in the Transceiver Toolkit allows you to chart eye-width data and view the data in a bathtub curve. After you run the EyeQ feature you can view the data in the **Report** pane of the Transceiver Toolkit and export the data in comma-separated value **(.csv)** format for further analysis.

For further details on the EyeQ feature, refer to *Transceiver Toolkit EyeQ Panel* in Quartus II Help.

Control Links

The Transceiver Toolkit Channel Control panels allow you to run tests in manual mode. The Channel Control panels allow you to view and manually modify settings for transmitter and receiver channels at run-time.

For more information about how to set up the Transceiver Toolkit, refer to *Working with the Transceiver Toolkit* in Quartus II Help.

Transceiver Link Debugging Examples

Altera provides design examples to assist you with setting up and using the Transceiver Toolkit. To learn more about the Quartus II software version used to create these design examples, the target device, and development board details, refer to the **readme.txt** of each example. Each example is verified and tested with the Quartus II software version referenced in the **readme.txt**. However, you may be able to use these examples with a later version of the Quartus II software.

If you are recompiling the design examples for a different board, refer to the "Changing Pin Assignments" on page 15–6 section to determine which pin assignments you must edit.

To download design examples, go to the Transciever Toolkit Examples located on the Altera website.

Test Setup for Link Debugging

Testing signal integrity for high-speed transceiver links involves using data patterns, such as pseudo-random binary sequences (PRBS). Although the sequences appear to be random, they have specific properties that you can use to measure the quality of a link. In the example designs available on the Altera website, data patterns are generated by a pattern generator and are then transmitted by the transmitter. The transceiver on the far end can then be looped back so that the same data is then received by the receiver of the transceiver. The data obtained is then checked by a data checker to verify any bit errors.

Figure 15–1 and Figure 15–2 show examples of the test setup for the transceiver link debugger. The first design example described in this section has a structure similar to the design shown in Figure 15–1. This figure represents only one possible configuration for your design. You can also have the transmitter on one FPGA and the receiver on a different FPGA.

Figure 15–2 shows a similar test setup for the second design example described in this section, except that there are four sets of transceivers and receivers rather than one.

Figure 15–1. Transceiver Link Debugger Test Setup

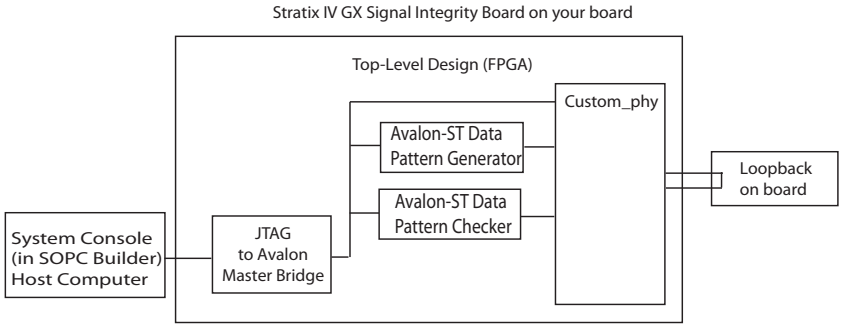
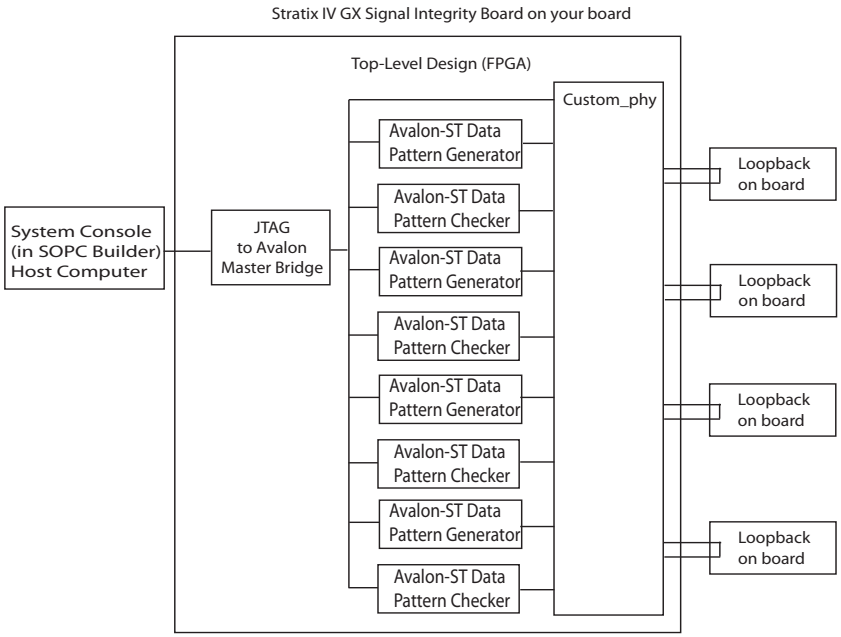


Figure 15–2. Transceiver Link Debugger Test Setup (Four Channels)



The design examples include the SOPC Builder system, which contains the following components:

- Custom Phy channel(s)
- Avalon-ST Data Pattern Generator
- Avalon-ST Data Pattern Checker
- JTAG to Avalon Master Master Bridge

Custom_phy

You can test all possible parallel data widths of the transceivers in these design examples. You can configure the *Custom_phy* IP core as 8-bit, 10-bit, 16-bit, 20-bit, 32-bit or 40-bit. The sweep tools disable word alignment during sweep. This mode is enabled to simplify timing closure. You can also use the Data Format Adapter IP component as required. You can have one or multiple channels in your design.

You use SOPC Builder to generate the *Custom_phy*. The *Custom_phy* IP core in the design examples are generated for Stratix IV GX devices.

To use the *Custom_phy* IP core with the Transceiver Toolkit, perform the following:

- Set the following parameters to meet your project requirements:
  - **Number of lanes**
  - **Bonded group size**
  - **Serialization factor**
  - **Data rate**
  - **Input clock frequency**
- Disable **8B/10B**
- Set **Word alignment mode** to manual
- Disable **rate match FIFO**
- Disable **byte ordering block**
- Change only the following **Analog Options**:
  - **Transmitter termination resistance**
  - **Receiver common mode voltage**
  - **Receiver termination resistance**

For more information about the components used in the *Custom_phy* IP core, refer to the *Custom PHY IP User Guide*.

Data Pattern Generator

This component produces data patterns in the test flow. You can use a variety of popular patterns to test transceiver signal integrity. The data pattern generator component is provided as an Altera IP component. You can use any pattern, but you must have a checker to verify that you receive that pattern properly.

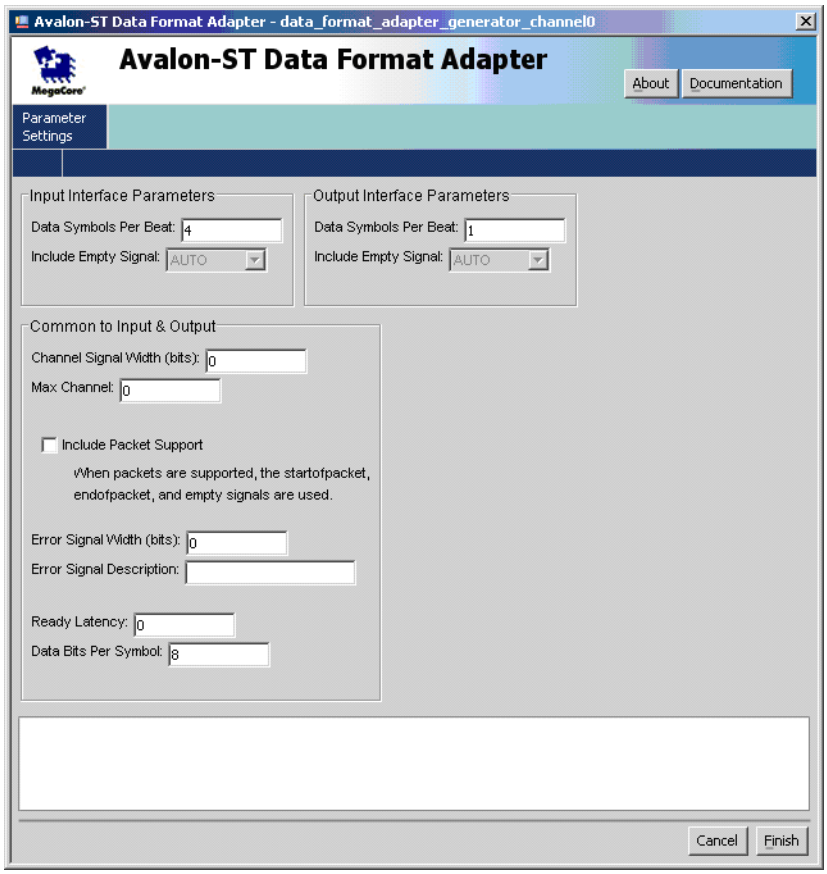
When you use the Avalon®-ST Data Pattern Generator, the width may be different in the pattern that you use, and you may need to use a data format adaptor. The Avalon-ST Data Pattern Generator component is available in the SOPC Builder library tree. The adaptor, however, does not form a new data generation component package, but serves as a data formatter.

The Avalon-ST Data Pattern Generator can generate industry-standard data patterns. Data patterns are generated on a 32-bit or 40-bit wide Avalon streaming source port.

For more information, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*.

Figure 15–3 shows the page in the wizard you use to set parameters for the Avalon-ST Data Format Adapter.

Figure 15–3. Avalon-ST Data Format Adapter



Data Checker

The Avalon-ST Data Pattern Checker components are provided as SOPC components. These components include Avalon-ST Data Pattern Checker, Avalon-ST Data Format Adapter, and Avalon-ST Timing Adapter. When the data checker runs, it attempts to find the selected data pattern in the incoming data by comparing the incoming pattern with the expected pattern.

The Avalon-ST Data Pattern Checker is available under **Debug and Performance** in the SOPC Builder component library tree. The checker is the core logic for data pattern checking. Data patterns are accepted on 32-bit or 40-bit wide Avalon streaming sink ports.

For more information, refer to *Avalon Streaming Data Pattern Generator and Checker Cores*.

You can use the design examples primarily to quickly test the functionality of receiver and transmitter channels in your design without needing to create any custom designs with data generators and checkers. When working with these examples, assume that your design is not complete, and that you want to check the transceiver link signal integrity. You can quickly change transceiver settings in the design examples to see how they affect transceiver link performance. You can also use the design examples to isolate and verify transceiver links without having to debug other logic in your design.

Compiling Design Examples

Once you have downloaded the design examples, open the Quartus II software version 10.0 or later and unarchive the project in the example. If you have access to the same development board with the same device as mentioned in the readme.txt of the example, you can directly program your device with the provided programming file in that example. If you want to recompile the design, you must regenerate the system in SOPC builder and recompile the design in the Quartus II software to get a new programming file.

If you have the same board as mentioned in readme.txt but a different device on your board, then you must choose the appropriate device and recompile the design. For example some early development boards are shipped with the engineering sample devices.

If you have a different board, you must edit the necessary pin assignments and recompile the design examples provided here.

Changing Pin Assignments

This section explains how to edit pin assignments for the first design example below. Design example named (first design example) is developed for the Stratix IV Transceiver Signal Integrity Development Kit (DK-SI-4SGX230N). This kit has a Stratix IV GX device. Table 15–1 lists the top-level pins in this design.

Table 15–1. Stratix IV GX Top-Level Pin Assignments (DK-SI-4SGX230N)

Top-Level Signal Name	I/O Standard	Pin Number on DK-SI-4SGX230N Board
REFCLK_GXB2_156M25 (input)	2.5 V LVTTL/LVCMOS	PIN_G38
S4GX_50M_CLK4P (input)	2.5 V LVTTL/LVCMOS	PIN_AR22
GXB1_RX1 (input)	1.4-V PCML	PIN_R38
GXB1_TX1 (output)	1.4-V PCML	PIN_P36

To change this example for a Stratix IV GX development kit (DK-DEV-4SGX230N), the pin assignments shown in Table 15–2 you must make before recompiling the design.

Table 15–2. Stratix IV GX Top-Level Pin Assignments (DK-DEV-4SGX230N)

Top-Level Signal Name	I/O Standard	Pin Number on DK-DEV-4SGX230N Board
REFCLK_GXB2_156M25 (input)	LVDS	PIN_AA2
S4GX_50M_CLK4P (input)	2.5 V LVTTL/LVCMOS	PIN_AC34
GXB1_RX1 (input)	1.4-V PCML	PIN_AU2
GXB1_TX1 (output)	1.4-V PCML	PIN_AT4

Similarly, you can change the pin assignment for your own board and recompile the design examples for use on your own board.

Transceiver Toolkit Link Test Setup

The following steps describe how to set up testing with the Transceiver Toolkit:

1. Program the device either with the programming file provided with the .zip file or with the new programming file after you recompile your project.
2. Open the Transceiver Toolkit from the Tools menu in the Quartus II software.
3. Make sure that the **.sof** of the device you use is programmed and that board settings, such as jumper settings, are correct.

Loading the Project in System Console

From the Transceiver Toolkit, load the Quartus project file (***.qpf**) by going to the File menu and selecting **Load Project**. Whether you have recompiled the design or not, the unzipped contents of the examples contain this file. Once the project is loaded, you can see the design information under **System Explorer** in the System Console.

Linking the Hardware Resource

Make sure the device board is turned ON and is available before you start the System Console or the Transceiver Toolkit. The connections are detected at startup of the tool. You can right click on a **Design Instance** under the **System Explorer** and link that instance to the device you want. Each design example contains one transceiver *Custom_phy* instance, the first one configured as one channel and the second one configured with four channels in bonded mode. Right-click on the design instance and click on the device to link that design instance with your device. If you are designing your own design example with more than one design instance and your board has more than one device, then you have the option of linking a certain device to a certain design instance. This is useful when you want to perform a link test between a transmitter and receiver on two separate devices.

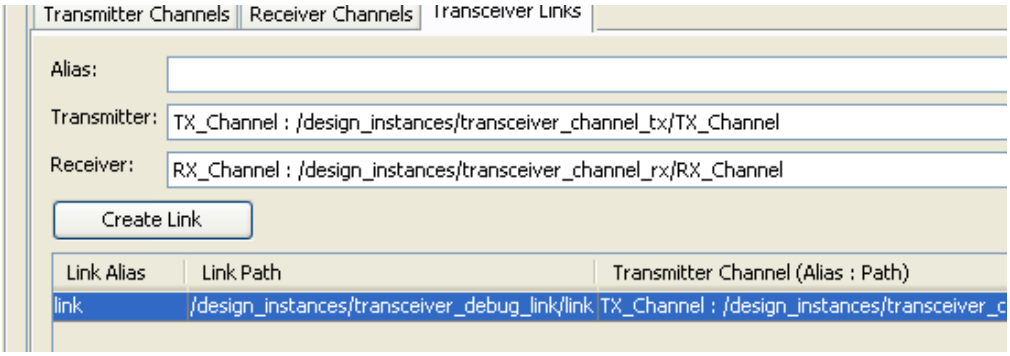
To set up the transmitter channels, receiver channels, and transceiver links, go to the Tools menu and select **Transceiver Toolkit**, or from the **Welcome to the Transceiver Toolkit** tab, click on the **Transceiver Toolkit**.

Creating the Channels

When you open the Transceiver Toolkit you can view three tabs in the System Console, including **Transmitter Channels**, **Receiver Channels**, and **Transceiver Links** tabs.

The Transmitter and Receiver Channel tabs are automatically populated with the existing transmitter and receiver channels in the design example, respectively. However, you can create your own additional transmitter and receiver channels in a situation where the expected configuration was not automatically populated. In the Link Channel tab you can create the link between a transmitter and receiver channel for which you want to perform the test. Links are automatically created when a receiver channel and transmitter channel share a transceiver channel. However, if you are not actually looping data back, but using it to transmit or receive to another transceiver channel, this is where you would define and create a new link. For example, in Figure 15–4 a link is created in the Link tab between a transmitter and receiver channel of the same device.

Figure 15–4. Creating a Link Channel in Transceiver Toolkit



You can also perform a physical link test without loopback by connecting one device transmitter to another device receiver. This panel is where you would define that connection into a link, and tests would run off that link. For example, as depicted in Figure 15–1, you typically use the transmitter and receiver channels of the same device and loop them back on the far side of the board trace to check the signal integrity of your high speed interface on the board trace. The link that you create in Transceiver Toolkit can be selected and has different buttons that you can use to start and control link tests.

You can also communicate with other devices that have the capability to generate and verify test patterns that Altera supports.

Running the Link Tests

You can click buttons in the Link Channel tab that control how you how you want to test the link. For example, you can use the **Auto Sweep** feature to sweep various transceiver settings parameters through a range of values, looking for the one that gives the best BER value. You can also open Transmitter, Receiver and Link Control panels to manually control PMA settings and run individual tests. You can change various controls in the panels to suit your requirements. To perform an auto sweep link test, perform the following steps:

1. Select the link to test and click **Auto Sweep** to open the auto sweep panel for the selected link.
2. Set the test pattern to test.
3. Select either **Smart Autosweep** or **Full AutoSweep**. **Full AutoSweep** runs a test of every combination of settings that falls within the bounds that you set. **Smart AutoSweep** tries to minimize the number of tests run to reach a good setting, which saves you time. **Smart Autosweep** will not test every possible setting, so may not achieve the very best setting possible; however, it can quickly find a good setting.
4. Set up run lengths for each test iteration. The run length limits you set are ignored in smart sweep mode.
5. Set up PMA sweep limits within the range you want to test.
6. Press **Start** and let the sweep run until complete.

When an Auto Sweep run has finished at least one iteration, you can create a report by clicking **Create Reports**, which opens the **Reports** tab. The report shows data from all runs that have completed so far. You can sort by columns and filter data by regular expression in the **Reports** tab. You can also export the reports to a comma-separated value (**.csv**) file by right clicking on a report. If you found a setting with the **Smart Sweep** mode, you can use the reported best case PMA settings, and apply a +/- 1 setting to them, returning the test to **Full Auto Sweep** mode. This can help you determine if the settings you chose are the best, by seeing if settings near those that you chose are also any better. You can then choose your own runtime conditions, reset t he Auto Sweep feature by pressing **Reset**, and re-run to generate BER data based on your own run length conditions. This procedure allows you to obtain a good setting for PMA in a shorter amount of time. You can then look at the **BER** column at the end of the full sweep to determine the best case.

When setting smart sweep ranges, try to include the 0 setting as often as possible, as that setting often provides the best results.

Viewing Results in the EyeQ Feature

Advanced FPGA devices such as Stratix IV have built-in EyeQ circuitry, which is used with the EyeQ feature in Transceiver Toolkit. The EyeQ feature allows you to estimate the horizontal eye opening at the receiver of the transceiver. With this feature, you can tune the PMA settings of your transceiver, which results in the best eye and BER at high data rates.

To understand the EyeQ circuit and how it works, refer to *AN 605: Using the EyeQ Feature in Stratix IV devices*.

The following paragraph describes how to use the EyeQ feature in the Transceiver Toolkit to view the results of the link tests that you run.

1. Select a **Transceiver Link** or **Receiver Channel** in the main Transceiver Toolkit panel that you want to run EyeQ against.
2. Click **Control Link** or **Control Channel** to open control panels. Use these control panels to set the PMA settings and test pattern that you want the EyeQ feature to run against. You can check the report panel or **Best Case** column from an Autosweep run for the settings with the best BER value and enter those PMA values through the Transmitter and Receiver control panels.
3. Click **EyeQ** to open the EyeQ feature.
4. Review the test stop conditions and set them to your preference.
5. Click **Run**, which allows the EyeQ feature to gather current settings of the channel and use those settings to start a phase sweep. 32 iterations of the phase sweep will run.
6. As the run progresses, you can view the status, which is displayed on the top section of the feature. You can read current progress and the chart is updated with values each time a run completes. When the run is completed, you should see a bathtub curve.
7. Any time at least one run has completed, you can click **Create Report** to view past data runs in a report panel.
8. When the run finishes, if the bathtub curve looks like a hill instead, click the center eye button, which then reorganizes the data into a bathtub curve.
9. If the data does not display on the chart, you can right-click and choose **Auto-Range**, which should make the data appear.
10. If you want to change PMA settings and re-run the EyeQ feature, make sure you first stop and reset the EyeQ feature. If you do not reset, the EyeQ feature continues testing based on the original PMA settings of the current test that it had begun, and overwrites any setting you may have changed through the control panel.
11. After you stop and reset the EyeQ feature, change settings in the link or receiver channel control panel. Then click **Start** on the EyeQ feature to start a new set of tests.

When you can see that you are running good PMA settings, the bathtub curve is wide, with sharp slopes near the edges. The curve may be up to 30 units wide. If the bathtub is narrow, maybe as small as two units wide, then the signal quality may be poor. The wider the bathtub curve, the wider the eye you have. Conversely, the smaller the bathtub curve, the smaller the eye.

For more information about how to use the EyeQ feature, refer to *Working with the Transceiver Toolkit* in Quartus II Help.

Tcl Script in System Console

System Console is the framework in the Quartus II software that supports the Transceiver Toolkit. System Console provides a number of Tcl commands that you can use to build a custom test routine script to test the transceiver link using the data generator and checker. System Console allows you to tune PMA parameter, such as those for changing DC gain. To get help on the Tcl commands available, type `help` in the Tcl console in System Console. The following steps describe a transceiver link test flow. You can perform all the tasks with Tcl commands in System Console.

1. Load the Quartus II project.
2. Find and link the design and service path.
3. Find and open links to transmitter channels and receiver channels.
4. Set up PRBS patterns to run on the link.
5. Set up PMA settings on transmitter and receiver channels.
6. Use PIO logic to generate a rising edge to enable the word aligner.
7. Start the link test.
8. Poll the receiver for BER data.
9. When the test is finished, stop the link test.
10. Close the link.

For more information about Tcl commands and their usage, refer to the *Analyzing and Debugging Designs with the System Console* chapter in volume 3 of the *Quartus II Handbook*.

The Tcl scripts provided with the design examples help you understand Tcl commands associated with each task.

Running Tcl Scripts

The tasks that you perform with the Transceiver Toolkit GUI for setting up your test environment you can also save as Tcl scripts. For example, you can save the steps you perform for loading your project in the Transceiver Toolkit by clicking **Create Tcl setup script** on the File menu.

You can save your steps at various stages when you add projects, create design instances, link designs to devices, and create transceiver links., so that you do not have to redo all the steps in the GUI to reach the initial setup you created to get your specific configuration ready for debugging. All the saved scripts are shown under the **Scripts** in the **System Explorer**.You can execute the script by right-clicking on **Scripts** under the **System Explorer** in the Transceiver Toolkit, and then clicking **Run**, or you can double-click on the script. You can also execute the script from the **System Console command line**.

Conclusion

You gain productivity when optimizing high speed transceiver links in your board designs with the Transceiver Toolkit and design examples provided from Altera. You can easily set up automatic testing of your transceiver channels so that you can monitor, debug, and optimize transceiver link channels in your board design quicker than before. You then know the optimal PMA settings to use for each channel in your final FPGA design. You can use standard design examples that you download from Altera's website, and then customize the examples to match your own requirements.

Document Revision History

Table 15–3 shows the revision history for this handbook chapter.

Table 15–3. Document Revision History

Date	Version	Changes Made
August 2010	10.0.1	Corrected links
July 2010	10.0.0	Initial release

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

Take an online survey to provide feedback about this handbook chapter.

16. Quick Design Debugging Using SignalProbe

QII53008-10.0.0

This chapter provides detailed instructions about how to use SignalProbe to quickly debug your design. 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.

The SignalProbe feature is supported with the Arria® GX, Stratix® series, Cyclone® series, and MAX® II, device families.

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.

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 engineering change order (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 16–1.
2. Perform a Full Compilation, described on page 16–2.
3. Assign a SignalProbe Source, described on page 16–2.
4. Add Registers to the Pipeline Path to SignalProbe Pin, described on page 16–2.
5. Perform a SignalProbe Compilation, described on page 16–3.
6. Analyze the Results of the SignalProbe Compilation, described on page 16–3.

Reserve the SignalProbe Pins

SignalProbe pins can only be reserved after compiling your design. 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. For more information, refer to *About SignalProbe* in Quartus II Help.

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 output pin.

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. For example, nodes and registers within Gigabit transceivers in Stratix IV devices cannot be probed because there are no physical routes available to the pins.

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. For more information, refer to *SignalProbe Pins Dialog Box* and *Add SignalProbe Pins Dialog Box* in Quartus II Help.

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*.

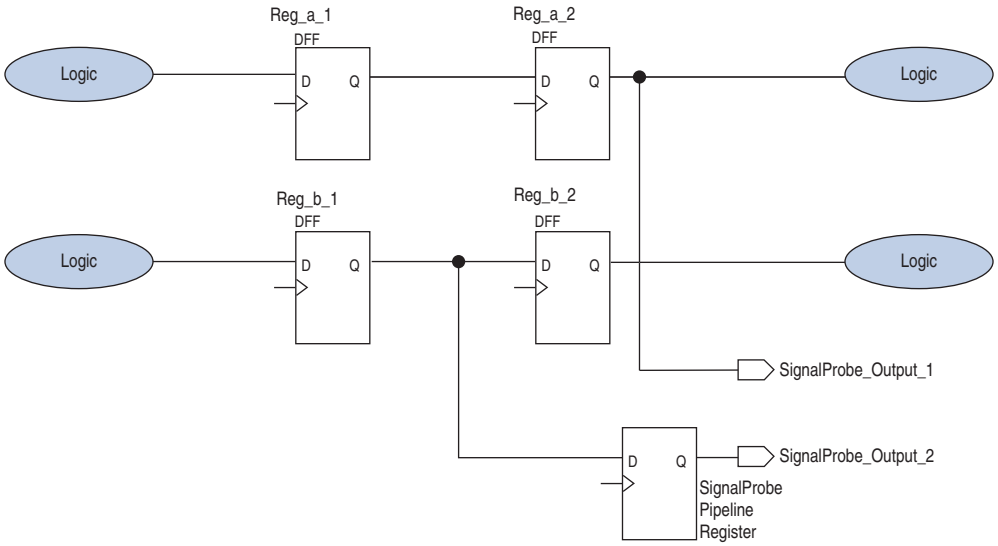
Add Registers to the Pipeline Path to SignalProbe Pin

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 16–1 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 16–1. Synchronizing SignalProbe Outputs with a SignalProbe Register



To pipeline an existing SignalProbe connection, refer to *Add SignalProbe Pins Dialog Box* in Quartus II Help.

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 16–6.

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 16–2). (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 16–2. 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**.

Performing a SignalProbe Compilation

After a full compilation, you can start a SignalProbe compilation either manually or automatically. 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, click **Start Check & Save All Netlist Changes**.

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.

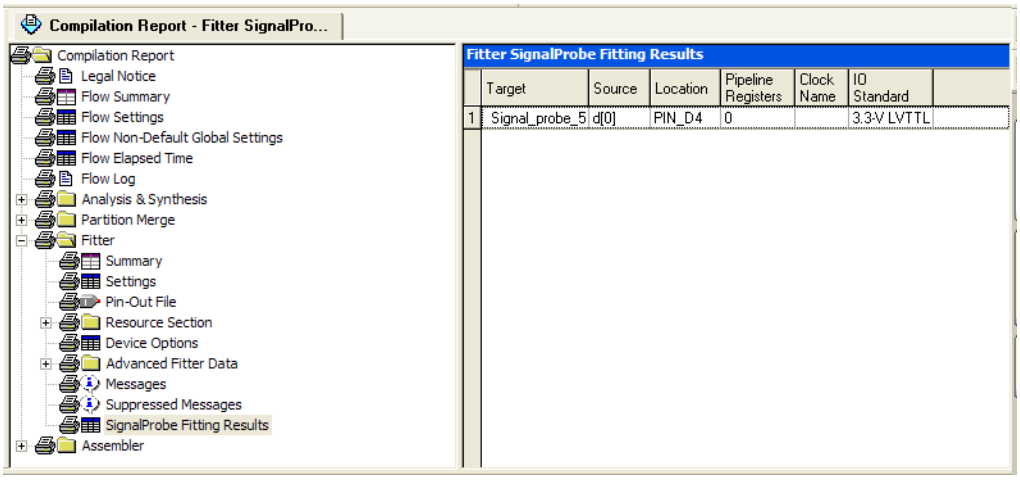
Understanding the Results of a SignalProbe Compilation

After a SignalProbe compilation, the results appear in two sections of the compilation report file. The fitting results and status (Table 16–1) of each SignalProbe pin is displayed in the **SignalProbe Fitting Result** screen in the Fitter section of the Compilation Report (Figure 16–3).

Table 16–1. Status Values

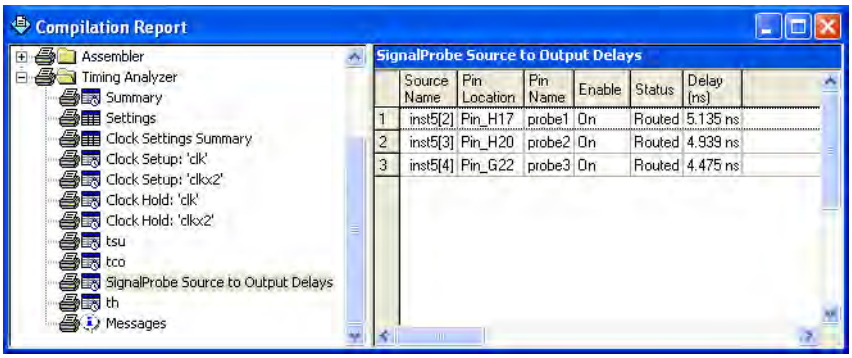
Status	Description
Routed	Connected and routed successfully
Not Routed	Not enabled
Failed to Route	Failed routing during last SignalProbe compilation
Need to Compile	Assignment changed since last SignalProbe compilation

Figure 16–3. 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 16–4).

Figure 16–4. SignalProbe Source to Output Delays Page in the Compilation Report Window



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.

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 Tcl commands in this section are part of the `::quartus::chip_planner` Quartus II Tcl API. Source or include the `::quartus::chip_planner` Tcl package in your scripts to make these commands available.

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 reserve 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, use the following Tcl command:

```
delete_sp [-h | -help] [-long_help] -pin_name <pin name>
```

Enable a SignalProbe Pin

To enable a SignalProbe pin, use the following Tcl command:

```
enable_sp [-h | -help] [-long_help] -pin_name <pin name>
```

Disable a SignalProbe Pin

To disable a SignalProbe pin, use the following Tcl command:

```
disable_sp [-h | -help] [-long_help] -pin_name <pin name>
```

Perform a SignalProbe Compilation

To perform a SignalProbe compilation, type the following command:

```
quartus_sh --flow signalprobe <project name>
```

Script Example

Example 16–1 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 16–1. Creating a SignalProbe Pin Called sp1

```
package require ::quartus::chip_planner
project_open project
read_netlist
make_sp -pin_name sp1 -src_name reg1
check_netlist_and_save
project_close
```

Reserving SignalProbe Pins

To reserve a SignalProbe pin, add the commands shown in Example 16–2 to the Quartus II Settings File **.qsf** for your project.

Example 16–2. 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 16–1.

Common Problems When Reserving 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.

Adding SignalProbe Sources

Use the following Tcl commands to add SignalProbe sources. For more information about adding SignalProbe sources, refer to *SignalProbe Pins Dialog Box* and *Add SignalProbe Pins Dialog Box* in Quartus II Help.

To assign the node name to a SignalProbe pin, use the following Tcl 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, use 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, use the following Tcl command:

```
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 16–4.

Run SignalProbe Manually

To run SignalProbe as part of a scripted flow using Tcl, use the following in your script:

```
execute_flow -signalprobe
```

To perform a Signal Probe compilation interactively at a command prompt, type the following command:

```
quartus_sh_fit --flow signalprobe <project name>
```

For more information about running SignalProbe manually, refer to "Performing a SignalProbe Compilation" on page 16–4.

Enable or Disable All SignalProbe Routing

Use the Tcl command in Example 16–3 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 16–3. 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 16–4.

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 16–6.

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.

Document Revision History

Table 16–2 shows the revision history for this chapter.

Table 16–2. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Revised for new UI. ■ Removed section SignalProbe ECO flows ■ Removed support for SignalProbe pin preservation when recompiling with incremental compilation turned on. ■ Removed outdated FAQ section. ■ Added links to Quartus II Help for procedural content.
November 2009	9.1.0	■ Removed all references and procedures for APEX devices. ■ Style changes.
March 2009	9.0.0	■ Removed the "Generate the Programming File" section ■ Removed unnecessary screenshots ■ Minor editorial updates
November 2008	8.1.0	■ Modified description for preserving SignalProbe connections when using Incremental Compilation ■ Added plausible scenarios where SignalProbe connections are not reserved in the design
May 2008	8.0.0	■ 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

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

Take an online survey to provide feedback about this handbook chapter.

© July 2010 Altera Corporation

17. Design Debugging Using the SignalTap II Logic Analyzer

QII53009-10.0.0

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.

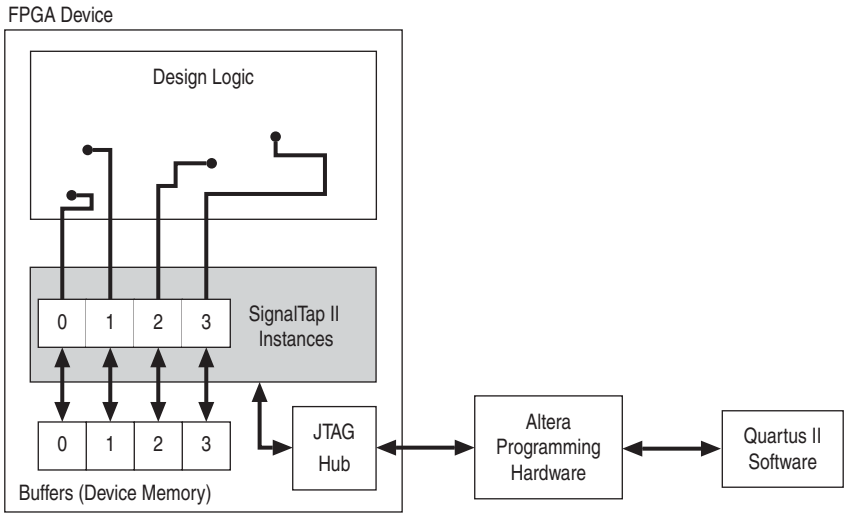
The SignalTap® II Logic Analyzer is scalable, easy to use, and is available as a stand-alone package or included with the Quartus® II software subscription. This logic analyzer helps debug an FPGA design by probing the state of the internal signals in the design without the use of external equipment. Defining custom trigger-condition logic provides greater accuracy and improves the ability to isolate problems. The SignalTap II 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 Logic Analyzer" on page 17–4
- "SignalTap II Logic Analyzer Task Flow" on page 17–4
- "Configure the SignalTap II Logic Analyzer" on page 17–8
- "Define Triggers" on page 17–25
- "Compile the Design" on page 17–44
- "Program the Target Device or Devices" on page 17–49
- "Run the SignalTap II Logic Analyzer" on page 17–52
- "View, Analyze, and Use Captured Data" on page 17–57
- "Other Features" on page 17–63
- "SignalTap II Scripting Support" on page 17–66
- "Design Example: Using SignalTap II Logic Analyzers in SOPC Builder Systems" on page 17–68
- "Custom Triggering Flow Application Examples" on page 17–69

The SignalTap II 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 Logic Analyzer supports the highest number of channels, largest sample depth, and fastest clock speeds of any logic analyzer in the programmable logic market. Figure 17–1 shows a block diagram of the components that make up the SignalTap II Logic Analyzer.

Figure 17–1. SignalTap II Logic Analyzer Block Diagram *(Note 1)*



Note to Figure 17–1:

(1) This diagram assumes that you compiled the SignalTap II Logic Analyzer 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 17–44.

This chapter is intended for any designer who wants to debug an FPGA design during normal device operation without the need for external lab equipment. Because the SignalTap II 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 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

You need the following components to perform logic analysis with the SignalTap II Logic Analyzer:

- Quartus II design software

or

Quartus II Web Edition (with the TalkBack feature enabled)

or

SignalTap II Logic Analyzer standalone software

- Download/upload cable
- Altera® development kit or your design board with JTAG connection to device under test

The Quartus II software Web Edition does not support the SignalTap II Logic Analyzer with the incremental compilation feature.

The device's memory blocks store captured data and transfers the data to the Quartus II software waveform display with a JTAG communication cable, such as EthernetBlaster or USB-Blaster™. Table 17–1 summarizes features and benefits of the SignalTap II Logic Analyzer.

Table 17–1. SignalTap II Logic Analyzer Features and Benefits

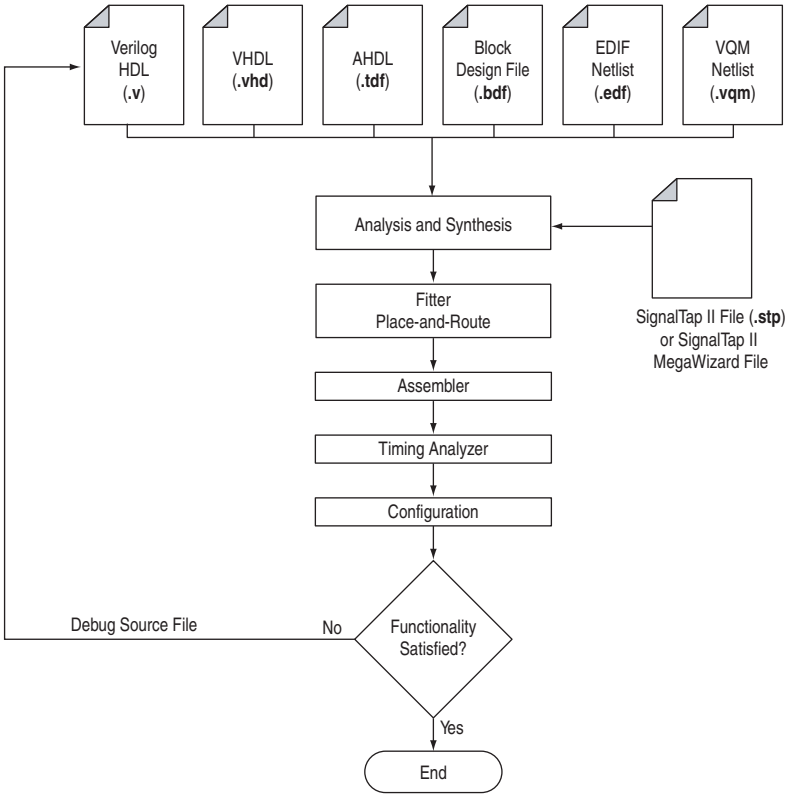
Feature	Benefit
Multiple logic analyzers in a single device	Captures data from multiple clock domains in a design at the same time.
Multiple logic analyzers in multiple devices in a single JTAG chain	Simultaneously captures data from multiple devices in a JTAG chain.
Plug-In Support	Easily specifies nodes, triggers, and signal mnemonics for IP, such as the Nios® II processor.
Up to 10 basic or advanced trigger conditions for each analyzer instance	Enables sending more complex data capture commands 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 logic analyzer captures.
Incremental compilation	Modifies the SignalTap II 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 debugging of your design.
MATLAB integration with included MEX function	Collects the SignalTap II 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 Logic Analyzer configurations.
No additional cost	The SignalTap II 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	You can use the SignalTap II Logic Analyzer in tandem with any JTAG based on-chip debugging tool, such as an In-System Memory Content editor allowing you to change signal values in real-time while you are running an analysis with the SignalTap II Logic Analyzer.

The Quartus II software offers a portfolio of on-chip debugging solutions. For an overview and comparison of all tools available in the In-System Verification Tool set, refer to *Section V. In-System Design Debugging*.

Design Flow Using the SignalTap II Logic Analyzer

Figure 17–2 shows a typical overall FPGA design flow for using the SignalTap II 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 figure shows the flow of operations from initially adding the SignalTap II Logic Analyzer to your design to final device configuration, testing, and debugging.

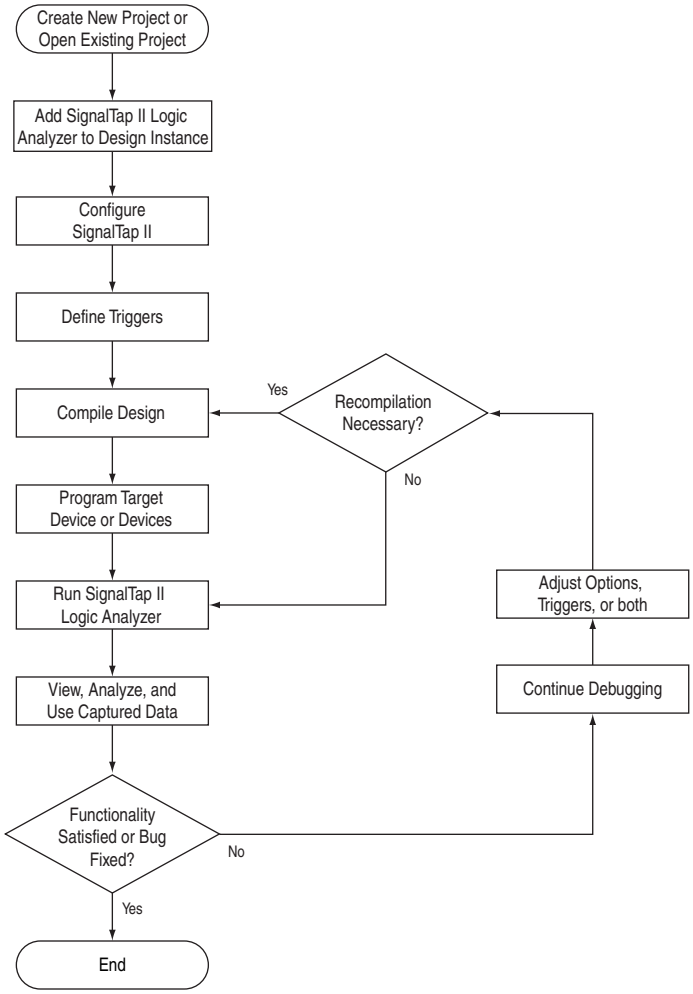
Figure 17–2. SignalTap II FPGA Design and Debugging Flow



SignalTap II Logic Analyzer Task Flow

To use the SignalTap II Logic Analyzer to debug your design, you perform a number of tasks to add, configure, and run the logic analyzer. Figure 17–3 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 17–3. SignalTap II Logic Analyzer Task Flow



Add the SignalTap II Logic Analyzer to Your Design

Create an **.stp** 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.

For information about creating an **.stp**, refer to *Setting Up the SignalTap II Logic Analyzer* in Quartus II Help.

Configure the SignalTap II Logic Analyzer

After you add the SignalTap II Logic Analyzer 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 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.

For information about configuring the SignalTap II Logic Analyzer, refer to *Setting Up the SignalTap II Logic Analyzer* in Quartus II Help.

Define Trigger Conditions

The SignalTap II 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 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 allow you to capture data from trigger events occurring immediately after the device enters user-mode after configuration.

For information about defining trigger conditions, refer to *Setting Up the SignalTap II Logic Analyzer* in Quartus II Help.

Compile the Design

With the **.stp** 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 Logic Analyzer, along with Quartus II incremental compilation, to reduce recompile times.

For information about compiling your design, refer to *Compiling a Design that Contains a SignalTap II Logic Analyzer* in Quartus II Help.

Program the Target Device or Devices

When you are debugging a design with the SignalTap II Logic Analyzer, you can program a target device directly from the **.stp** without using the Quartus II Programmer. You can also program multiple devices with different designs and simultaneously debug them.

For instructions on programming devices in the Quartus II software, refer to *Running the SignalTap II Logic Analyzer* in Quartus II Help.

Run the SignalTap II 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** for analysis.

For information about analyzing results from the SignalTap II Logic Analyzer, refer to *Analyzing Data in the SignalTap II Logic Analyzer* in Quartus II Help.

View, Analyze, and Use Captured Data

After you have captured data and read it into the **.stp**, that data is available for analysis and debugging. Set up mnemonic tables, either manually or with a plug-in, to simplify reading and interpreting 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 the data to other formats for sharing and further study.

For information about analyzing results from the SignalTap II Logic Analyzer, refer to *Analyzing Data in the SignalTap II Logic Analyzer* in Quartus II Help.

Embedding Multiple Analyzers in One FPGA

The SignalTap II Logic Analyzer Editor includes support for adding multiple logic analyzers by creating instances in the **.stp**. You can create a unique logic analyzer for each clock domain in the design.

For information about creating instances, refer to *Running the SignalTap II Logic Analyzer* in Quartus II Help.

Monitoring FPGA Resources Used by the SignalTap II Logic Analyzer

The SignalTap II 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 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** pane of the SignalTap II Logic Analyzer 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 17–2 shows the SignalTap II Logic Analyzer M4K memory block resource usage for the listed devices per signal width and sample depth.

Table 17–2. SignalTap II Logic Analyzer M4K Block Utilization *(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 17–2:

(1) When you configure a SignalTap II 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 Logic Analyzer

You can create a SignalTap II 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 logic analyzer. These features are described in the following sections:

- "Adding Finite State Machine State Encoding Registers" on page 17–13
- "Using the Storage Qualifier Feature" on page 17–16
- "State-Based Triggering" on page 17–29

For information about creating a SignalTap II instance with the MegaWizard Plug-In Manager, refer to *Setting Up the SignalTap II Logic Analyzer* in Quartus II Help.

Configure the SignalTap II Logic Analyzer

The **.stp** 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 Logic Analyzer because of the requirements for configuring a logic analyzer. All settings allow you 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 17–40.

Assigning an Acquisition Clock

Assign a clock signal to control the acquisition of data by the SignalTap II 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.

For information about assigning an acquisition clock, refer to *Working with Nodes in the SignalTap II Logic Analyzer* in Quartus II Help.

Altera recommends that you exercise caution when using a recovered clock from a transceiver as an acquisition clock for the SignalTap II Logic Analyzer. Incorrect or unexpected behavior has been noted, particularly when a recovered clock from a transceiver is used as an acquisition clock with the power-up trigger feature.

If you do not assign an acquisition clock in the SignalTap II Logic Analyzer 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** 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**. 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**.

For more information about incremental compilation, refer to *About Incremental Compilation* in Quartus II Help.

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** for correct operation. The SignalTap II Status Indicator also indicates if an invalid node name exists in the **.stp**.

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 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 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 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 files 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 Logic Analyzer, refer to "Faster Compilations with Quartus II Incremental Compilation" on page 17–44.

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 Logic Analyzer. Because only post-fitting signals can be added to the SignalTap II 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 Logic Analyzer. This is often necessary when a plug-in is used to add a group of signals for a particular IP.

If you use incremental compilation flow with the SignalTap II Logic Analyzer, pre-synthesis nodes may not be connected to the SignalTap II 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.

For more information about node preservation or how to avoiding these warnings, refer to *Working with Nodes in the SignalTap II Logic Analyzer* in Quartus II Help.

Assigning Data Signals Using the Technology Map Viewer

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** for your design or a new **.stp.**

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**. 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 17–25.

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 with a plug-in. The SignalTap II Logic Analyzer comes with one plug-in already installed for the Nios II processor. Besides easy signal addition, plug-ins also provide 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 17–25 and "View, Analyze, and Use Captured Data" on page 17–57.

To add signals to the **.stp** 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, choose 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 informs you that you cannot use the selected plug-in.

2. The **Select Hierarchy Level** dialog box appears showing the IP hierarchy of your design. Select the IP that contains the signals you want to monitor with the plug-in and click **OK**.
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 specify options for the plug-in. With the Nios II plug-in, you can optionally select an **.elf** containing program symbols from your Nios II Integrated Development Environment (IDE) software design. Specify 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 **Create debugging nodes for IP cores**.

All the signals included in the plug-in are added to the node list.

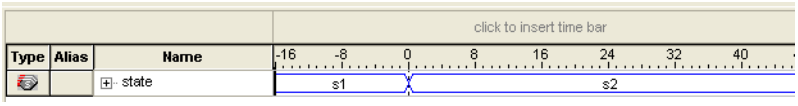
Adding Finite State Machine State Encoding Registers

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 can 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.

The SignalTap II Logic Analyzer GUI can detect FSMs in your compiled design. The SignalTap II Logic Analyzer configuration automatically tracks the FSM state signals as well as state encoding through the compilation process. Shortcut menu commands from the SignalTap II Logic Analyzer GUI allow you to add all of the FSM state signals to your 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. The FSM debugging feature supports adding FSM signals from both the pre-synthesis and post-fit netlists.

Figure 17–4 shows the waveform viewer with decoded signal values from a state machine added with the FSM debugging feature.

Figure 17–4. 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*.

For information about adding FSM signals to the configuration file, refer to *Setting Up the SignalTap II Logic Analyzer* in Quartus II Help.

Modifying and Restoring Mnemonic Tables for State Machines

When you add FSM state signals via the FSM debugging feature, the SignalTap II Logic Analyzer 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. To restore a FSM mnemonic table to a new record, turn off **Overwrite existing mnemonic table** 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 17–60.

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 use other EDA synthesis tools.

If you add post-fit FSM signals, the SignalTap II Logic Analyzer FSM debug feature may not track all 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 can still 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 specify 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 Logic Analyzer with some devices, you have the option to select the RAM type where acquisition data is stored. Once SignalTap II Logic Analyzer is allocated to a particular RAM block, the entire RAM block becomes a dedicated resource for the logic analyzer. RAM selection allows you to preserve a specific memory block for your design and allocate another portion of memory for SignalTap II Logic Analyzer data acquisition. For example, if your design has an application that requires a large block of memory resources, such a large instruction or data cache, you would choose to use MLAB, M512, or M4k blocks for data acquisition and leave the M9k blocks for the rest of your design.

To select the RAM type to use for the SignalTap II Logic Analyzer 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.

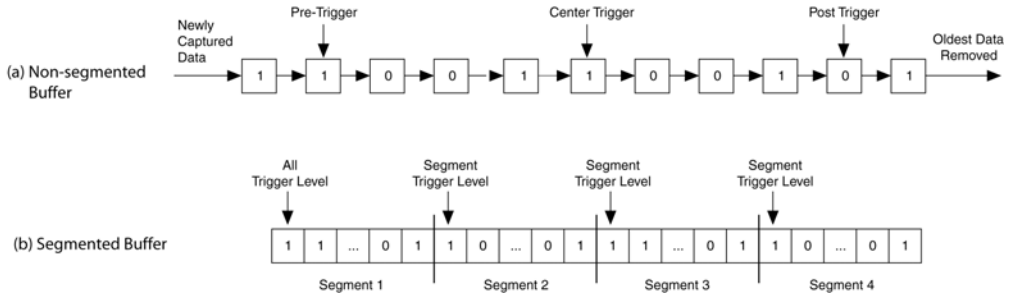
Choosing the Buffer Acquisition Mode

The Buffer Acquisition Type Selection feature in the SignalTap II 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 Logic Analyzer—a non-segmented buffer and a segmented buffer. With a non-segmented buffer, the SignalTap II Logic

Analyzer treats entire memory space as a single FIFO, continuously filling the buffer until the 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 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 17–5 illustrates the differences between the two buffer types.

Figure 17–5. Buffer Type Comparison in the SignalTap II Logic Analyzer *(Note 1)*



Note to Figure 17–5:

(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 17–39 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 17–16.

Non-Segmented Buffer

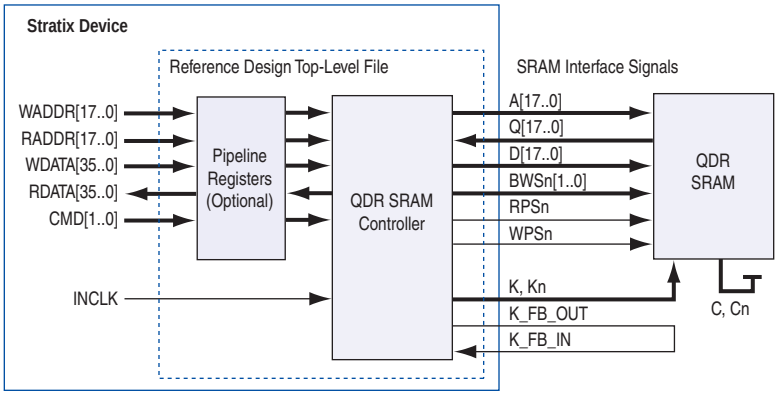
The non-segmented buffer (also known as a circular buffer) shown in Figure 17–5 (a) is the default buffer type used by the SignalTap II Logic Analyzer. While the logic analyzer is running, data is stored in the buffer until it fills up, at which point new data replaces the oldest data. This continues until a specified trigger event, consisting of a set of trigger conditions, occurs. When the trigger event 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**. To capture the majority of the data before the trigger occurs, select **Post trigger position** from the list. To capture the majority of the data after the trigger, select **Pre-trigger position**. To center the trigger position in the data, select **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 17–39.

Segmented Buffer

A segmented buffer allows you to debug systems that contain relatively infrequent recurring events. The acquisition memory is split into evenly sized segments, with a set of trigger conditions defined for each segment. Each segment acts as a non-segmented buffer. Figure 17–6 shows an example of a segmented buffer system.

Figure 17–6. Example System that Generates Recurring Events



The SignalTap II Logic Analyzer verifies the functionality of the design shown in Figure 17–6 to ensure that the correct data is written to the SRAM controller. Buffer acquisition in the SignalTap II 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 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 Logic Analyzer Editor and select the number of segments to use. In the example in Figure 17–6, 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 *Configuring the Trigger Flow in the SignalTap II Logic Analyzer* 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 Logic Analyzer is to sample data on every clock cycle. With a non-segmented buffer, there is one data window that represents a comprehensive snapshot of the datastream. Similarly, segmented buffers use several smaller sampling windows spread out over more time, with each sampling window representing a contiguous data set.

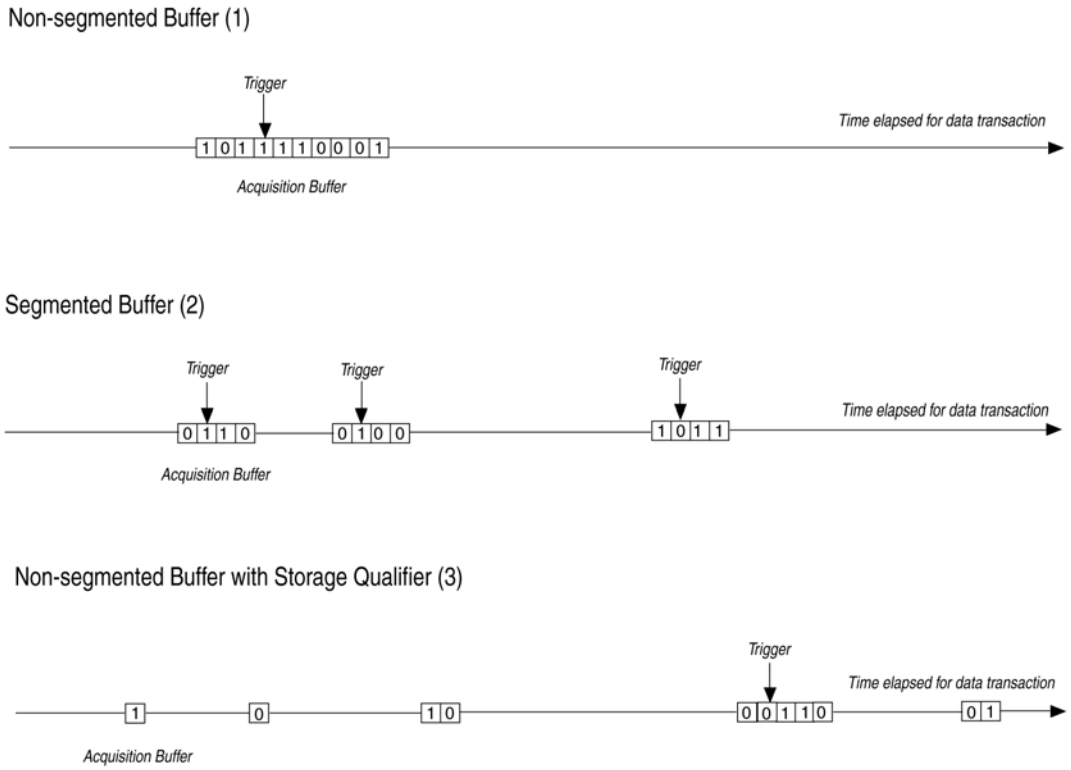
With carefully chosen trigger conditions and a generous sample depth for the acquisition buffer, analysis using segmented and non-segmented buffers captures a majority of functional errors in a chosen signal set. However, each data window can have a considerable amount of redundancy associated with it; for example, a capture of a data stream containing long periods of idle signals between data bursts. With default behavior using the SignalTap II Logic Analyzer, you cannot 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 during each clock cycle of 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 in the specified number of samples over a longer period of analysis.

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 17–7 illustrates three ways the SignalTap II Logic Analyzer writes into acquisition memory.

You can only use the Storage Qualification feature with a non-segmented buffer. The MegaWizard Plug-In Manager instantiated flow only supports the Input Port mode for the Storage Qualification feature.

Figure 17–7. Data Acquisition Using Different Modes of Controlling the Acquisition Buffer



Notes to Figure 17–7:

(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 six 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 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 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 17–6.

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 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 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** to create a SignalTap II Logic Analyzer instance, specify the storage qualifier signal using the input port field located on the **Setup** tab. You must specify this port for your project to compile.

If you use the MegaWizard Plug-In Manager flow, the storage qualification input port, if specified, appears in the MegaWizard-generated instantiation template. You can then connect this port to a signal in your RTL.

Figure 17–8 shows a data pattern captured with a segmented buffer. Figure 17–9 shows a capture of the same data pattern with the storage qualification feature enabled.

Figure 17–8. Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Input port mode)

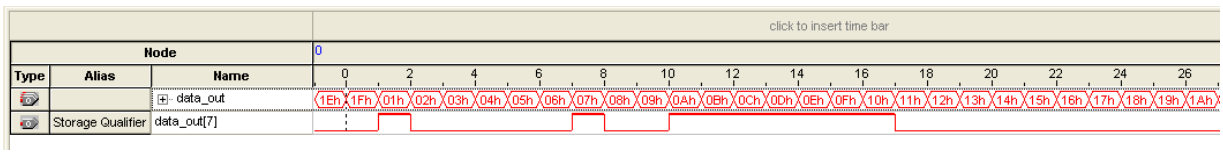
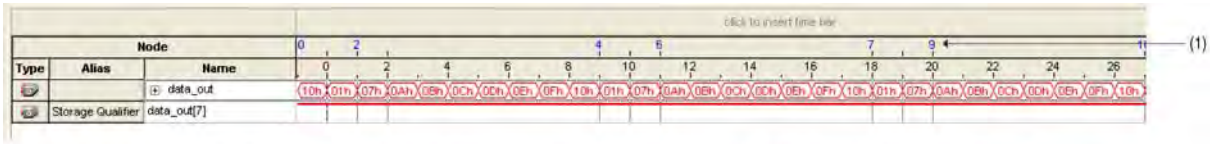


Figure 17–9. Data Acquisition of a Recurring Data Pattern Using an Input Signal as a Storage Qualifier



(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 17–10 shows the transitional storage qualifier setup. Figure 17–11 and Figure 17–12 show captures of a data pattern in continuous capture mode and a data pattern using the Transitional mode for storage qualification.

Figure 17–10. Transitional Storage Qualifier Setup

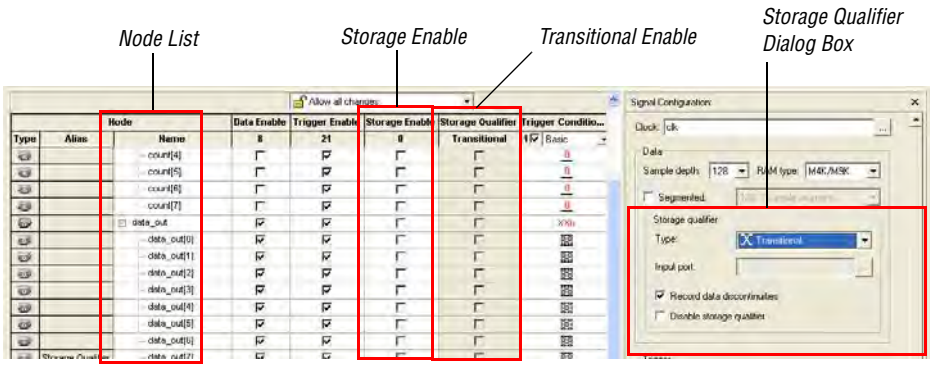


Figure 17–11. Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Transitional mode)

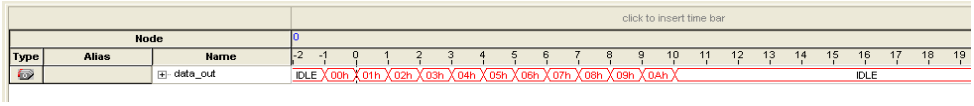
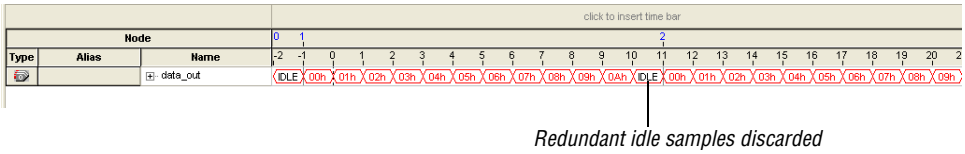


Figure 17–12. Data Acquisition of Recurring Data Pattern Using a Transitional Mode as a Storage Qualifier



Conditional Mode

In Conditional mode, the SignalTap II 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 Logic Analyzer writes into the buffer during the clock cycles in which the condition you specify evaluates TRUE.

You can select either **Basic** or **Advanced** storage qualifier conditions. A **Basic** storage qualifier condition matches each signal to one of the following:

- Don't Care
- Low
- High
- Falling Edge
- Either Edge

If you specify a Basic Storage qualifier condition for more than one signal, the SignalTap II 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 17–13 details the conditional storage qualifier setup in the **.stp**.

You can specify up storage qualification conditions similar to the manner in which trigger conditions are specified. For details about basic and advanced trigger conditions, refer to the sections "Creating Basic Trigger Conditions" on page 17–25 and "Creating Advanced Trigger Conditions" on page 17–26. Figure 17–14 and Figure 17–15 show a data capture with continuous sampling, and the same data pattern using the conditional mode for analysis, respectively.

Figure 17–13. Conditional Storage Qualifier Setup

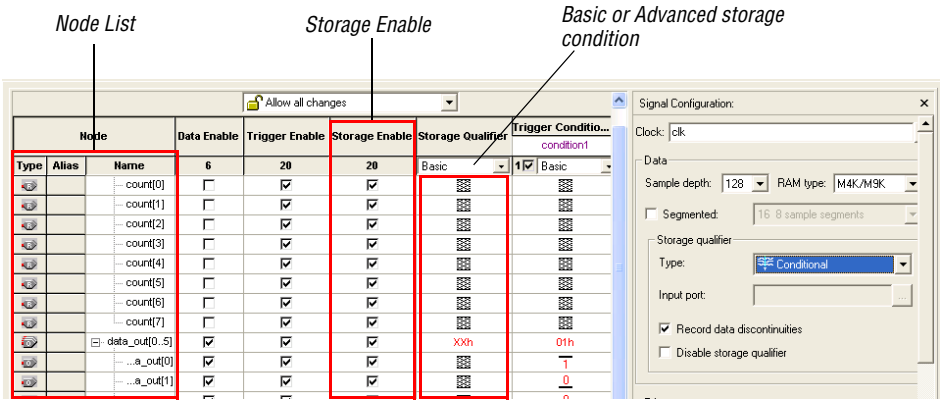
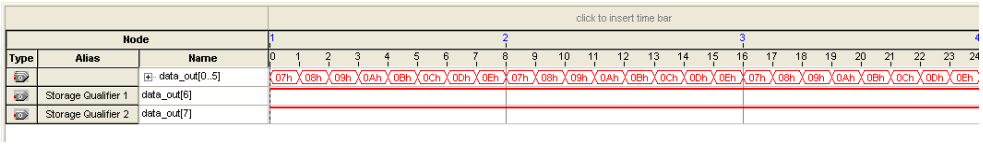


Figure 17–14. Data Acquisition of a Recurring Data Pattern in Continuous Capture Mode (to illustrate Conditional capture)

(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 17–15. 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 is 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 a trigger by pressing the S**top** button if the buffer fails to fill to completion due to a stop condition.

Figure 17–16 shows the Start/Stop mode storage qualifier setup. Figure 17–17 and Figure 17–18 show captures data pattern in continuous capture mode and a data pattern in using the Start/Stop mode for storage qualification.

Figure 17–16. Start/Stop Mode Storage Qualifier Setup

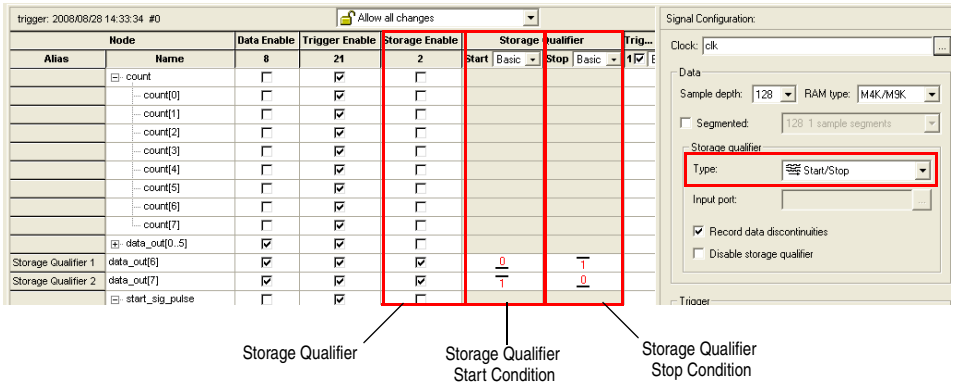


Figure 17–17. Data Acquisition of a Recurring Data Pattern in Continuous Mode (to illustrate Start/Stop mode)

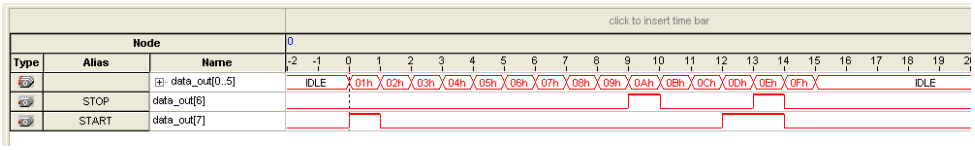
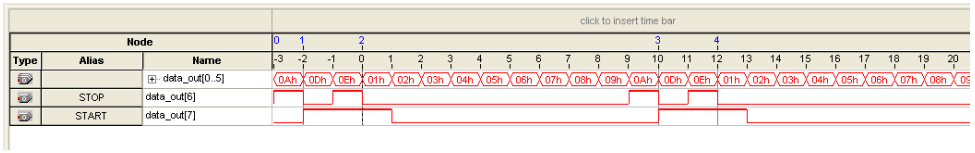


Figure 17–18. Data Acquisition of a Recurring Data Pattern with Start/Stop Storage Qualifier Enabled



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 "State-Based Triggering" on page 17–29.

Showing Data Discontinuities

When you turn on **Record data discontinuities**, the SignalTap II 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

You can turn off the storage qualifier quickly with the **Disable Storage Qualifier** option, 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.

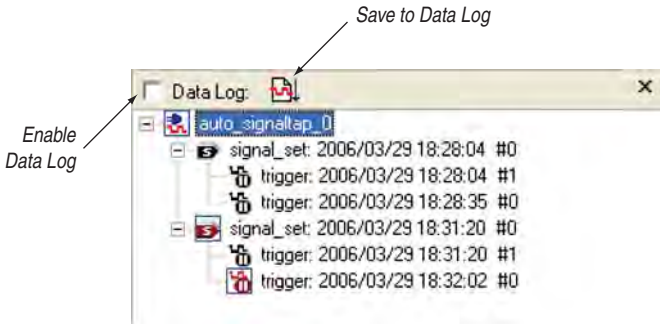
For a detailed explanation of Runtime Reconfigurable options available with the SignalTap II Logic Analyzer, and storage qualifier application examples using runtime reconfigurable options, refer to "Runtime Reconfigurable Options" on page 17–54.

Managing Multiple SignalTap II Files and Configurations

You may have more than one **.stp** 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**, 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** for the logic analyzer to run properly when the device is programmed. Use the Data Log feature and the SOF Manager to manage all of the **.stp** files and their associated settings and programming files.

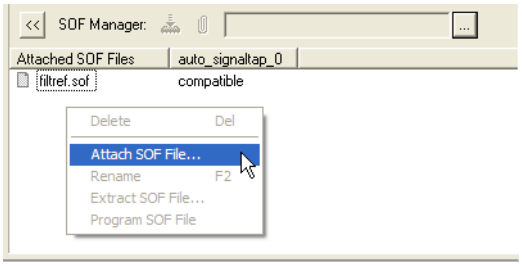
The Data Log allows you to store multiple SignalTap II configurations within a single **.stp**. Figure 17–19 shows two signal set configurations with multiple trigger conditions in one **.stp**. To toggle between the active configurations, double-click on an entry in the Data Log. As you toggle between the different configurations, the signal list and trigger conditions change in the **Setup** tab of the **.stp**. The active configuration displayed in the **.stp** is indicated by the blue square around the signal specified in the Data Log. To store a configuration in the Data Log, on the Edit menu, click **Save to Data Log** or click **Save to Data Log** at the top of the Data Log.

Figure 17–19. Data Log



The SOF Manager allows you to embed multiple **SOF**s into one **.stp**. Embedding an **SOF** in an **.stp** lets you move the **.stp** to a different location, either on the same computer or across a network, without the need to include the associated **.sof** as a separate. To embed a new **SOF** in the **.stp**, right-click in the SOF Manager, and click **Attach SOF File** (Figure 17–20).

Figure 17–20. SOF Manager



As you switch between configurations in the Data Log, you can extract the **SOF** that is compatible with that particular configuration. You can use the programmer in the SignalTap II Logic Analyzer to download the new **SOF** to the FPGA, ensuring that the configuration of your **.stp** always matches the design programmed into the target device.

Define Triggers

When you start the SignalTap II Logic Analyzer, it samples activity continuously from the monitored signals. The SignalTap II Logic Analyzer "triggers"—that is, the logic analyzer 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 specify using the SignalTap II Logic Analyzer on the **Signal Configuration** pane.

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 Logic Analyzer Editor. If you select the **Basic** trigger type, you must specify the trigger pattern for each signal you have added in the **.stp**. To specify 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** that have an associated mnemonic table, you can right-click and select an entry from the table to specify 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 17–57, 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** 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 Logic Analyzer's basic triggering capabilities, you can build more complex triggers with extra logic that enables you to capture data when a combination of conditions exist. If you select the **Advanced** trigger type at the top of the **Trigger Conditions** column in the node list of the SignalTap II Logic Analyzer Editor, a new tab named **Advanced Trigger** appears where you can build a complex trigger expression using a simple GUI. Drag-and-drop operators into the Advanced Trigger Configuration Editor window to build the complex trigger condition in an expression tree. To configure the operators' settings, double-click or right-click the operators that you have placed and select **Properties**. Table 17–3 lists the operators you can use.

Table 17–3. 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 17–3:

(1) For more information about each of these operators, refer to the Quartus II Help.

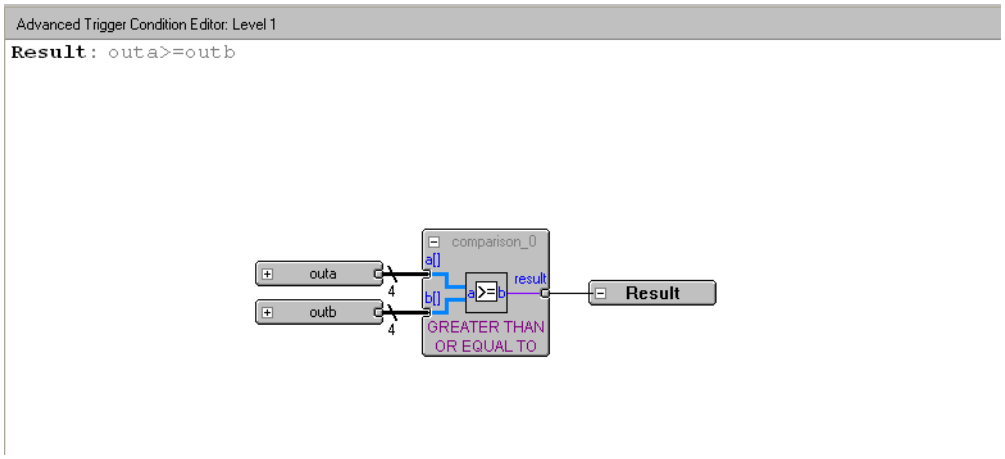
Adding many objects to the Advanced Trigger Condition Editor can make the work space cluttered and difficult to read. To keep objects organized while you build your advanced trigger condition, use the shortcut menu and select **Arrange All Objects**. You can also use the **Zoom-Out** command to fit more objects into the Advanced Trigger Condition Editor window.

Examples of Advanced Triggering Expressions

The following examples show how to use Advanced Triggering:

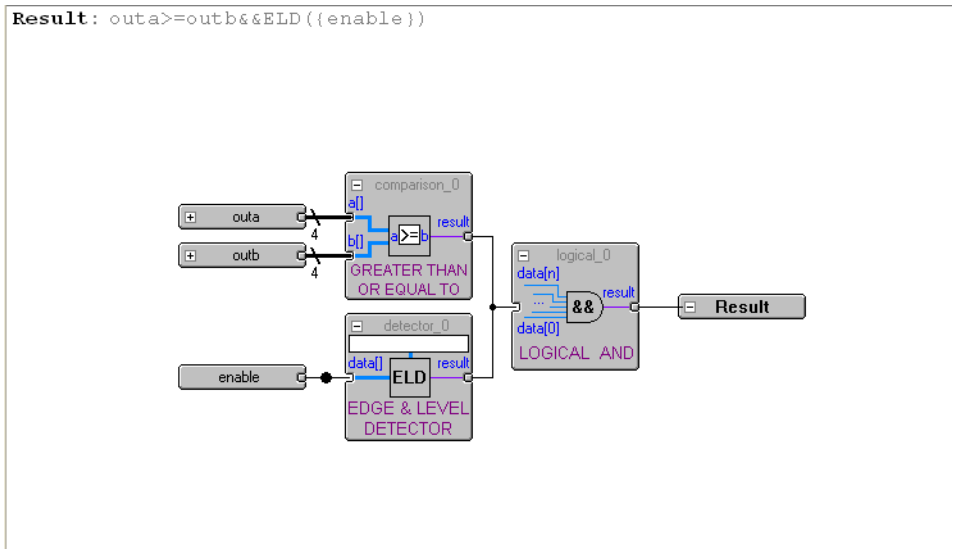
- Trigger when bus `outa` is greater than or equal to `outb` (Figure 17–21).

Figure 17–21. Bus outa is Greater Than or Equal to Bus outb



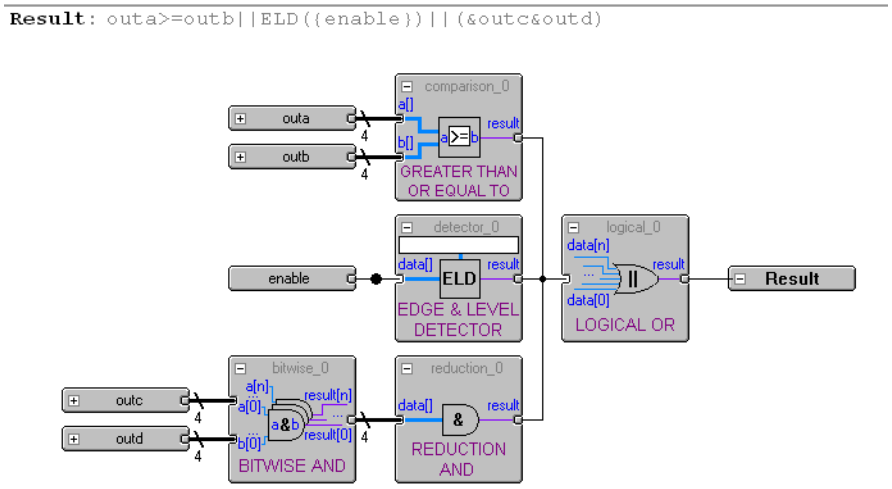
- Trigger when bus `outa` is greater than or equal to bus `outb`, and when the enable signal has a rising edge (Figure 17–22).

Figure 17–22. 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 17–23).

Figure 17–23. Bitwise AND Operation



Trigger Condition Flow Control

The SignalTap II Logic Analyzer offers multiple triggering conditions to give you precise control of the method in which 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 Logic Analyzer **Signal Configuration** pane offers two flow control mechanisms for organizing trigger conditions:

- Sequential Triggering—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.
- State-Based Triggering—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.

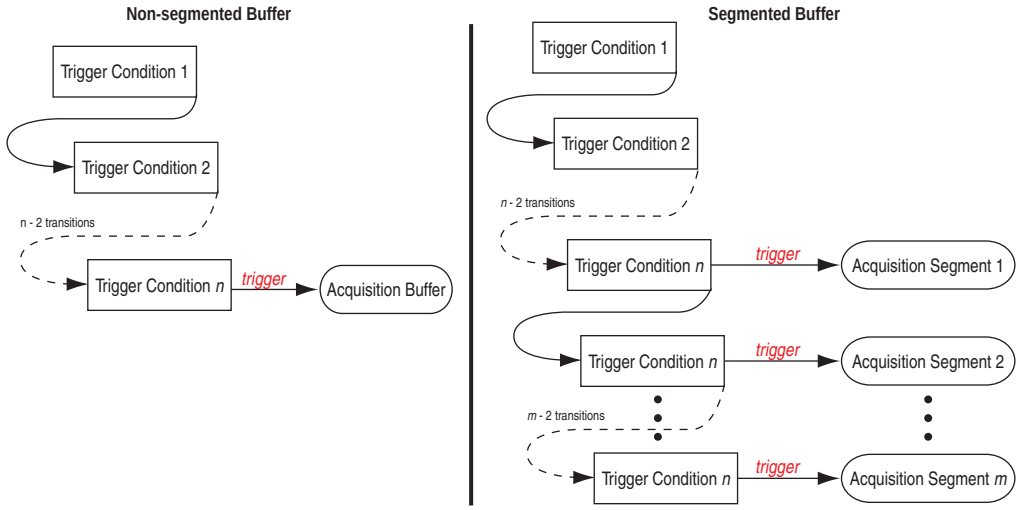
You can use either method with either a segmented or a non-segmented buffer.

Sequential Triggering

Sequential triggering flow allows you to cascade up to 10 levels of triggering conditions. The SignalTap II Logic Analyzer sequentially evaluates each of the triggering conditions. When the last triggering condition evaluates to `TRUE`, the SignalTap II 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 17–24 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 17–24. Sequential Triggering Flow *(Note 1)*, *(2)*



Notes to Figure 17–24:

(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 17–42.

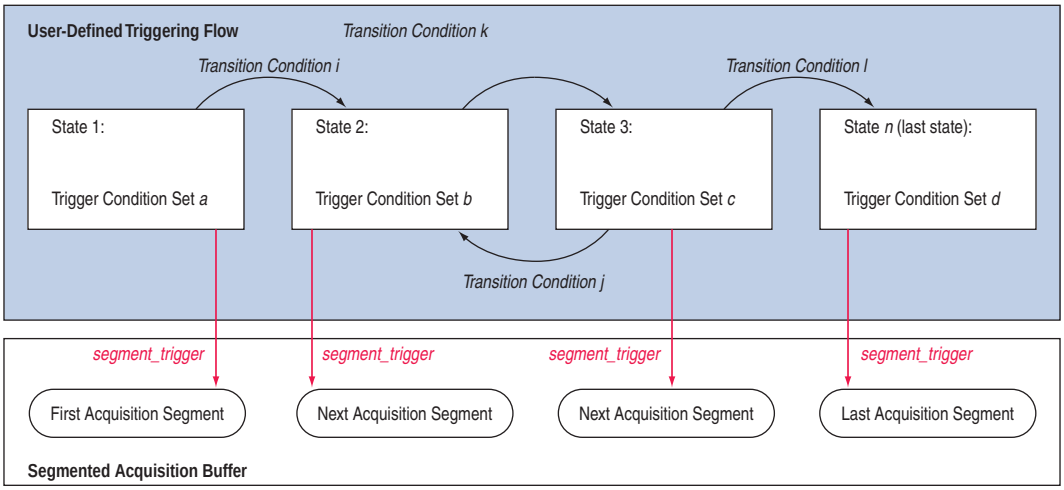
To configure the SignalTap II Logic Analyzer for Sequential triggering, in the SignalTap II editor on the **Trigger flow control** list, select **Sequential**. Select the desired number of trigger conditions from the **Trigger Conditions** list. After you select the desired number of trigger conditions, configure each trigger condition in the node list. To disable any trigger condition, turn on the trigger condition at the top of the column in the node list.

State-Based Triggering

Custom State-based triggering provides the most control over triggering condition arrangement. The State-Based Triggering flow allows you 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 17–25 illustrates the custom State-based triggering flow. Events that trigger the acquisition buffer are organized by a state diagram that you define. 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 17–25. State-Based Triggering Flow *(Note 1)*, *(2)*



Notes to Figure 17–25:

(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 17–42.

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 Logic Analyzer 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 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, select **State-based** on the **Trigger Flow Control** list. (Note that when **Trigger Flow Control** is specified as **Sequential**, the **State-Based Trigger Flow** tab is hidden.)

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 the states. You can adjust the number of available states by using the 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 17–32. You can also refer to *SignalTap II Trigger Flow Description Language* in 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 specify 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 specify a 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.

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 17–4 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 17–54.

Table 17–4. 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 you to modify comparison values in Boolean expressions at runtime. In addition, you can modify the `segment_trigger` and trigger action post-fill count argument at runtime.
Comparison operators	Allows you to modify the operators in Boolean expressions at runtime.
Logical operators	Allows you to modify the logical operators in Boolean expressions at runtime.

You can restrict changes to your SignalTap configuration to include only the options that do not require a recompilation by using the menu above the trigger list in the **Setup** tab. **Allow trigger condition changes only** restricts changes to only the configuration settings that have the **configurable at runtime** specified. 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 17–48.

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 17–1 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 17–1).

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 17–69.

Example 17–1. 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 17–1:

(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` 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 17–5.

Table 17–5. 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 17–6.

Table 17–6. 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 17–6:

(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 17–7 shows the description and syntax of each action.

Table 17–7. Resource Manipulation Action

Action	Description	Syntax
increment	Increments a counter resource by 1	increment <*counter_identifier*>;
decrement	Decrements a counter resource by 1	decrement <*counter_identifier*>;
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 17–8 shows the description and syntax of each action.

Table 17–8. 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 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.

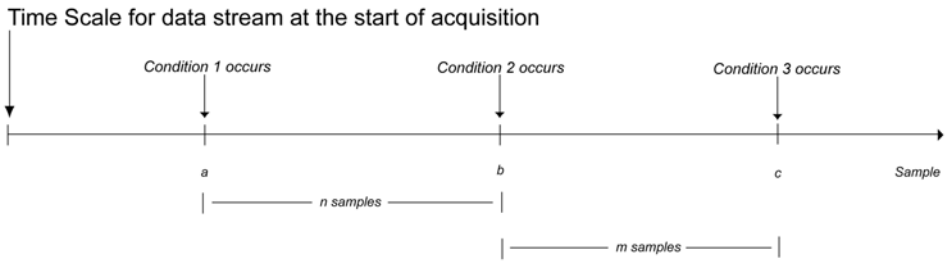
The following examples illustrate the behavior of the State-based trigger flow with the storage qualification commands.

Figure 17–26 shows a hypothetical scenario with three trigger conditions that happen at different times after you click **Start Analysis**. The trigger flow description in Example 17–2, when applied to the scenario shown in Figure 17–26, illustrates the functionality of the storage qualification feature for the state-based trigger flow.

Example 17–2. Trigger Flow Description 1

```
State 1: ST1:
if ( condition1 )
   start_store;
else if ( condition2 )
   trigger value;
else if ( condition3 )
   stop_store;
```

Figure 17–26. Capture Scenario for Storage Qualification with the State-Based Trigger Flow



In this example, the SignalTap II 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 finishes the acquisition and displays 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 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 click **Stop Analysis** to manually stop and force the acquisition to trigger. You can use counter values, flags, and the State diagram to help you perform the trigger flow. The counter values, flags, and the current state are updated in real-time during a data acquisition.

Figure 17–27 and Figure 17–28 show a real data acquisition of the scenario. Figure 17–27 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 17–28 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 17–27. Storage Qualification with Post-Fill Count Value Less than *m* (Acquisition successfully completes)

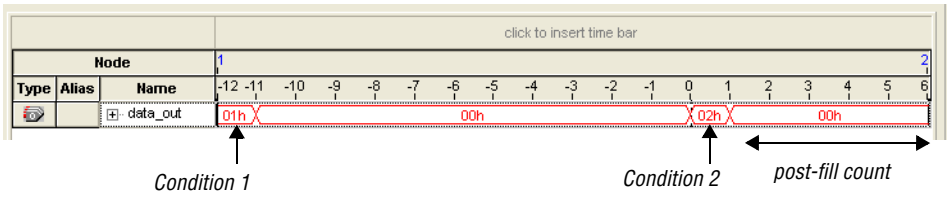
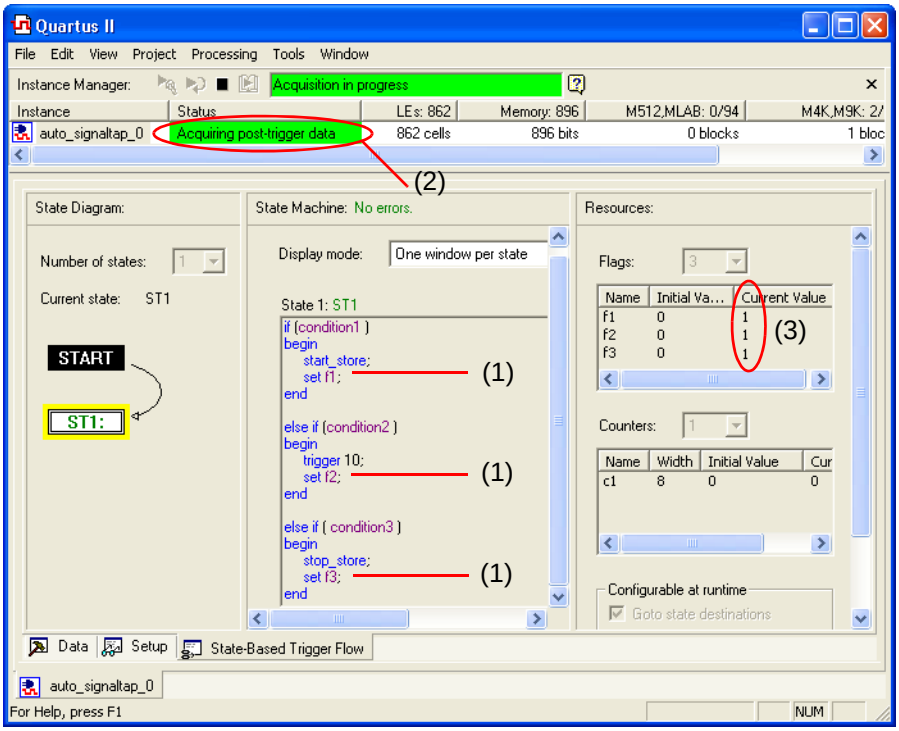
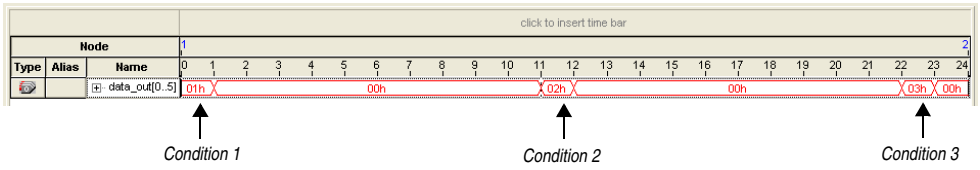


Figure 17–28. Storage Qualification with Post-Fill Count Value Greater than *m* (Acquisition indefinitely paused)



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 are written into the acquisition buffer. Example 17–3 shows a trigger flow description that skips three clock cycles of samples after hitting condition 1. Figure 17–29 shows the data transaction on a continuous capture and Figure 17–31 shows the data capture with the Trigger flow description in Example 17–3 applied.

Example 17–3. 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 17–29. Continuous Capture of Data Transaction for Example 2

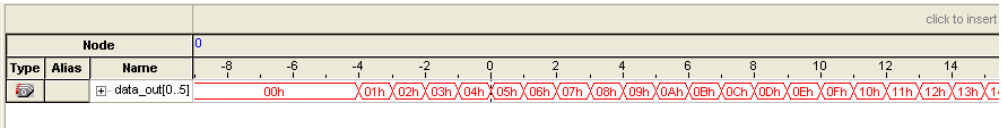
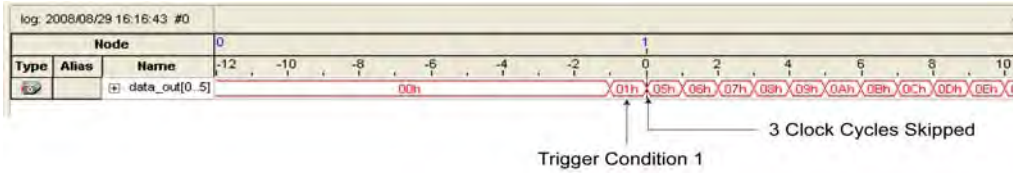


Figure 17–30. Capture of Data Transaction with Trigger Flow Description Applied



Specifying the Trigger Position

The SignalTap II Logic Analyzer allows you to specify the amount of data that is acquired before and after a trigger event. You can specify 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**—Saves signal activity that occurred after the trigger (12% pre-trigger, 88% post-trigger).
- **Center**—Saves 50% pre-trigger and 50% post-trigger data.
- **Post**—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 17–1:

Equation 17–1.

$$\text{Sample Number of Trigger Position} = (N - \text{Post-Fill Count})$$

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 define use of 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 "State-Based Triggering" on page 17–29.

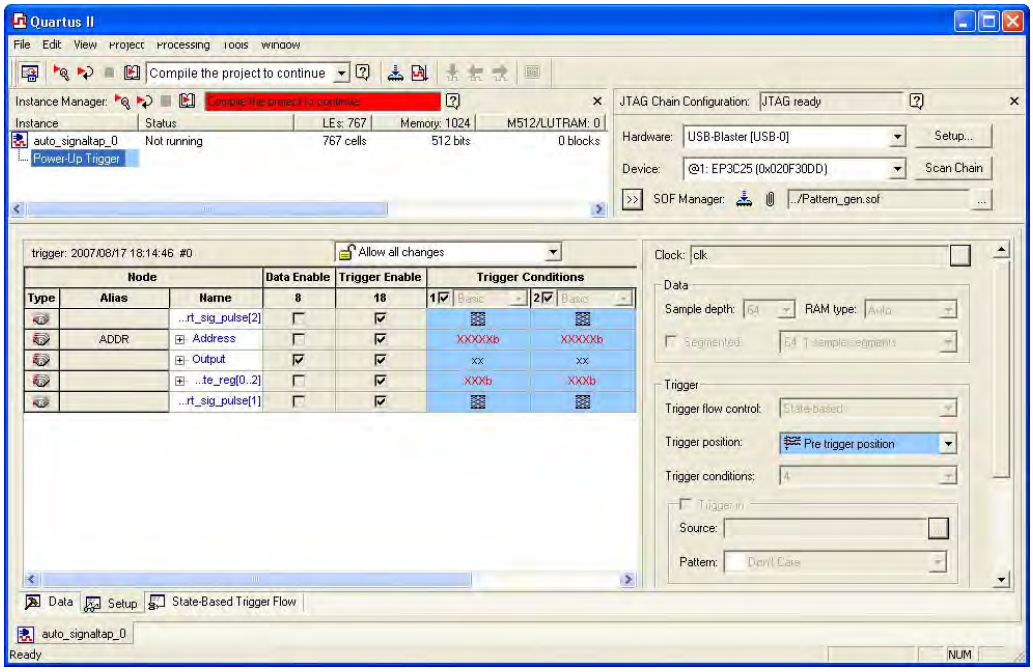
Creating a Power-Up Trigger

Typically, the SignalTap II 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 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** pane. 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 specified in the node list. Figure 17–31 shows the SignalTap II Logic Analyzer Editor when Power-Up Trigger is enabled.

Figure 17–31. SignalTap II Logic Analyzer 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 the trigger as you do with a Run-Time Trigger. Power-Up Trigger conditions that you can adjust are color coded light blue, while Run-Time Trigger conditions you cannot adjust 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. To apply these changes 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 you recompile the SignalTap II Logic Analyzer instance, 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**.

You can also use In-System Sources and Probes in conjunction with the SignalTap II 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*.

Using External Triggers

You can create a trigger input that allows you to trigger the SignalTap II Logic Analyzer from an external source. The external trigger input behaves like trigger condition 1, is evaluated, and must be TRUE before any other configured trigger conditions are evaluated. The logic analyzer supplies a signal to trigger external devices or other SignalTap II Logic Analyzer 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.

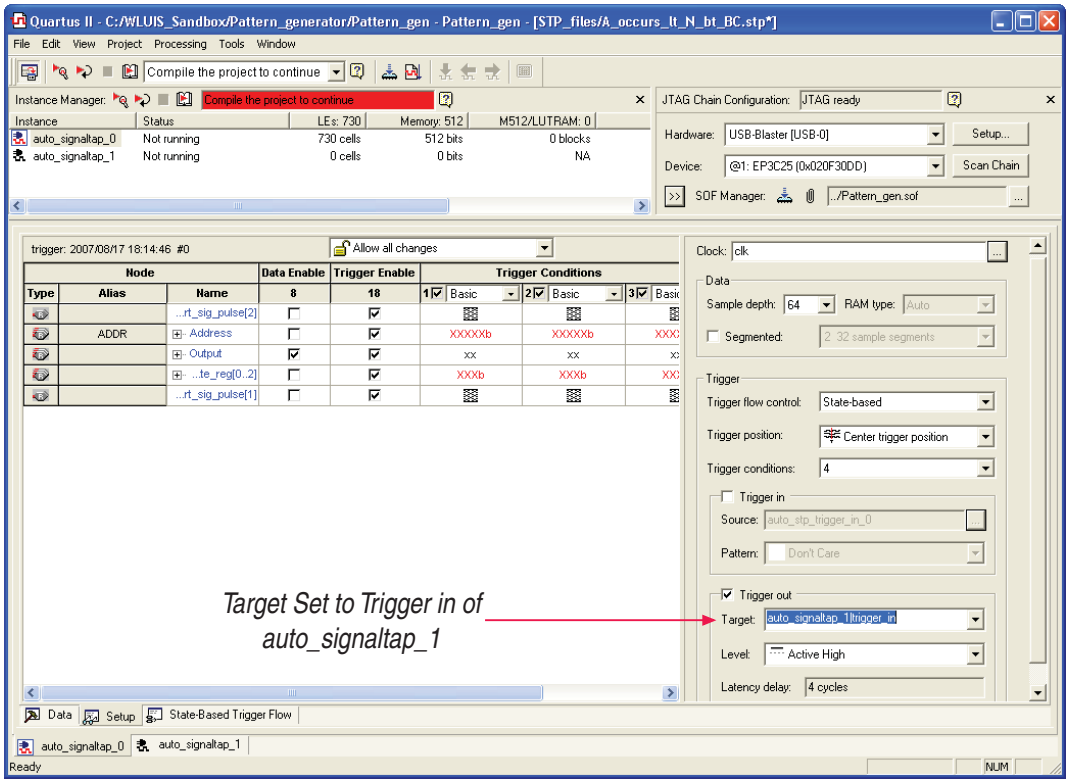
For more information about setting up external triggers, refer to *Signal Configuration Pane* in Quartus II Help.

Using the Trigger Out of One Analyzer as the Trigger In of Another Analyzer

An advanced feature of the SignalTap II 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.

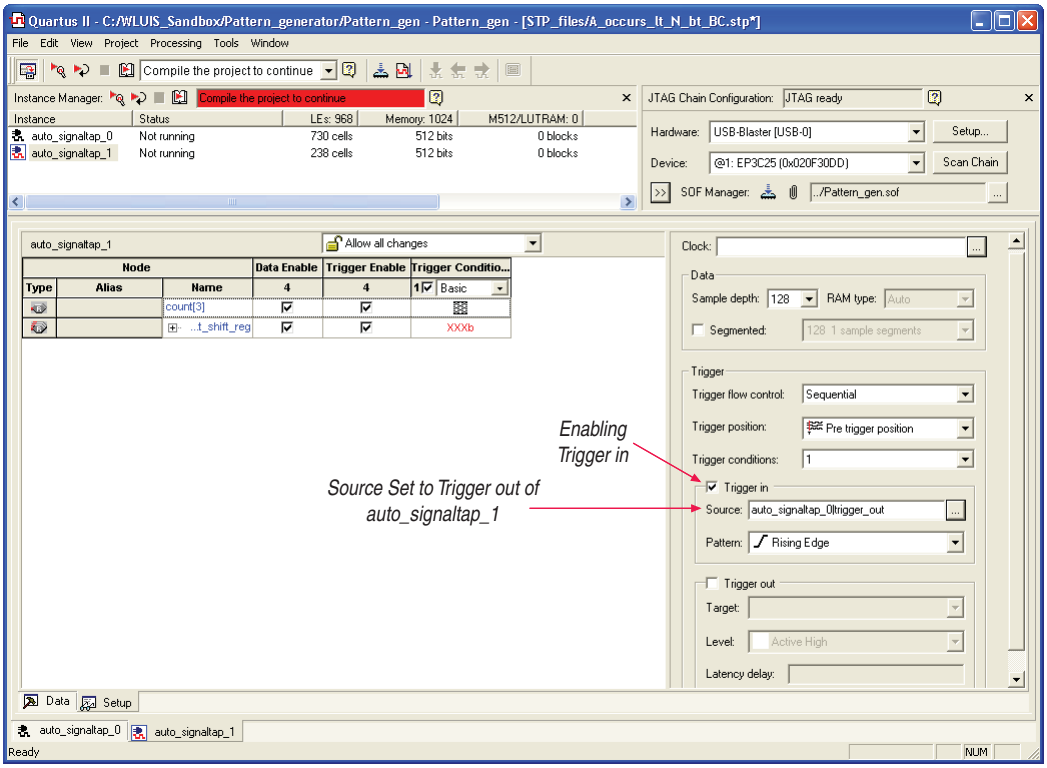
To perform this operation, first turn on **Trigger out** for the source logic analyzer instance. On the **Target** list of the **Trigger out** trigger, select the targeted logic analyzer instance. For example, if the instance named `auto_signaltap_0` should trigger `auto_signaltap_1`, select `auto_signaltap_1|trigger_in` from the list (Figure 17–32).

Figure 17–32. Configuring the Trigger Out Signal



- Turning on **Trigger out** automatically enables the **Trigger in** of the targeted logic analyzer instance and fills in the **Source** field of the **Trigger in** trigger 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 17–33).

Figure 17–33. Configuring the Trigger In Signal



Compile the Design

When you add an **.stp** to your project, the SignalTap II Logic Analyzer becomes part of your design. You must compile your project to incorporate the SignalTap II logic and enable the JTAG connection you use to control the logic analyzer. When you are debugging with a traditional external logic analyzer, you must often make changes to the signals monitored as well as the trigger conditions. Because these adjustments require that you recompile your design when using the SignalTap II Logic Analyzer, use the SignalTap II Logic Analyzer feature along with incremental compilation in the Quartus II software to reduce recompilation time.

For more information on reducing your recompilation burden with incremental compilation, refer to *Using the Incremental Compilation Design Flow* in Quartus II Help.

Faster Compilations with Quartus II Incremental Compilation

When you compile your design with an **.stp**, 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 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 Logic Analyzer to your design without recompiling your original source code. Incremental compilation is also useful when you want to modify the configuration of the **.stp**. 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 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 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

Your project is fully compiled the first time, establishing the design partitions you have created. When enabled for your design, the SignalTap II Logic Analyzer is always a separate partition. After the first compilation, you can use the SignalTap II Logic Analyzer to analyze signals from the post-fit netlist. If your partitions are designed correctly, subsequent compilations due to SignalTap II Logic Analyzer settings take less time.

The netlist type for the top-level partition defaults to **source**. To take advantage of incremental compilation, specify the Netlist types for the partitions you wish to tap as **Post-fit**.

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 Logic Analyzer

The SignalTap II Logic Analyzer is automatically configured to work with the incremental compilation flow. For all signals that you want to connect to the SignalTap II 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 Logic Analyzer.

CAUTION 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 specified as 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 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.

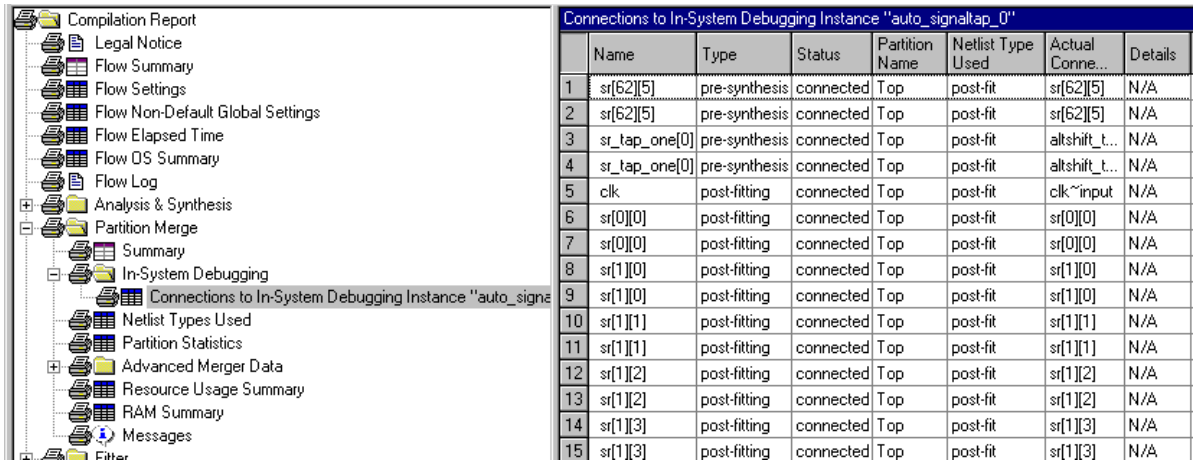
If you do use incremental compile flow with the SignalTap II Logic Analyzer and source file changes are necessary, be aware that you may have to remove compiler-generated post-fit net names. Source code changes force the affected partition to go through resynthesis. During synthesis, the compiler cannot find compiler-generated net names from a previous compilation.

Altera recommends using only registered and user-input signals as debugging taps in your **.stp** 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** limits the changes you need to make to your SignalTap II Logic Analyzer configuration.

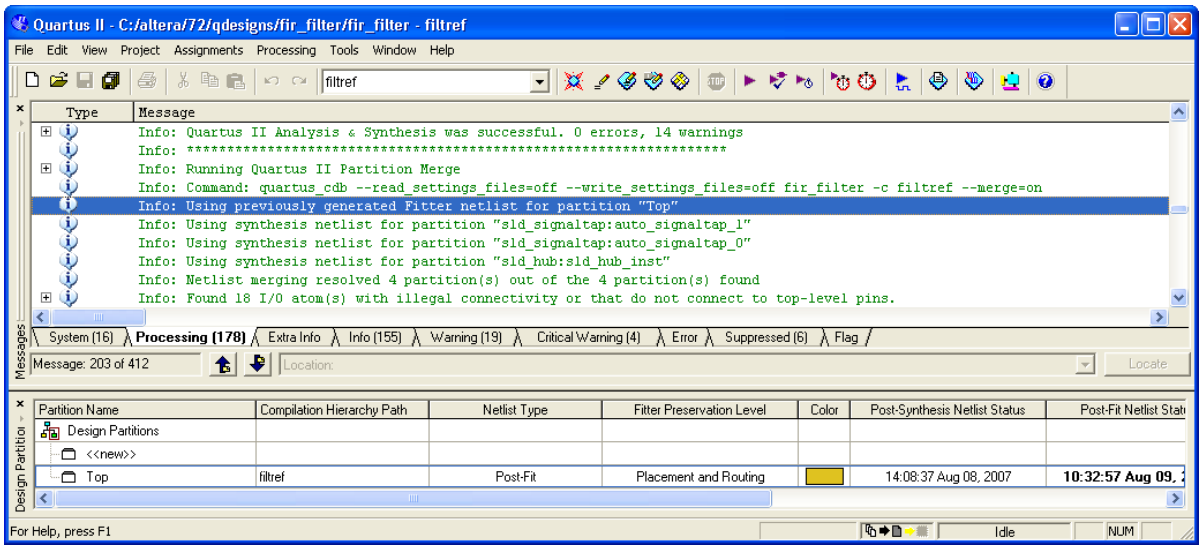
You can check the nodes that are connected to each SignalTap II instance using the In-System Debugging compilation reports. These reports list each node name you selected to connect to a SignalTap II instance, the netlist type used for the particular connection, and the actual node name used after compilation. 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 17–34 shows an example of an In-System Debugging compilation report for a design using incremental compilation.

Figure 17–34. 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 17–35 shows an example of the messages displayed.

Figure 17–35. 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**, only the SignalTap II design partition must be recompiled, reducing your recompilation time.

Preventing Changes Requiring Recompilation

You can configure the **.stp** 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 *Setup Tab (SignalTap II Logic Analyzer)* in Quartus II Help.

Timing Preservation with the SignalTap II 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 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 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 Logic Analyzer instance 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**.
- Minimize the number of combinational signals you add to your **.stp** 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 Logic Analyzer, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Performance and Resource Considerations

There is a necessary trade-off between the runtime flexibility of the SignalTap II Logic Analyzer, the timing performance of the SignalTap II Logic Analyzer, and resource usage. The SignalTap II 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 complete debugging as quickly as possible; however, you can adjust these settings to determine whether there is a more optimal configuration for your design.

The following tips 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 Logic Analyzer consumes if your design is resource-constrained.

If SignalTap II logic is part of your critical path, follow these tips to speed up the performance of the SignalTap II Logic Analyzer:

- **Disable runtime configurable options**—Certain resources are allocated to accommodate for runtime flexibility. If you use 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 signals that you add to the **.stp** 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 logic to a gate-based triggering condition, such as a basic trigger condition or a logical reduction operator in the advanced trigger tab, turn on **Perform register retiming**. This can help balance combinational logic across LABs.

If your design is resource constrained, follow these tips to reduce the amount of logic or memory used by the SignalTap II 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 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 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 you compile your project, including the SignalTap II Logic Analyzer, configure the FPGA target device. When you are using the SignalTap II Logic Analyzer for debugging, configure the device from the **.stp** instead of the Quartus II Programmer. Because you configure from the **.stp**, you can open more than one **.stp** and program multiple devices to debug multiple designs simultaneously.

The settings in an **.stp** must be compatible with the programming **.sof** used to program the device. An **.stp** is considered compatible with an **.sof** 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 is 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 Logic Analyzer Editor.

To ensure programming compatibility, make sure to program your device with the latest **.sof** created from the most recent compilation. Checking whether or not a particular **SOF** is compatible with the current SignalTap II configuration is achieved quickly by attaching the **SOF** to the SOF manager. For more details about using the SOF manager, refer to "Managing Multiple SignalTap II Files and Configurations" on page 17–24.

Before starting a debugging session, do not make any changes to the **.stp** settings that would requires recompiling the project. You can check the SignalTap II status display at the top of the **Instance Manager** pane to verify whether a change you made requires recompiling the project, producing a new **.sof**. This gives you the opportunity to undo the change, so that you do not need to recompile your project. To prevent any such changes, select **Allow trigger condition changes only** to lock the **.stp**.

Although the Quartus II project is not required when using an **.stp**, it is recommended. The project database contains information about the integrity of the current SignalTap II Logic Analyzer session. Without the project database, there is no way to verify that the current **.stp** matches the **.sof** that is downloaded to the device. If you have an **.stp** that does not match the **.sof**, incorrect data is captured in the SignalTap II Logic Analyzer.

For instructions on programming devices in the Quartus II software, refer to *Running the SignalTap II Logic Analyzer* in Quartus II Help.

Querying the SignalTap II Instance on a Programmed Device

Each SignalTap II instance is defined by unique ID code and a CRC value on the JTAG chain. The **.stp** uses the CRC value from each SignalTap II instance to determine compatibility with the configuration defined within the SignalTap II file.

You can use the TCL commands in the JTAG package of the `quartus_stp` command line executable to query for the active SignalTap II instances in a programmed device.

The following script returns all the SignalTap II instances and their associated CRC values contained on a device. The returned CRC value can be compared with the CRC values for each SignalTap II instance in the Quartus II project listed in the JTAG debug information (**.jdi**) file.

The **.jdi** file is an xml file format and lists all the parameters for each Altera JTAG based on-chip debugging application in the project. This file is located in the project directory. Figure 17–36 shows highlighted code from a sample **.jdi** showing the CRC value for a particular SignalTap II instance.

The sample script in Example 17–5 on page 17–52 returns the CRC value in hexadecimal format per SignalTap II instance.

Figure 17–36. Segment of Sample .jdi File Listing CRC Values

```
<parameters>
        <parameter name="lpm_type" type="string" value="sld_signaltap"/>
        <parameter name="sld_node_info" type="unknown" value="805334528"/>
        <parameter name="SLD_IP_VERSION" type="dec" value="6"/>
        <parameter name="SLD_IP_MINOR_VERSION" type="dec" value="0"/>
        <parameter name="SLD_COMMON_IP_VERSION" type="dec" value="0"/>
        <parameter name="sld_data_bits" type="unknown" value="14"/>
        <parameter name="sld_trigger_bits" type="unknown" value="14"/>
        <parameter name="SLD_NODE_CRC_BITS" type="dec" value="32"/>
        <parameter name="sld_node_crc_hiword" type="unknown" value="53125"/>
        <parameter name="sld_node_crc_loword" type="unknown" value="42125"/>
```

Example 17–4. Script to Produce Hexadecimal CRC Values (Part 1 of 2) *(Note 1)*

```
#All JTAG based IP contain a unique identifier.
#These ID values are also found in the .jdi file
# The identifiers below are used to filter STP instances.
# Note that version number must match the value contained in the .JDI file
set version 6
set mfg_id 110
set type_id 0
puts "Programming Hardwares:"
foreach hardware_name [get_hardware_names] {
   puts $hardware_name
   if { [string match "USB-Blaster*" $hardware_name] } {
      set usbblaster_name $hardware_name
   }
}
puts "\nSelect JTAG chain connected to $usbblaster_name.\n";

# List all devices on the chain, and select the first device on the chain.
puts "\nDevices on the JTAG chain:"
foreach device_name [get_device_names -hardware_name $usbblaster_name] {
   puts $device_name
   if { [string match "@1*" $device_name] } {
      set test_device $device_name
   }
}
puts "\nSelect device: $test_device.\n";

# Open device
open_device -hardware_name $usbblaster_name -device_name $test_device

puts "Query SignalTap II instance information from the device:"
puts "============================================================"

device_lock -timeout 10000
set total_inst 0
for {set inst_index 0} {$inst_index <= 127} {incr inst_index} {
   set crc_val ""

   # Test the existance of the instance and set the CRC value query instruction
   set ret_code [
      catch {
         device_virtual_ir_shift -ir_value 32 -no_captured_ir_value -instance_index
$inst_index -mfg_id $mfg_id -type_id $type_id -version
      $version}
   ]
```

Example 17–5. Script to Produce Hexadecimal CRC Values (Part 2 of 2) *(Note 1)*

```
   # Retrieve the CRC value nibble by nibble
   if { $ret_code == 0 } {
      incr total_inst
      for {set nibble_index 0} {$nibble_index < 8} {incr nibble_index} {
         set nibble [device_virtual_dr_shift  -length 4 -value_in_hex -instance_index
$inst_index -mfg_id $mfg_id -type_id $type_id -version $version]

         set crc_val "$nibble$crc_val"
      }

      puts "SignalTap II Instance $inst_index CRC Value: 0x$crc_val"
   }
}
device_unlock
close_device

puts "========================================================="
puts "Total SignalTap II Instances Found: $total_inst\n\n"
```

Note to Example 17–4 and Example 17–5:

(1) This code sample is a script that returns the CRC value per SignalTap II instance in hexadecimal format.

(2) The version numbers referenced in the code must match the version numbers in the .jdi.

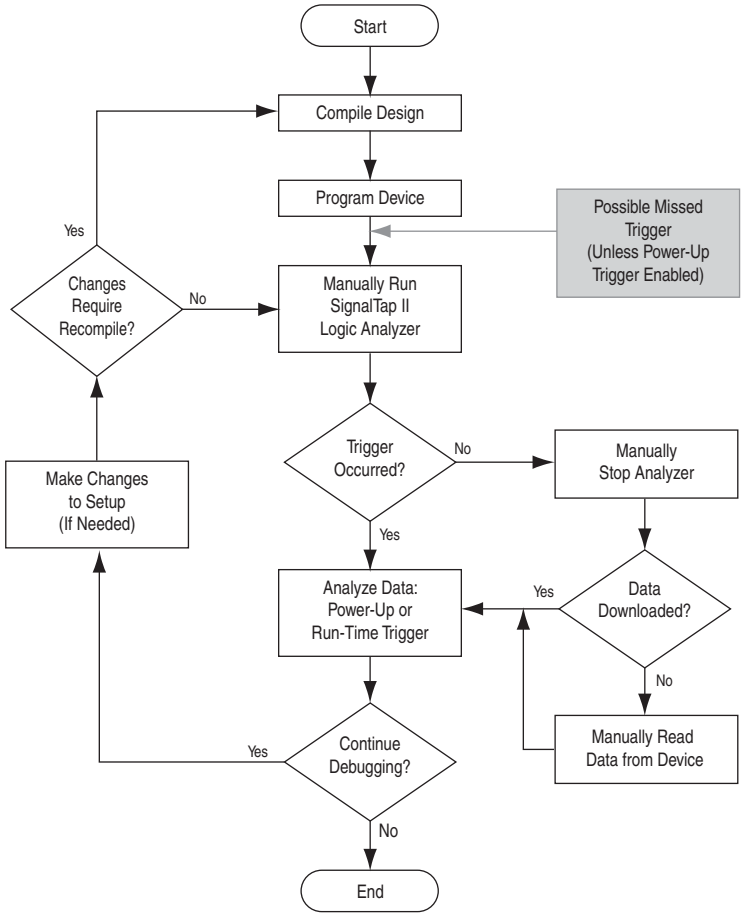
To run the script, save the example code into a file with the extension **.tcl**, or download sample code from the *On Chip Debugging Design Examples* page on the Altera website. Source the script with **quartus_stp** executable from the command line:

```
(syntax) :   quartus_stp -t <filename>.tcl
```

Run the SignalTap II Logic Analyzer

After the device is configured with your design that includes the SignalTap II Logic Analyzer, perform debugging operations in a manner similar to when you use an external logic analyzer. You initialize 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** with the JTAG connection. You can also perform the equivalent of a force trigger instruction that lets you view the captured data currently in the buffer without a trigger event occurring. Figure 17–37 illustrates a flow that shows how you operate the SignalTap II 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 17–37. Power-Up and Runtime Trigger Events Flowchart



For information on running the analyzer from the **Instance Manager** pane, refer to *Running the SignalTap II Logic Analyzer* in Quartus II Help.

You can also use In-System Sources and Probes in conjunction with the SignalTap II Logic Analyzer to force trigger conditions. The In-System Sources and Probes feature allows you to drive and sample values on to selected signals 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** are changeable without recompiling your design when you use Runtime Trigger mode. Runtime Reconfigurable features are described in Table 17–9.

Table 17–9. Runtime Reconfigurable Features

Runtime Reconfigurable Setting	Description
Basic Trigger Conditions and Basic Storage Qualifier Conditions	All signals that have the Trigger condition turned on 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 turned on in the **Object Properties** dialog box.
Switching between a storage-qualified and a continuous acquisition	Within any storage-qualified mode, you can switch to continuous capture mode without recompiling the design. To enable this feature, turn on **disable storage qualifier**.
State-based trigger flow parameters	Table 17–4 lists 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. You may experience a slight impact to the performance and logic utilization of the SignalTap II IP core. You can turn off Runtime re-configurability for Advanced Trigger Conditions and the State-based trigger flow parameters, boosting performance and decreasing area utilization.

You can configure the **.stp** 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 17–6 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 17–6. 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 17–6:

(1) *m* x *n* must equal the sample depth to efficiently use the space in the sample buffer.

Figure 17–38 shows a segmented buffer described by the trigger flow in Example 17–6.

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 17–38. Segmented Buffer Created with Storage Qualifier and State-Based Trigger *(Note 1)*

Note to Figure 17–38:

(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 17–7 shows a modified description of Example 17–6 with an additional state inserted. You use this extra state to specify a different trigger condition that does not use the storage qualifier feature. You insert status flags into the conditional statements to control the execution of the trigger flow.

Example 17–7. Modified Trigger Flow Description of Example 17–6 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 17–10 describes the text messages that might appear in the SignalTap II Status Indicator in the **Instance Manager** pane before, during, and after a data acquisition. Use these messages to monitor the state of the logic analyzer or what operation it is performing.

Table 17–10. 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 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 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 Logic Analyzer is acquiring the post-trigger data. The amount of post-trigger data collected is you define 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 Logic Analyzer is waiting for you to initialize the analyzer.

Note to Table 17–10:

(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.

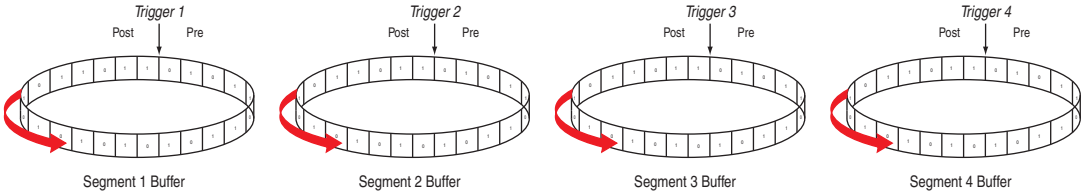
For information about what you can do with captured data, refer to *Analyzing Data in the SignalTap II Logic Analyzer* in Quartus II Help.

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 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 17–39 shows a segmented acquisition buffer with four segments represented as four separate non-segmented buffers.

Figure 17–39. Segmented Acquisition Buffer

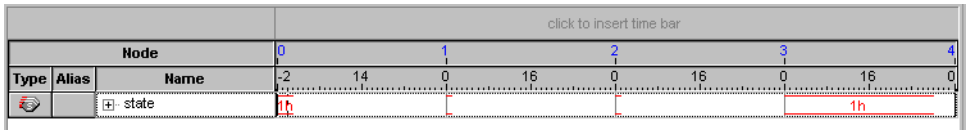


The SignalTap II Logic Analyzer finishes an acquisition with a segment, and advances to the next segment to start a new acquisition. Depending on when a trigger condition occurs, it may affect the way the data capture appears in the waveform viewer. Figure 17–39 illustrates the method in which data is captured. The Trigger markers in Figure 17–39—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 use 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 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 allows the SignalTap II Logic Analyzer to accurately capture all of the trigger conditions that have occurred. Samples that have not been used appear as a blank space in the waveform viewer.

Figure 17–40 shows an example of a capture using sequential flow control with the trigger condition for each segment specified as **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 specified as 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 are left over from the pre-trigger samples that the SignalTap II Logic Analyzer allocated to the buffer.

Figure 17–40. Segmented Capture with Preemption of Acquisition Segments *(Note 1)*



Note to Figure 17–40:

(1) A segmented acquisition buffer using the sequential trigger flow with a trigger condition specified as 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 specific to your debugging requirements, you can help maximize the use of the allocated buffer space.

Differences in Pre-fill Write Behavior Between Different Acquisition Modes

The SignalTap II Logic Analyzer uses one of the following three modes when writing into the acquisition memory:

- Non-segmented buffer
- Non-segmented buffer with a storage qualifier
- Segmented buffer

There are subtle differences in the amount of data captured immediately after running the SignalTap II Logic Analyzer and before any trigger conditions occur. A non-segmented buffer, running in continuous mode, completely fills the buffer with sampled data before evaluating any trigger conditions. Thus, a non-segmented capture without any storage qualification enabled always shows a waveform with a full buffer's worth of data captured.

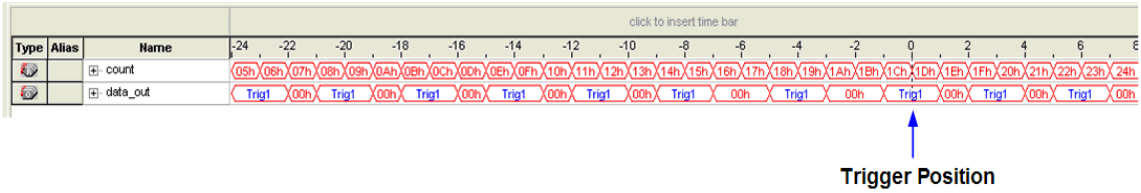
Filling the buffer provides you with as much data as possible within the capture window. The buffer gets pre-filled with data samples prior to evaluating the trigger condition. As such, SignalTap requires that the buffer be filled at least once before any data can be retrieved through the JTAG connection and prevents the buffer from being dumped during the first acquisition prior to a trigger condition when you perform a **Stop Analysis**.

For segmented buffers and non-segmented buffers using any storage qualification mode, the SignalTap II Logic Analyzer immediately evaluates all trigger conditions while writing samples into the acquisition memory. The logic analyzer evaluates each trigger condition before acquiring a full buffer's worth of samples. This evaluation is especially important when using any storage qualification on the data set. The logic analyzer may miss a trigger condition if it waits until a full buffer's worth of data is captured before evaluating any trigger conditions.

If the trigger event occurs on any data sample before the specified amount of pre-trigger data has occurred, then the SignalTap II Logic Analyzer triggers and begins filling memory with post-trigger data, regardless of the amount of pre-trigger data you specify. For example, if you set the trigger position to 50% and set the logic analyzer to trigger on a processor reset, start the logic analyzer, and then power on your target system, the logic analyzer triggers. However, the logic analyzer memory is filled only with post-trigger data, and not any pre-trigger data, because the trigger event, which has higher precedence than the capture of pre-trigger data, occurred before the pre-trigger condition was satisfied.

Figure 17–41 and Figure 17–42 on page 17–60 show the difference between a non-segmented buffer in continuous mode and a non-segmented buffer using a storage qualifier. The logic analyzer for the waveforms below is configured with a sample depth of 64 bits, with a trigger position specified as **Post trigger position**.

Figure 17–41. SignalTap II Logic Analyzer Continuous Data Capture *(Note 1)*



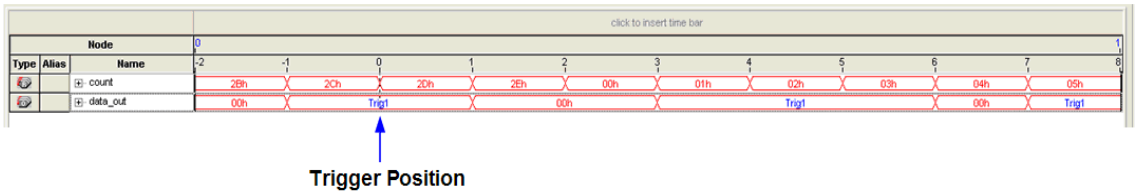
Note to Figure 17–41:

(1) Continuous capture mode with post-trigger position.

(2) Capture of a recurring pattern using a non-segmented buffer in continuous mode. The SignalTap II Logic Analyzer is configured with a basic trigger condition (shown in the figure as "Trig1") with a sample depth of 64 bits.

Notice in Figure 17–41 that Trig1 occurs several times in the data buffer before the SignalTap II Logic Analyzer actually triggers. A full buffer's worth of data is captured before the logic analyzer evaluates any trigger conditions. After the trigger condition occurs, the logic analyzer continues acquisition until it captures eight additional samples (12% of the buffer, as defined by the "post-trigger" position).

Figure 17–42. SignalTap II Logic Analyzer Conditional Data Capture *(Note 1)*



Note to Figure 17–42:

(1) Conditional capture, storage always enabled, post-fill count.

(2) SignalTap II Logic Analyzer capture of a recurring pattern using a non-segmented buffer in conditional mode. The logic analyzer is configured with a basic trigger condition (shown in the figure as "Trig1"), with a sample depth of 64 bits. The “Trigger in” condition is specified as "Don't care", which means that every sample is captured.

Notice in Figure 17–42 that the logic analyzer triggers immediately. As in Figure 17–41, the logic analyzer completes the acquisition with eight samples, or 12% of 64, the sample capacity of the acquisition buffer.

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** 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 the table 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.

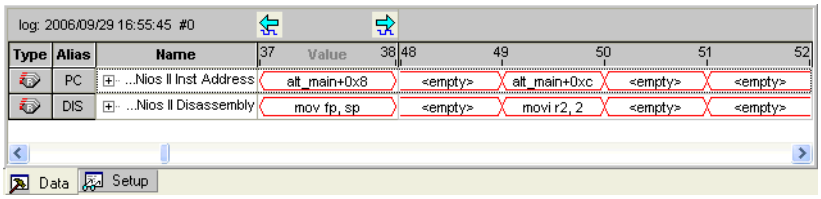
You use the labels you create in a table in different ways on the **Setup** and **Data** tabs. On the **Setup** tab, you can create basic triggers with meaningful names by right-clicking an entry in the **Trigger Conditions** column and selecting a label from the table you assigned to the signal group. 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**, mnemonic tables for the added signals are automatically created and assigned to the signals defined in the plug-in. To enable these mnemonic tables manually, right-click on the name of the signal or signal group. On the **Bus Display Format** shortcut menu, then click the name of the mnemonic table that matches the plug-in.

As an example, the Nios II plug-in helps you to monitor your design's signal activity as code is executed. If you set up the logic analyzer to trigger on a function name in your Nios II code based on data from an **.elf**, 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 17–43. Captured data samples around the trigger are referenced as offset addresses from the trigger function name.

Figure 17–43. 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 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 Logic Analyzer in one of the Quartus II software tools or your design files, right-click on the signal in the **.stp**, and click **Locate in *<tool name>***.

You can locate a signal from the node list with the following tools:

- 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 SignalTap II Logic 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 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. To open the **Data Log** pane, on the View menu, select **Data Log**. To turn on data logging, turn on **Enable data log** in the **Data Log** (Figure 17–19). To recall and activate a data log for a given trigger set, double-click the name of the data log in the list.

You can use the Data Log feature 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 17–24.

Exporting Captured Data to Other File Formats

You can export captured data to the following file formats, for use 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 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 in the Quartus II software, on the File menu, select **Create/Update** and click **Create SignalTap II List File**.

Other Features

The SignalTap II 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 Logic Analyzer directly into a matrix in the MATLAB environment. If you use the MATLAB MEX function in a loop, you can perform as many acquisitions in the same amount of time as you can when using SignalTap II in the Quartus II software environment.

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.
2. In the node list in the **Data** tab of the SignalTap II Logic Analyzer 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 Logic Analyzer and transferred into the MATLAB MEX function are limited to a width of 32 signals. If you want to use the MATLAB 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** and compile your design. Program your device and run the SignalTap II Logic Analyzer to ensure your trigger conditions and signal acquisition work 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 MATLAB MEX function to open the JTAG connection to the device and run the SignalTap II Logic Analyzer to acquire data. When you finish acquiring data, close the JTAG 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 you must assign a filename, for example, *<stp filename>* as a requirement of the MATLAB MEX function. Other MATLAB MEX function options are described in Table 17–11.

Table 17–11. 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**, `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**. 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 with the following command:

```
alt_signaltap_run('END_CONNECTION'); ↵
```

For more information about the use of MATLAB 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 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 standalone version of the SignalTap II 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 Logic Analyzer

You can use the SignalTap II 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 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 standalone version of the SignalTap II 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.

For information about enabling remote access to a JTAG server, refer to *Using the JTAG Server* in Quartus II Help.

Using the SignalTap II 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 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 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 Logic Analyzer. After the FPGA is configured with a SignalTap II Logic Analyzer instance in the design, when you open the SignalTap II Logic Analyzer in the Quartus II software, you then scan the chain and are ready to acquire data with the JTAG connection.

Backward Compatibility with Previous Versions of Quartus II Software

You can open an **.stp** created in a previous version in a current version of the Quartus II software. However, opening an **.stp** modifies it so that it cannot be opened in a previous version of the Quartus II software.

If you have a Quartus II project file from a previous version of the software, you may have to update the **.stp** configuration file to recompile the project. You can update the configuration file by opening the SignalTap II Logic Analyzer. If you need to update your configuration, a prompt appears asking if you would like to update the **.stp** 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 at the command prompt:

```
quartus_sh --qhelp ↵
```

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook* or *About Tcl Scripting* in Quartus II Help.

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 Logic Analyzer using the command prompt, use the `quartus_stp` command. Table 17–12 shows the options that help you use the `quartus_stp` executable.

Table 17–12. SignalTap II Command-Line Options

Option	Usage	Description
`stp_file`	`quartus_stp --stp_file <stp_filename>`	Assigns the specified **.stp** to the `USE_SIGNALTAP_FILE` in the **.qsf**.
`enable`	`quartus_stp --enable`	Creates assignments to the specified **.stp** in the **.qsf** and changes `ENABLE_SIGNALTAP` to `ON`. The SignalTap II Logic Analyzer is included in your design the next time the project is compiled. If no **.stp** is specified in the **.qsf**, the `--stp_file` option must be used. If the `--enable` option is omitted, the current value of `ENABLE_SIGNALTAP` in the **.qsf** is used.
`disable`	`quartus_stp --disable`	Removes the **.stp** reference from the **.qsf** and changes `ENABLE_SIGNALTAP` to `OFF`. The SignalTap II 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** is used.
`create_signaltap_hdl_file`	`quartus_stp --create_signaltap_hdl_file`	Creates an **.stp** representing the SignalTap II instance in the design generated by the SignalTap II 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** properly. Analogous to the **Create SignalTap II File from Design Instance(s)** command in the Quartus II software.

Example 17–8 illustrates how to compile a design with the SignalTap II Logic Analyzer at the command line.

Example 17–8.

```
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 Logic Analyzer to compile properly into your design.

Example 17–9 shows how to create a new **.stp** after building the SignalTap II Logic Analyzer instance with the MegaWizard Plug-In Manager.

Example 17–9.

```
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 Quartus II software. They must be executed from the command line with the **quartus_stp** executable. To execute a Tcl file that has SignalTap II Logic Analyzer Tcl commands, use the following command:

`quartus_stp -t <Tcl file>` ↵

Table 17–13 shows the Tcl commands that you can use with SignalTap II Logic Analyzer.

Table 17–13. SignalTap II Tcl Commands (Part 1 of 2)

Command	Argument	Description
`open_session`	`-name <stp_filename>`	Opens the specified **.stp**. 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.

Table 17–13. SignalTap II Tcl Commands (Part 2 of 2)

Command	Argument	Description
`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**. You cannot run the analyzer after you close the **.stp**.

For more information about SignalTap II Tcl commands, refer to the Quartus II Help.

Example 17–10 is an excerpt from a script you can use to continuously capture data. Once the trigger condition is met, the data is captured and stored in the data log.

Example 17–10.

```
#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** that you used to capture data to examine the contents of the Data Log.

Design Example: Using SignalTap II 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 you to press a button. After you press a button, the processor initiates a DMA transfer, which you analyze using the SignalTap II Logic Analyzer.

For more information about this example and using the SignalTap II Logic Analyzer with SOPC builder systems, refer to *AN 323: Using SignalTap II 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 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 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

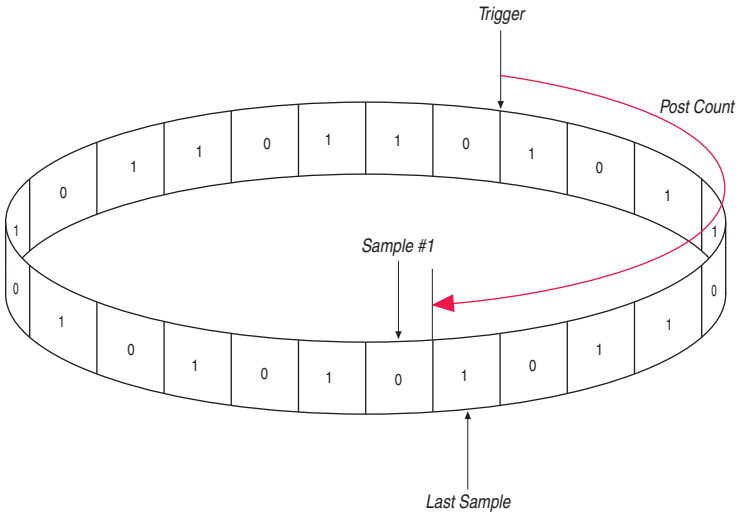
Actions to the acquisition buffer can accept an optional post-count argument. This post-count argument enables you to define a custom triggering position for each segment in the acquisition buffer. Example 17–11 shows an example that applies a trigger position to all segments in the acquisition buffer. 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 17–11.

```
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 17–44 illustrates the triggering position.

Figure 17–44. 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 17–12 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 17–12.

```
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 are replaced with new technologies that maximize productivity. The SignalTap II Logic Analyzer gives you the same benefits as a traditional logic analyzer, without the many shortcomings of a piece of dedicated test equipment. The SignalTap II Logic Analyzer provides many new and innovative features that allow you to capture and analyze internal signals in your FPGA, allowing you to quickly debug your design.

Document Revision History

Table 17–14 shows the revision history for this chapter.

Table 17–14. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Add new acquisition buffer content to the "View, Analyze, and Use Captured Data" section. ■ Added script sample for generating hexadecimal CRC values in programmed devices. ■ Created cross references to Quartus II Help for duplicated procedural content.
November 2009	9.1.0	No change to content.
March 2009	9.0.0	■ Updated Table 17–1 ■ Updated "Using Incremental Compilation with the SignalTap II Logic Analyzer" on page 17–45 ■ Added new Figure 17–34 ■ Made minor editorial updates
November 2008	8.1.0	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
May 2008	8.0.0	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

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

Take an online survey to provide feedback about this handbook chapter.

18. In-System Debugging Using External Logic Analyzers

QII53016-10.0.1

The Quartus II Logic Analyzer Interface (LAI) allows you to use an external logic analyzer and a minimal number of FPGA I/O pins to examine the behavior of internal signals while your design is running at full speed on your FPGA.

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 and output pins, the Quartus II LAI enables you to 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 LAI" on page 18–4
- "Working with LAI Files" on page 18–4
- "Controlling the Active Bank During Runtime" on page 18–6
- "Using the LAI with Incremental Compilation" on page 18–7

The term "logic analyzer" when used in this chapter includes both logic analyzers and oscilloscopes equipped with digital channels, commonly referred to as mixed signal analyzers or MSOs.

Refer to *Devices and Adapters* in Quartus II Help for a list of Altera supported devices.

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 SignalTap® II Logic Analyzer
- An external logic analyzer, which connects to internal signals in your FPGA by using the Quartus II LAI

Table 18–1 describes the advantages of each debugging tool.

Table 18–1. Comparing the SignalTap II Logic Analyzer with the Logic Analyzer Interface

Feature and Description	Logic Analyzer Interface	SignalTap II Logic Analyzer
Sample Depth You have access to a wider sample depth with an external logic analyzer. In the SignalTap II 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 the SignalTap II Logic Analyzer with your design. An external logic analyzer adds minimal logic, which removes resource limits on place-and-route.	✓	—
Triggering Capability The SignalTap II Logic Analyzer offers triggering capabilities that are comparable to 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*.

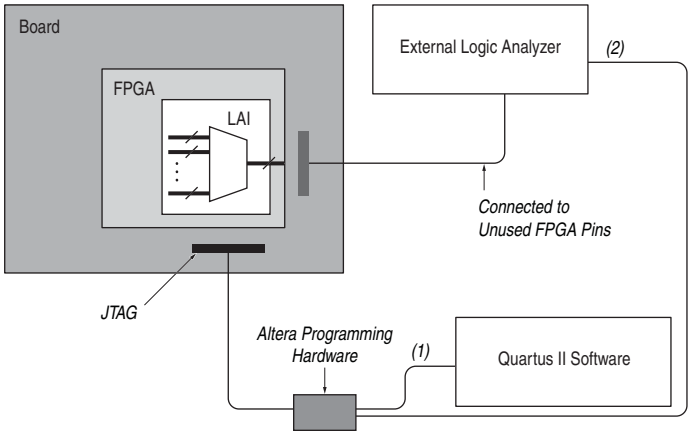
Required Components

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 18–1 shows the LAI and the hardware setup.

Figure 18–1. LAI and Hardware Setup



Notes to Figure 18–1:

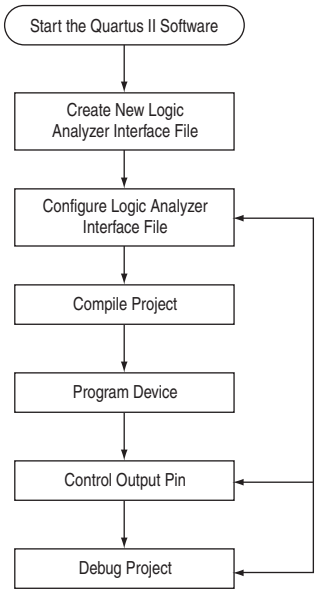
(1) Configuration and control of the LAI using a computer loaded with the Quartus II software via the JTAG port.

(2) Configuration and control of the LAI using a third-party vendor logic analyzer via the JTAG port. Support varies by vendor.

Debugging Your Design Using the LAI

Figure 18–2 shows the steps you must follow to debug your design with the Quartus II LAI.

Figure 18–2. LAI and Hardware Setup



Notes to Figure 18–1:

(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.

Working with LAI Files

The **.lai** file stores the configuration of an LAI instance. The **.lai** file opens in the LAI editor. The editor allows you to group multiple internal signals to a set of external pins. The configuration parameters are described in the following sections.

To create a new **.lai** file or open an existing **.lai** file, refer to *Setting Up the Logic Analyzer Interface* in Quartus II Help.

Configuring the File Core Parameters

After you create the **.lai** file, you must configure the **.lai** file core parameters by clicking on the **Setup View** list, and then selecting **Core Parameters**. Table 18–2 lists the **.lai** file core parameters.

Table 18–2. LAI 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 LAI File Pins to Available I/O Pins

To configure the **.lai** file I/O pin parameters, select **Pins** in the **Setup View** list. To assign pin locations for the LAI, double-click the **Location** column next to the reserved pins in the **Name** column, and the Pin Planner opens.

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 LAI 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**.

To view all of your bank connections, click **Setup View** and select **All Banks**.

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. 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.

Compiling Your Quartus II Project

When you save your **.lai** file, a dialog box prompts you to enable the LAI instance for the active project. Alternatively, you can specify the **.lai** file your project uses in the **Global Project Settings** dialog box.

After you specify 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 18–3).

Figure 18–3. 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 LAI

After compilation completes, you must configure your FPGA before using the LAI.

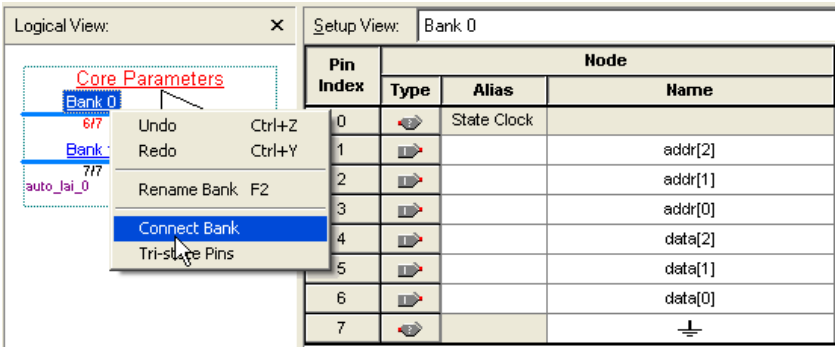
You can use the LAI with multiple devices in your JTAG chain. Your JTAG chain can also consist of devices that do not support the LAI or non-Altera, JTAG-compliant devices. To use the LAI in more than one FPGA, create an **.lai** file and configure an **.lai** file for each FPGA that you want to analyze.

To configure a device or a set of devices for use with LAI, refer to *Enabling the Logic Analyzer Interface* in Quartus II Help.

Controlling the Active Bank During Runtime

When you have programmed your FPGA, you can control which bank you map to the reserved **.lai** file output pins. To control which bank you map, in the schematic in the logical view, right-click the bank and click **Connect Bank** (Figure 18–4).

Figure 18–4. 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 about how to establish connections between debugging headers and logic analyzers, refer to the documentation for your logic analyzer.

Using the LAI with Incremental Compilation

The Incremental Compilation feature in the Quartus II software allows you to preserve the synthesis and fitting results of your design. This is an effective feature for reducing compilation times if you only modify a portion of a design or you wish to preserve the optimization results from a previous compilation.

The Incremental Compilation feature is well suited for use with LAI since LAI comprises a small portion of most designs. Because LAI consists of only a small portion of your design, incremental compilation helps to minimize your compilation time. Incremental compilation works best when you are only changing a small portion of your design. Incremental compilation yields an accurate representation of your design behavior when changing the **.lai** file through multiple compilations.

For further details on how to use Incremental Compilation with the LAI, refer to *Enabling the Logic Analyzer Interface* in Quartus II Help.

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 deliver your product in the shortest amount of time.

Document Revision History

Table 18–3 shows the revision history for this chapter.

Table 18–3. Document Revision History

Date	Version	Changes Made
August 2010	10.0.1	Corrected links
July 2010	10.0.0	■ Created links to the Quartus II Help ■ Editorial updates ■ Removed Referenced Documents section
November 2009	9.1.0	■ Removed references to APEX devices ■ Editorial updates
March 2009	9.0.0	■ Minor editorial updates ■ Removed Figures 15–4, 15–5, and 15–11 from 8.1 version
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content
May 2008	8.0.0	■ 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

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

Take an *online survey* to provide feedback about this handbook chapter.

19. In-System Modification of Memory and Constants

QII53012-10.0.1

This chapter explains how to use the Quartus II® In-System Memory Content Editor as part of your FPGA design and verification flow.

The In-System Memory Content Editor allows you to view and update memories and constants with the JTAG port connection.

The In-System Memory Content Editor allows access to dense and complex FPGA designs. When you program devices, you have read and write access to the memories and constants through the JTAG interface. You can then identify, test, and resolve issues with your design by testing changes to memory contents in the FPGA while your design is running.

Overview

This chapter contains the following sections:

- "Device Megafunction Support" on page 19–2
- "Updating Memory and Constants in Your Design" on page 19–2
- "Creating In-System Modifiable Memories and Constants" on page 19–3
- "Running the In-System Memory Content Editor" on page 19–3

When you use the In-System Memory Content Editor in conjunction with the SignalTap II Logic Analyzer, you can more easily view and debug your design in the hardware lab.

For more information about the SignalTap II Logic Analyzer, refer to the *Design Debugging Using the SignalTap II 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. The write capability allows 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 Memory 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 the error handling functionality of your design.

The Quartus II software offers a variety 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 Design 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 19–1 lists the types of memory supported by the MegaWizard™ Plug-In Manager and the In-System Memory Content Editor.

Table 19–1. MegaWizard Plug-In Manager Support

Installed Plug-Ins Category	Megafunction
Gates	LPM_CONSTANT
Memory Compiler	RAM: 1-PORT, ROM: 1-PORT
Storage	ALTSYNCRAM, LPM_RAM_DQ, LPM_ROM

Table 19–2 lists support for in-system updating of memory and constants for the Stratix® series, Arria® GX, and Cyclone® series device families.

Table 19–2. Supported Megafunctions

Megafunction	Arria GX / Stratix Series			Cyclone Series
	M512 Blocks	M4K Blocks	MegaRAM Blocks	
LPM_CONSTANT	Read/ Write	Read/ Write	Read/ Write	Read/ Write
LPM_ROM	Write	Read/ Write	N/A	Read/ Write
LPM_RAM_DQ	N/A	Read/ Write	Read/ Write	Read/ Write
ALTSYNCRAM (ROM)	Write	Read/ Write	N/A	Read/ Write
ALTSYNCRAM (Single-Port RAM Mode) (1)	N/A	Read/ Write	Read/ Write	Read/ Write

Note to Table 19–2:

(1) Only write-only mode is applicable for this single-port RAM. In read-only mode, use LPM_ROM instead of LPM_RAM_DQ.

Updating Memory and Constants in Your Design

To use the In-System Updating of Memory and Constants feature, perform 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 a 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 19–1.

To enable your memory or constant to be modifiable, refer to *Setting up the In-System Memory Content Editor* in Quartus II Help.

If you instantiate a memory or constant megafunction directly with 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 has three separate panes: the **Instance Manager**, the **JTAG Chain Configuration**, and the **Hex Editor**.

The **Instance Manager** pane displays all available run-time modifiable memories and constants in your FPGA device. The **JTAG Chain Configuration** pane 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** pane.

If you have more than one device with 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. Each In-System Memory Content Editor can access the in-system memories and constants in a single device.

Instance Manager

When you scan the JTAG chain to update the **Instance Manager** pane, you can view a list of all run-time modifiable memories and constants in the design. The **Instance Manager** pane 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 with the **Instance Manager** pane. For more information refer to *Instance Manager Pane* in Quartus II Help.

In addition to the buttons available in the **Instance Manager** pane, you can read and write data by selecting commands from the Processing menu, or the right-click menu in the **Instance Manager** pane or **Hex Editor** pane.

The status of each instance is also displayed beside each entry in the **Instance Manager** pane. The status indicates if the instance is **Not running**, **Offloading data**, or **Updating data**. The health monitor provides 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 number with the corresponding instance ID.

Editing Data Displayed in the Hex Editor Pane

You can edit data read from your in-system memories and constants displayed in the **Hex Editor** pane by typing values directly into the editor or by importing memory files.

For more information, refer to *Working with In-System Memory Content Editor Data* in Quartus II Help.

Importing and Exporting Memory Files

The In-System Memory Content Editor allows you to import and export data values for memories that have the In-System Updating feature enabled. Importing from a data file 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.

For more information, refer to *Working with In-System Memory Content Editor Data* in Quartus II Help.

Viewing Memories and Constants in the Hex Editor Pane

For each instance of an in-system memory or constant, the **Hex Editor** pane 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** pane displays data in columns of words that contain columns of bytes.

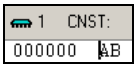
Unprintable ASCII characters are represented by a period (.). The color of the data changes as you perform reads and writes. Data displayed in black indicates the data in the **Hex Editor** pane was the same as the data read from the device. If the data in the **Hex Editor** pane changes color to red, the data previously shown in the **Hex Editor** pane 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.

The bit offset is the bit position of the cursor within the word. In the following example, a word is set to 8 bits.

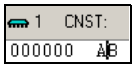
With the cursor in the position shown in Figure 19–1, the word location is 0x0000 and the bit position is 0x0007.

Figure 19–1. Hex Editor Cursor Positioned at Bit 0×0007



With the cursor in the position shown in Figure 19–2, the word location remains 0x0000, but the bit position is 0x0003.

Figure 19–2. 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 ↵
```

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook* and *API Functions for Tcl* in Quartus II Help. 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 of the command options and scripting examples, refer to the Tcl API Help Browser and the *API Functions for Tcl* in Quartus II Help.

Programming the Device with 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, refer to *Setting up the In-System Memory Content Editor* in Quartus II Help.

Example: Using the In-System Memory Content Editor with the SignalTap II Logic Analyzer

The following scenario describes how you can use the In-System Updating of Memory and Constants feature with the SignalTap II Logic Analyzer to efficiently debug your design in-system. You can use the In-System Memory Content Editor and the SignalTap II Logic Analyzer simultaneously with the JTAG interface.

Scenario: 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 Logic Analyzer.
2. Using the SignalTap II Logic Analyzer to tap and trigger on internal design nodes, you find the FIR filter to be functioning outside of the expected cutoff 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 with the In-System Memory Content Editor.

In this scenario, you can quickly locate the source of the problem using both the In-System Memory Content Editor and the SignalTap II Logic Analyzer. You can also verify the functionality of your device by changing the coefficient values before modifying the design source files.

You can also 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 to a device for efficient debugging in a hardware lab. You can use the In-System Memory and Content Editor with the SignalTap II Logic Analyzer to maximize the visibility into an Altera FPGA. By maximizing visibility and access to internal logic of the device, you can identify and resolve problems with your design more easily.

Document Revision History

Table 19–3 shows the revision history of this chapter.

Table 19–3. Document Revision History

Date	Version	Changes Made
August 2010	10.0.1	Corrected links
July 2010	10.0.0	■ Inserted links to Quartus II Help ■ Removed Reference Documents section
November 2009	9.1.0	■ Delete references to APEX devices ■ Style changes
March 2009	9.0.0	No change to content
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content
May 2008	8.0.0	■ 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

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

Take an online survey to provide feedback about this handbook chapter.

20. Design Debugging Using In-System Sources and Probes

QII53021-10.0.0

This chapter provides detailed instructions about how to use the In-System Sources and Probes Editor and Tcl scripting in the Quartus® II software to debug your design.

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 design. You can make the debugging cycle more efficient when you can drive any internal signal manually within your design, which allows you to perform the following actions:

- Force the occurrence of trigger conditions set up 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

The In-System Sources and Probes Editor in the Quartus II software extends the portfolio of verification tools, and allows you to easily control any internal signal and provides you with a completely dynamic debugging environment. Coupled with either the SignalTap II Logic Analyzer or SignalProbe, the In-System Sources and Probes Editor 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 variety 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*.

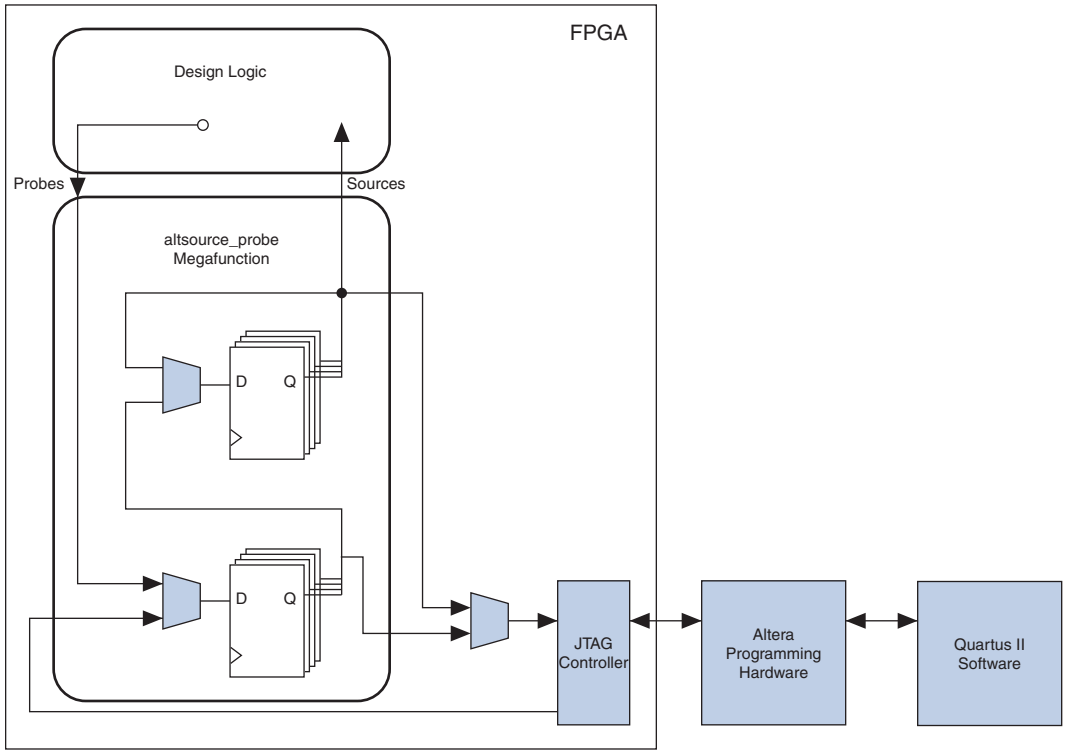
Overview

This chapter includes the following topics:

- "Design Flow Using the In-System Sources and Probes Editor" on page 20–3
- "Running the In-System Sources and Probes Editor" on page 20–6
- "Tcl interface for the In-System Sources and Probes Editor" on page 20–9
- "Design Example: Dynamic PLL Reconfiguration" on page 20–12

The In-System Sources and Probes Editor 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. When you compile your design, the ALTSOURCE_PROBE megafunction sets up a register chain to either drive or sample the selected nodes in your logic design. During run time, the In-System Sources and Probes Editor uses a JTAG connection to shift data to and from the ALTSOURCE_PROBE megafunction instances. Figure 20–1 shows a block diagram of the components that make up the In-System Sources and Probes Editor.

Figure 20–1. In-System Sources and Probes Editor Block Diagram



The ALTSOURCE_PROBE megafunction hides the detailed transactions between the JTAG controller and the registers instrumented in your design to give you a basic building block for stimulating and probing your design. Additionally, the In-System Sources and Probes Editor provides single-cycle samples and single-cycle writes to selected logic nodes. You can use this feature to input simple virtual stimuli and to capture the current value on instrumented nodes. Because the In-System Sources and Probes Editor gives you access to logic nodes in your design, you can toggle the inputs of low-level components during the debugging process. 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.

The In-System Sources and Probes Editor allows you to easily implement control signals in your design as virtual stimuli. This feature can be especially helpful for prototyping your design, such as in the following operations:

- Creating virtual push buttons
- Creating a virtual front panel to interface with your design
- Emulating external sensor data
- Monitoring and changing run time constants on the fly

The In-System Sources and Probes Editor supports Tcl commands that interface with all your ALTSOURCE_PROBE megafunction instances to increase the level of automation.

Hardware and Software Requirements

The following components are required to use the In-System Sources and Probes Editor:

- Quartus II software

or

- Quartus II Web Edition (with the TalkBack feature turned on)
- Download Cable (USB-Blaster™ download cable or ByteBlaster™ cable)
- Altera® development kit or user design board with a JTAG connection to device under test

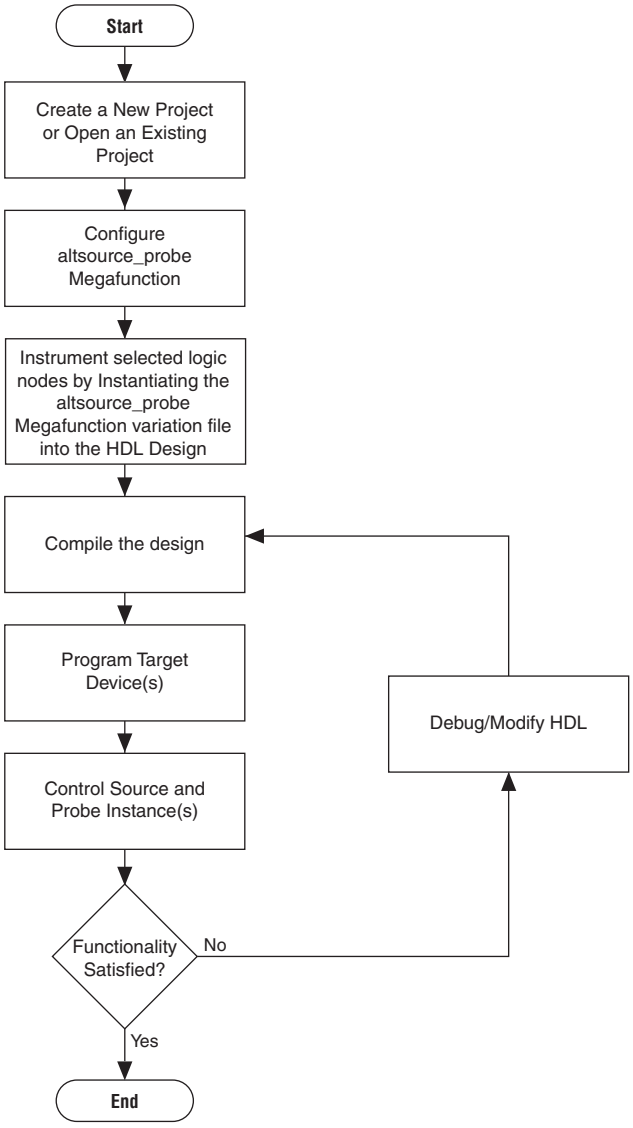
The In-System Sources and Probes Editor supports the following device families:

- Arria® GX
- Stratix® series
- HardCopy® II
- Cyclone® series
- MAX® II

Design Flow Using the In-System Sources and Probes Editor

The In-System Sources and Probes Editor supports an RTL flow. Signals that you want to view in the In-System Sources and Probes editor are connected to an instance of the ALTSOURCE_PROBE megafunction. After you compile the design, you can control each ALTSOURCE_PROBE instance via the **In-System Sources and Probes Editor** pane or via a Tcl interface. The complete design flow is shown in Figure 20–2.

Figure 20–2. FPGA Design Flow Using the In-System Sources and Probes Editor



Configuring the ALTSOURCE_PROBE Megafunction

To use the In-System Sources and Probes Editor in your design, you must first instantiate the ALTSOURCE_PROBE megafunction variation file. You can configure the ALTSOURCE_PROBE megafunction with the MegaWizard™ Plug-In Manager. Each source or probe port can be up to 256 bits. You can have up to 128 instances of the ALTSOURCE_PROBE megafunction in your design.

To configure the ALTSOURCE_PROBE megafunction, performing the following steps:

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 of the MegaWizard Plug-In Manager, make the following selections:
   a. In the **Installed Plug-Ins** list, expand the **JTAG-accessible Extensions** folder and select **In-System Sources and Probes**.

   Verify that the currently selected device family matches the device you are targeting.

   b. Select an output file type and enter the 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 of the MegaWizard Plug-In Manager, make the following selections:
   a. Under **Do you want to specify an Instance Index?**, turn on **Yes**.
   b. Specify the **'Instance ID'** of this instance.
   c. Specify the width of the probe port. The width can be from 0 bit to 256 bits.
   d. Specify the width of the source port. The width can be from 0 bit to 256 bits.
7. On page 3 of the MegaWizard Plug-In Manager, you can click **Advanced Options** and specify other options, including the following:
   - **What is the initial value of the source port, in hexadecimal?**—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**—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 turned on, creates a clock enable input for the synchronization registers. You can turn on this option only when the **Write data to the source port synchronously to the source clock** option is turned on.

The In-System Sources and Probes Editor does not support simulation. You must remove the ALTSOURCE_PROBE megafunction instantiation before you create a simulation netlist.

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 20–1.

Table 20–1. ALTSOURCE_PROBE Megafunction Port Information

Port Name	Required?	Direction	Comments
probe[]	No	Input	The outputs from your design.
source_clk	No	Input	Source Data is written synchronously to this clock. This input is required if you turn on **Source Clock** 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, if your device has available resources. Each instance of the ALTSOURCE_PROBE megafunction uses a pair of registers per signal for the width of the widest port in the megafunction. Additionally, there is some fixed overhead logic to accommodate communication between the ALTSOURCE_PROBE instances and the JTAG controller. You can also specify an additional pair of registers per source port for synchronization.

Compiling the Design

When you compile your design with the In-System Sources and Probes megafunction instantiated, an instance of the ALTSOURCE_PROBE and SLD_HUB instances are added to your compilation hierarchy automatically. These instances provide communication between the JTAG controller and your instrumented logic.

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 instance in the Project Navigator. You can then modify the connections in the HDL source file. You must recompile your design after you make changes.

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 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*.

Running the In-System Sources and Probes Editor

The In-System Sources and Probes Editor gives you control over all ALTSOURCE_PROBE megafunction instances within your design. The editor allows you to view all available run time controllable instances of the ALTSOURCE_PROBE megafunction in your design, provides a push-button interface to drive all your source nodes, and provides a logging feature to store your probe and source data.

To run the In-System Sources and Probes Editor, on the **Tools** menu, click **In-System Sources and Probes Editor.**

The In-System Sources and Probes Editor contains 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 data that the In-System Sources and Probes Editor acquires.
- **In-System Sources and Probes Editor**—Logs all data read from the selected instance and allows you to modify source data that is written to your device.

When you use the In-System Sources and Probes Editor, you do not need to open a Quartus II software 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 device selected in the **JTAG Chain Configuration** pane. You can also use a previously saved configuration to run the In-System Sources and Probes Editor.

Each **In-System Sources and Probes Editor** pane 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** panes to access the megafunction instances in each device.

Programming Your Device With JTAG Chain Configuration

After you compile your project, you must configure your FPGA before you use the In-System Sources and Probes Editor. To configure a device to use with the In-System Sources and Probes Editor, perform the following steps:

1. Open the In-System Sources and Probes Editor.
2. In the **JTAG Chain Configuration** pane, point to **Hardware**, and then select the 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 (the device may be automatically detected). You may need to click **Scan Chain** to detect your target device.
4. In the **JTAG Chain Configuration** pane, click to browse for the SRAM Object File (**.sof**) that includes the In-System Sources and Probes instance or instances. (The **.sof** may be automatically detected).
5. Click **Program Device** to program the target device.

Instance Manager

The **Instance Manager** pane 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 following buttons and sub-panes are provided in the **Instance Manager** pane:

- **Read Probe Data**—Samples the probe data in the selected instance and displays the probe data in the **In-System Sources and Probes Editor** pane.
- **Continuously Read Probe Data**—Continuously samples the probe data of the selected instance and displays the probe data in the **In-System Sources and Probes Editor** pane; you can modify the sample rate via the **Probe read interval** setting.
- **Stop Continuously Reading Probe Data**—Cancels continuous sampling of the probe of the selected instance.
- **Write Source Data**—Writes data to all source nodes of the selected instance.
- **Probe Read Interval**—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**—Controls the event log in the **In-System Sources and Probes Editor** pane.
- **Write Source Data**—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** pane. 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 information about the sources and probes instances in your design.

In-System Sources and Probes Editor Pane

The **In-System Sources and Probes Editor** pane allows you to view data from all sources and probes in your design. The data is organized according to the index number of the instance. The editor provides an easy way to manage your signals, and allows you to rename signals or group them into buses. All data collected from in-system source and probe nodes is recorded in the event log and you can view the data as a timing diagram.

Reading Probe Data

You can read data by selecting the ALTSOURCE_PROBE instance in the **Instance Manager** pane and clicking **Read Probe Data**. This action produces a single sample of the probe data and updates the data column of the selected index in the **In-System Sources and Probes Editor** pane. You can save the data to an event log by turning on the **Save data to event log** option in the **Instance Manager** pane.

If you want to sample data from your probe instance continuously, in the **Instance Manager** pane, click the instance you want to read, and then click **Continuously read probe data**. While reading, the status of the active instance shows **Unloading**. You can read continuously from multiple instances.

You can access read data with the shortcut menus in the **Instance Manager** pane.

To adjust the probe read interval, in the **Instance Manager** pane, turn on the **Manual** option in the **Probe read interval** sub-pane, and specify the 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. You can adjust the event log window buffer size in the **Maximum Size** box.

Writing Data

To modify the source data you want to write into the ALTSOURCE_PROBE instance, click the name field of the signal you want to change. For buses of signals, you can double-click the data field and type the value you want to drive 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 are not written out to the ALTSOURCE_PROBE instances appear in red. To update the ALTSOURCE_PROBE instance, highlight the instance in the **Instance Manager** pane and click **Write source data**. The **Write source data** function is also available via the shortcut menus in the **Instance Manager** pane.

The In-System Sources and Probes Editor provides the option to continuously update each ALTSOURCE_PROBE instance. Continuous updating allows any modifications you make to the source data buffer to also write immediately to the ALTSOURCE_PROBE instances. To continuously update the ALTSOURCE_PROBE instances, change the **Write source data** field from **Manually** to **Continuously**.

Organizing Data

The **In-System Sources and Probes Editor** pane allows you to group signals into buses, and also 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 shortcut menus.

The **In-System Sources and Probes Editor** pane allows you to rename any signal. To rename a signal, double-click the name of the signal and type 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 active instance as you move your pointer over the data samples.

You can save the changes that you make and the recorded data to a Sources and Probes File (**.spf**). To save changes, on the File menu, click **Save**. The file contains all the modifications you made to the signal groups, as well as the current data event log.

Tcl interface for the In-System Sources and Probes Editor

To support automation, the In-System Sources and Probes Editor supports the procedures described in this chapter in the form of Tcl commands. The Tcl package for the In-System Sources and Probes Editor is included by default when you run `quartus_stp`.

The Tcl interface for the In-System Sources and Probes Editor provides a powerful platform to help you debug your design. The Tcl interface is especially helpful for debugging designs that require toggling multiple sets of control inputs. You can combine multiple commands with a Tcl script to define a 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 Manual*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

Table 20–2 shows the Tcl commands you can use instead of the In-System Sources and Probes Editor.

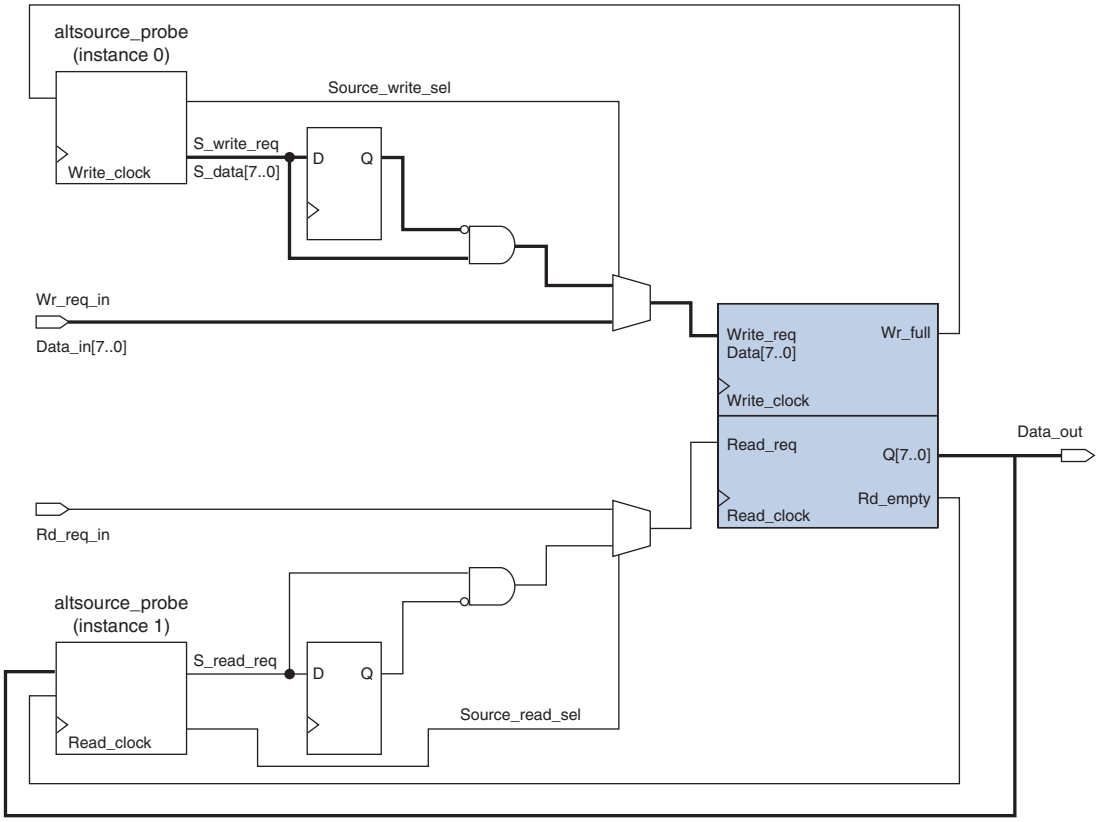
Table 20–2. In-System Sources and Probes Tcl Commands

Command	Argument	Description
`start_insystem_source_probe`	`-device_name <device name>` `-hardware_name <hardware name>`	Opens a handle to a device with 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 is 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 that specifies 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 that specifies the status of each source, with the MSB as the left-most bit.
write_source_data	`-instance_index <instance_index>` `-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 20–1 shows an excerpt from a Tcl script with procedures that control the ALTSOURCE_PROBE instances of the design as shown in Figure 20–3. 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 to 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 20–1, 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.

Use the Tcl script in debugging situations 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 20–3. A DCFIFO Example Design Controlled by the Tcl Script in Example 20–1



Example 20–1. Tcl Script Procedures for Reading and Writing to the DCFIFO in Figure 20–3 (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 20–1. Tcl Script Procedures for Reading and Writing to the DCFIFO in Figure 20–3 (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
}
```

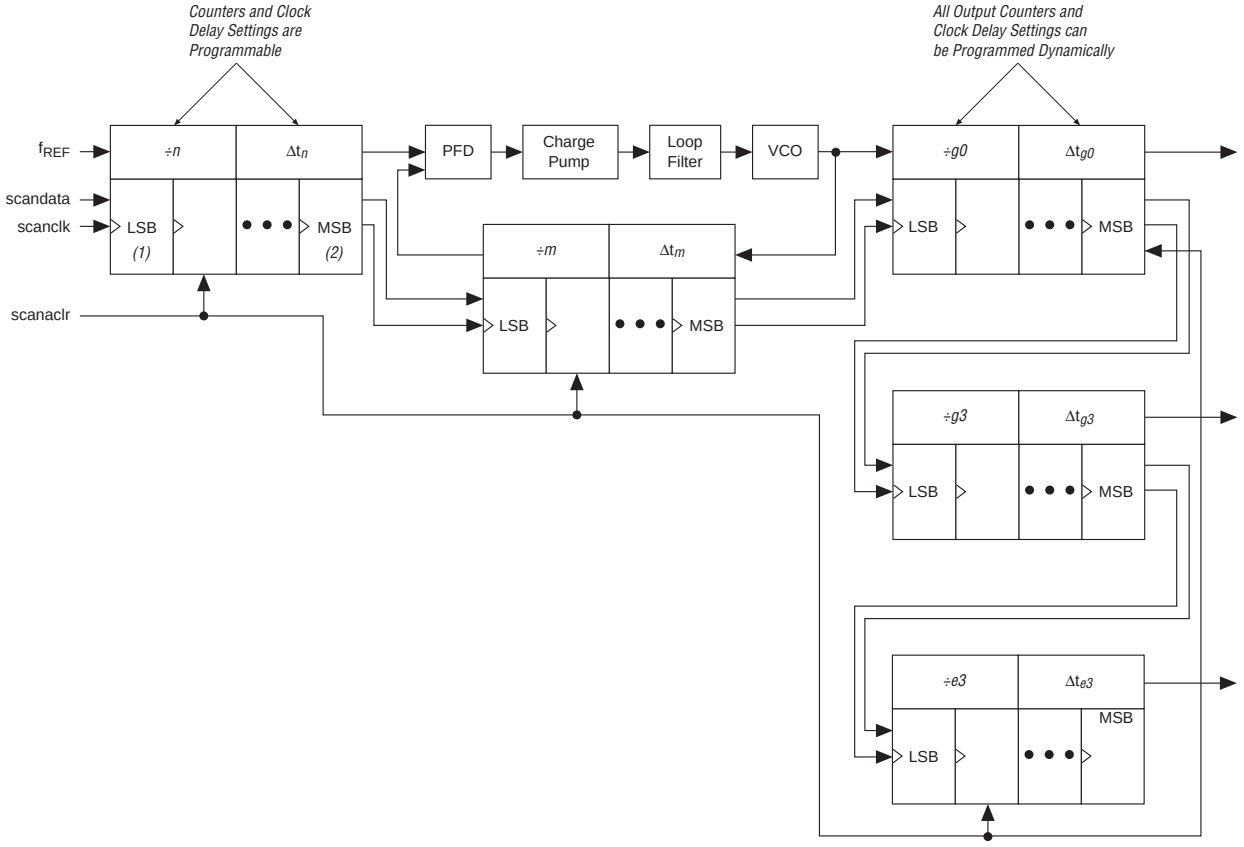
Design Example: Dynamic PLL Reconfiguration

The In-System Sources and Probes Editor can help you create a virtual front panel during the prototyping phase of your design. You can create relatively simple, high functioning designs of in a short amount of time. The following PLL reconfiguration example demonstrates how to use the In-System Sources and Probes Editor to provide a GUI to dynamically reconfigure a Stratix PLL.

Stratix PLLs allow 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 that contains all modifiable PLL parameters. After you update all the PLL parameters in the cache, the ALTPLL_RECONFIG megafunction drives the PLL register chain to update the PLL with the updated parameters. Figure 20–4 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 20–4. 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 reconfiguration signal on the ALTPLL_RECONFIG megafunction. The reconfiguration 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 20–5. The Tk GUI is shown in Figure 20–6.

Figure 20–5. Block Diagram of Dynamic PLL Reconfiguration Design Example

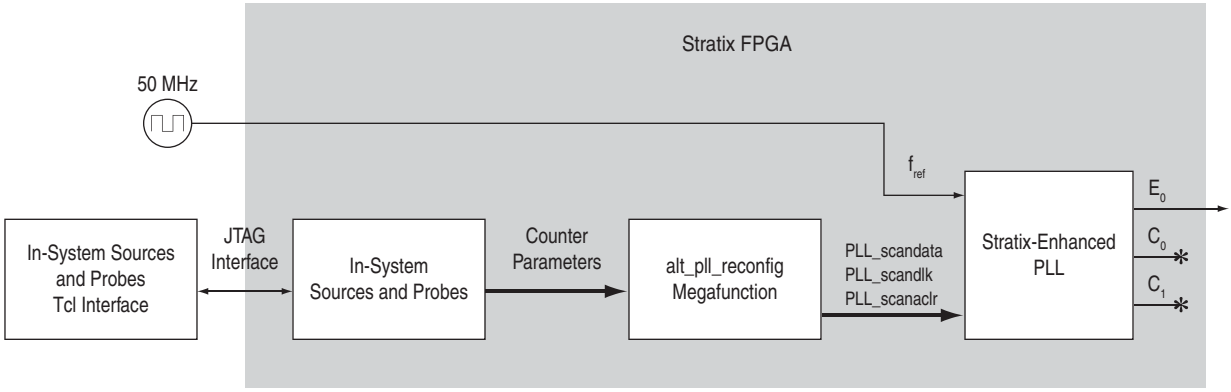
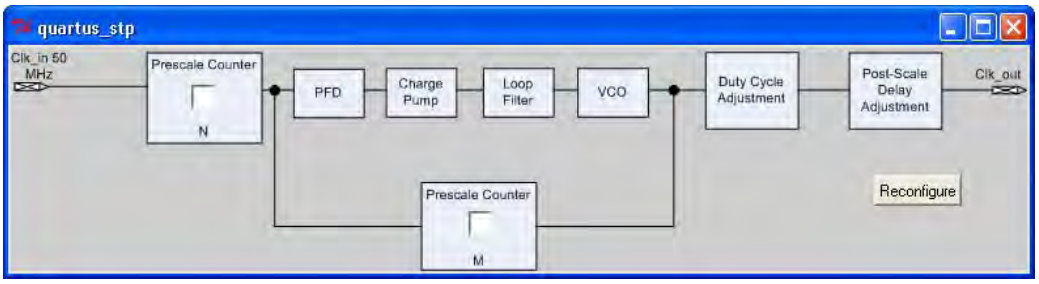


Figure 20–6. Interactive PLL Reconfiguration GUI Created with Tk and In-System Sources and Probes Tcl Package



This design example was created using a Nios® II Development Kit, Stratix Edition. The file **sourceprobe_DE_dynamic_pll.zip** contains all the necessary files for running this design example, including the following:

- **Readme.txt**—A text file that describes the files contained in the design example and provides instructions about running the Tk GUI shown in Figure 20–6.
- **Interactive_Reconfig.qar**—The archived Quartus II project for this design example.

Download the **sourceprobe_DE_dynamic_pll.zip** file from the Literature: Quartus II Handbook page of the Altera website.

Conclusion

The In-System Sources and Probes Editor provides stimuli and receives responses from the target design during run time. With the simple and intuitive interface, you can add virtual inputs to your design during run time without using external equipment. When used in conjunction with the SignalTap II Logic Analyzer, you can use the In-System Sources and Probes Editor to obtain greater control of the signals in your design, and thus help shorten the verification cycle.

Document Revision History

Table 20–3 shows the revision history for this chapter.

Table 20–3. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	Minor corrections.
November 2009	9.1.0	■ Removed references to obsolete devices. ■ Style changes.
March 2009	9.0.0	No change to content.
November 2008	8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	8.0.0	■ 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

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

Take an online survey to provide feedback about this handbook chapter.

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 21, Cadence Encounter Conformal Support

21. Cadence Encounter Conformal Support

QII53011-10.0.0

The Quartus® II software provides formal verification support for Altera® designs through interfaces with a formal verification EDA tool, the Cadence Encounter Conformal Logic Equivalence Check (LEC) software.

The two types of formal verification are equivalence checking and model checking. This chapter discusses equivalence checking with the Conformal LEC software.

Use the Conformal LEC 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 Conformal LEC 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, Cyclone®, Cyclone II, Cyclone III, HardCopy® II, Stratix®, Stratix II, Stratix II GX, Stratix III, and Stratix IV device families.

This chapter contains the following sections:

- "Formal Verification Design Flow" on page 21–2
- "RTL Coding Guidelines for Quartus II Integrated Synthesis" on page 21–4
- "Black Boxes in the Conformal LEC Flow" on page 21–8
- "Generating the Post-Fit Netlist Output File and the Conformal LEC Setup Files" on page 21–10
- "Understanding the Formal Verification Scripts for Conformal LEC" on page 21–12
- "Comparing Designs Using Conformal LEC" on page 21–15
- "Known Issues and Limitations" on page 21–16
- "Black Box Models" on page 21–18
- "Conformal Dofile/Script Example" on page 21–19

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 can 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 for 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 foregoing 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 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 21–16 before you consider using the formal verification flow in your design methodology.

Formal Verification Design Flow

Altera supports formal verification using the Conformal LEC software for the following two synthesis tools:

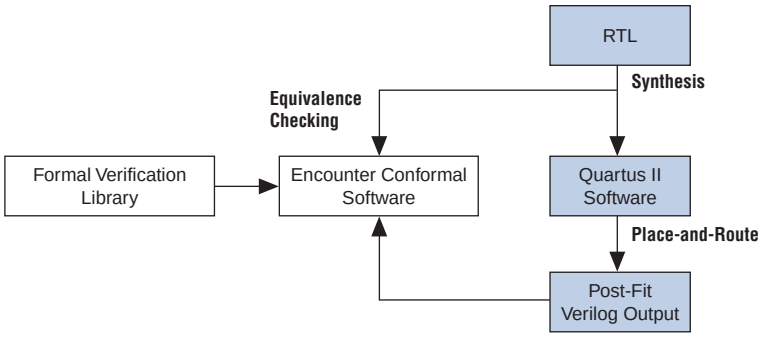
- "Quartus II Integrated Synthesis"
- "Synplify Pro" on page 21–3

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 21–1. 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 21–1. Formal Verification Using Quartus II Integrated Synthesis and the Conformal LEC Software



EDA Tool Support for Quartus II Integrated Synthesis

The formal verification flow using the Quartus II software and Conformal LEC software supports the following software versions and operating systems:

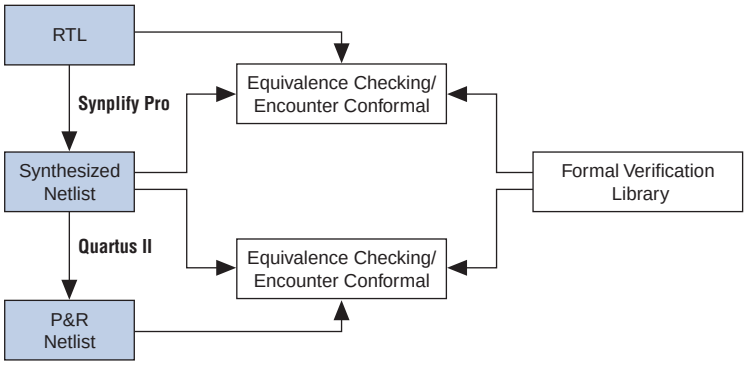
- Quartus II software beginning with version 4.2
- Cadence Conformal LEC software beginning with version 4.3.5A
- Solaris and Linux operating systems

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 21–2.

For additional information about performing equivalency checking between RTL and post-synthesis netlists generated from Synplify Pro software, refer to the Synplify Pro documentation.

Figure 21–2. Formal Verification Flow Using Synplify Pro and the Conformal LEC Software



EDA Tool Support for Synplify Pro

Formal verification flow using the Quartus II software, the Synopsys Synplify Pro, and the Conformal LEC software supports the software versions and operating systems shown in Table 21–1.

Table 21–1. Compatible Software Versions (Part 1 of 2)

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

Table 21–1. Compatible Software Versions (Part 2 of 2)

Quartus II Software Version	Cadence Conformal LEC Version	Synplify Pro Version
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
9.1	8.1	C2009.06
10.0	8.1	C2009.06

RTL Coding Guidelines for Quartus II Integrated Synthesis

The Conformal LEC software can compare the RTL code against the post-fit netlist generated by the Quartus II software. The Conformal LEC 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 Conformal LEC 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 the Conformal LEC software are also supported in the formal verification flow. The Quartus II integrated synthesis and Conformal LEC 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 Conformal LEC software. Therefore, it is important to use caution with unsupported pragmas because they can lead to verification mismatches.

For example, you can use Quartus II integrated synthesis to synthesize RTL code with the synthesis directive `read_comments_as_HDL` (Example 21–1 and Example 21–2).

Example 21–1. 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 21–2. 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 Conformal LEC software does not support the synthesis directive `read_comments_as_HDL`, and the directive has no affect on the Conformal LEC software.

Table 21–2 lists supported pragmas and trigger keywords for formal verification.

Table 21–2. 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 21–2:

(1) Do not use Verilog 2001-style pragma declarations. The Quartus II software and the Conformal LEC 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 the Conformal LEC software does not find the same optimizations, it can lead to unmapped points in the golden netlist. Example 21–3 illustrates the issue.

Example 21–3. 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 21–4.

Example 21–4. 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 Conformal LEC 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 Conformal LEC software. For details about setting black box properties on a particular module, refer to Table 21–3 on page 21–9.

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 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 Conformal LEC software infers a latch primitive in the Conformal LEC 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 about issues with combinational loops, refer to "Known Issues and Limitations" on page 21–16.

Finite State Machine Coding Styles

When a state machine is inferred by the Conformal LEC 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** setting 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* when writing finite state machines (FSMs). This allows the Quartus II integrated synthesis and the Conformal LEC software to infer a similar state machine for the same RTL code.

Black Boxes in the Conformal LEC 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, which consists of the modules listed above and its implemented version, the modules must be treated as black boxes by the Conformal LEC 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 with their required user actions described in Table 21–3 on page 21–9.

Verilog Output Files (**.vo**) written by the Quartus II software contain the black box hierarchy when you make an EDA Formal Verification Hierarchy assignment with the value `BLACKBOX`.

If this assignment is 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 Conformal LEC software.

Table 21–3. Black Boxes and Required User Action

Type of Black Box	Required User Action
Altera library of parameterized modules (LPMs) and megafunctions.	No action required. The Quartus II software automatically creates a black box list of components and preserves the hierarchy.
Any parameterized entity other than those listed in the *Guidelines for Creating a Design for Use with the Encounter Conformal and Quartus II Software* topic in Quartus II Help.	User must designate the wrapper that instantiates the parameterized entity as a black box.
Non-parameterized entities that the user wants to designate as a black box.	User can designate the entity itself as a black box.

You can also use Tcl commands or the 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 command shown in Example 21–5 to preserve the boundary interface of a black box entity: `dram`.

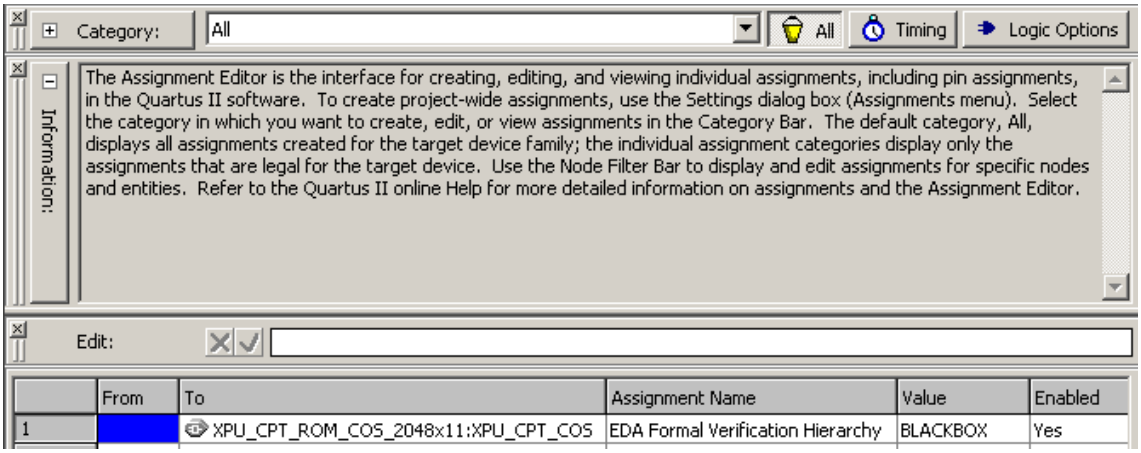
Example 21–5. Tcl Command to Create a Black Box

```
set_instance_assignment -name EDA_FV_HIERARCHY BLACKBOX -to | -entity dram
```

GUI

To preserve the boundary interface of an entity using the GUI, make an EDA Formal Verification Hierarchy assignment to the entity with the value BLACKBOX as shown in Figure 21–3.

Figure 21–3. Setting the Black-Box Property on a Module



Generating the Post-Fit Netlist Output File and the Conformal LEC 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 Conformal LEC 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 **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, click **EDA Tool Settings**.

   If you are using 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.

   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 **Compilation Process Settings** and click **Physical Synthesis Optimizations**. The **Physical Synthesis Optimizations** page appears.
6. Turn off **Perform register retiming**.

If **Perform register retiming** is not turned off, an error occurs during compilation: "Physical Netlist Optimization Register retiming is not supported by Formal Verification tool Conformal LEC"

7. Under **Optimize for fitting (physical synthesis for density)**, turn off both **Perform physical synthesis for combinational logic** and **Perform logic to memory mapping** to prevent logic from being mapped to RAMs.

   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.

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.

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 21–4).

Table 21–4. Quartus II Software Compiler-Generated Files and Directories (Part 1 of 2)

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 information for instance equivalences.
	<*proj rev*>**.cep**	The <*proj rev*>**.cep** file contains information for black box pin equivalences in the design.
	<*proj rev*>**.cmp**	The <*proj rev*>**.cmp** file contains 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.

Table 21–4. Quartus II Software Compiler-Generated Files and Directories (Part 2 of 2)

File or Directory	Name	Details
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 21–4:

(1) This file is used with the Conformal LEC 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, Conformal LEC 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 Conformal LEC software. To learn more about black boxes, refer to "Black Boxes in the Conformal LEC Flow" on page 21–8.

Understanding the Formal Verification Scripts for Conformal LEC

The Quartus II software generates scripts to use with the Conformal LEC software. This section elaborates on the details of the Conformal LEC 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 Conformal LEC constraints to verify your netlists.

A sample script generated by the Quartus II software is provided in "Conformal Dofile/Script Example" on page 21–19.

Conformal LEC 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 Conformal LEC software to automatically map key points when the names are not the same:

```
add renaming rule <rule>
```

The 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 Conformal LEC software does not always automatically map all of the keypoints, or can incorrectly map some keypoints. To help the Conformal LEC 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 Conformal LEC <*file_name*>**.cmc** script.

```
add mapped points <key_points>
```

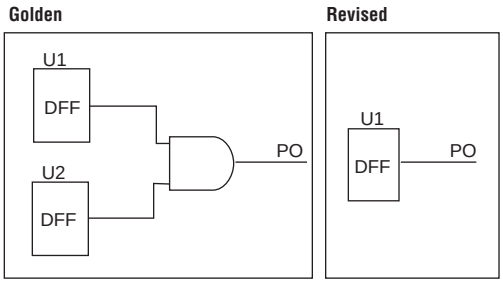
There are situations in which the inverter in front of the register is moved after the register. In this situation, the following command is used:

```
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 21–4. Instance Equivalence



During the process of optimization, the Quartus II software might merge two registers into one (Figure 21–4). 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 21–6.

```
add instance constraint <constraint_value>
```

Comparing Designs Using Conformal LEC

This section addresses using the Conformal LEC software to compare designs, and to prove logical equivalence between two versions of the design.

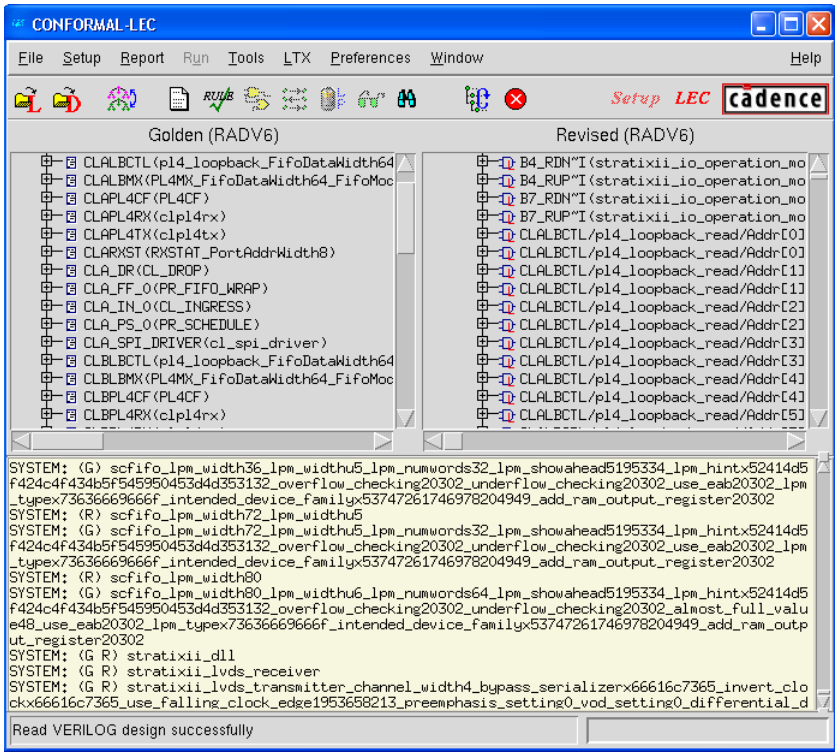
Running the Conformal LEC Software from the GUI

To run the Conformal LEC software from the GUI, follow these steps:

1. Open the Conformal LEC software.
2. On the File menu, click **Do Dofile**.
3. Select the file *<path to project directory>***/fv/conformal/***<proj rev>***.ctc**.

The Conformal LEC software GUI displays the comparison results (Figure 21–5 on page 21–15). 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 from 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 21–5. Conformal LEC 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 Conformal LEC 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 Conformal LEC software user documentation.

Running the Conformal LEC Software From a System Command Prompt

To run the Conformal LEC software without using the GUI, type the command shown in Example 21–6 at a system command prompt.

Example 21–6. 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, refer to the *Formal Verification Design Example* page of the Altera website.

For more information about the latest debugging tips and solutions for formal verification flow between the Conformal LEC software and the Quartus II software, go to www.altera.com and perform an advanced search with keywords "formal verification".

Known Issues and Limitations

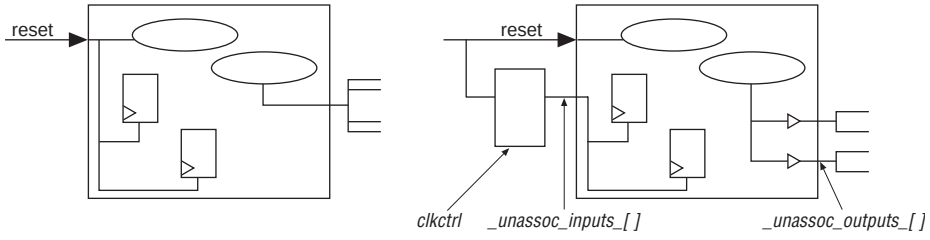
The following known issues and limitations can occur when using the formal verification flow described in this chapter:

- 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 21–6). This issue is generally associated with reset and enable signals. Figure 21–6 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 21–6 shows the creation of the `_unassoc_inputs_[]` and `_unassoc_outputs_[]` ports for the reset signal.

Figure 21–6. Creation of _unassoc_inputs_[] and _unassoc_outputs_[]

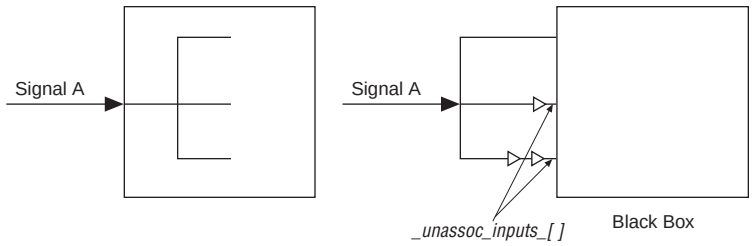


Another common occurrence of this issue is in HardCopy II designs. Whenever a port drives large fan-out within the black box, the Quartus II software inserts a buffer on the net and moves the logic outside of the black box (Figure 21–7 on page 21–17).

To fix the problem of `_unassoc_input_[]` ports causing black box mismatches, use Conformal LEC commands to change the type of the black box `unassoc_input_[]` keypoint to a primary output keypoint, and then mark the appropriate pin equivalences. Similarly, to fix the problem of register mismatches due to `_unassoc_output_[]` pins from black boxes, use Conformal LEC commands to change the type of the blackbox `_unassoc_output_[]` keypoint to a primary input, and then mark the equivalent pins as such. The commands to perform these actions are written in the *<proj rev>***.cep** file.

Figure 21–7 on page 21–17 shows the creation of `_unassoc_inputs_[]` for a signal with large fan-out.

Figure 21–7. Creation of _unassoc_inputs_[] for a Signal with Large Fan-out



- In designs with combinational feedback loops, the Conformal LEC 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 under the following conditions:
  - 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 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 sets, resets, or both. These cut points are also reported during the mapping step in Quartus II software with warning messages. You can add Conformal LEC 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 LEC software might report mismatches or abort comparisons 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 Conformal LEC 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.

For a complete list of supported black box models, refer to *Guidelines for Creating a Design for Use with the Encounter Conformal and Quartus II Software* in Quartus II Help.

Conformal Dofile/Script Example

The following example script (Example 21–7), generated by the Quartus II software, lists some of the setup commands used in Conformal LEC software:

Example 21–7. Conformal LEC Script (Part 1 of 2)

```
// 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
// 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
```

Example 21–7. Conformal LEC Script (Part 2 of 2)

```
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


// 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.

Document Revision History

Table 21–5 shows the revision history for this chapter.

Table 21–5. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	Updates for new GUI changes, and added link to Help.
November 2009	9.1.0	Updated "Black Boxes in the Encounter Conformal Flow" on page 18–8.
March 2009	9.0.0	Updated Table 21–1 on page 21–3.
November 2008	8.1.0	■ 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.
May 2008	8.0.0	■ 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.

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

Take an online survey to provide feedback about this chapter.

Section VI. Device Programming

The Quartus® II software offers a complete software solution for system designers who design with Altera® FPGA and CPLD devices, including device programming. 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 22, Quartus II Programmer

22. Quartus II Programmer

QII53022-10.0.0

The Quartus® II Programmer is part of the Quartus II software package, and allows you to program Altera CPLD and configuration devices and configure Altera® FPGA devices. The Quartus II software offers a complete software solution for system designers who design with Altera FPGA and CPLD 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"
- "Quartus II Programmer GUI" on page 22–3
- "Programming and Configuration" on page 22–4
- "Device Programming and Configuration" on page 22–5
- "Programming Files" on page 22–6
- "Flash Loaders" on page 22–10
- "Hardware Setup" on page 22–10
- "JTAG Chain Debugger Tool" on page 22–10
- "Stand-Alone Quartus II Programmer" on page 22–11
- "Scripting Support" on page 22–11

For more information about how to use the Quartus II Programmer GUI to program and configure your device, refer to *About Programming* in Quartus II Help.

Programming Flow

The programming flow is as follows:

1. Design compilation, in which the Quartus II Assembler generates the programming or configuration file.
2. Perform programming or configuration file conversion for different configuration devices, or optional programming and configuration file creation.
3. Configure or program the FPGA, CPLD, or configuration devices with the programming or configuration file with the Quartus II Programmer.

Figure 22–1 shows the programming file generation flow. This flow covers the types of configuration and programming files that you use with the Quartus II Programmer. Refer to "Programming Files" on page 22–6 for information about other optional programming files.

Figure 22–1. Programming File Generation Flow

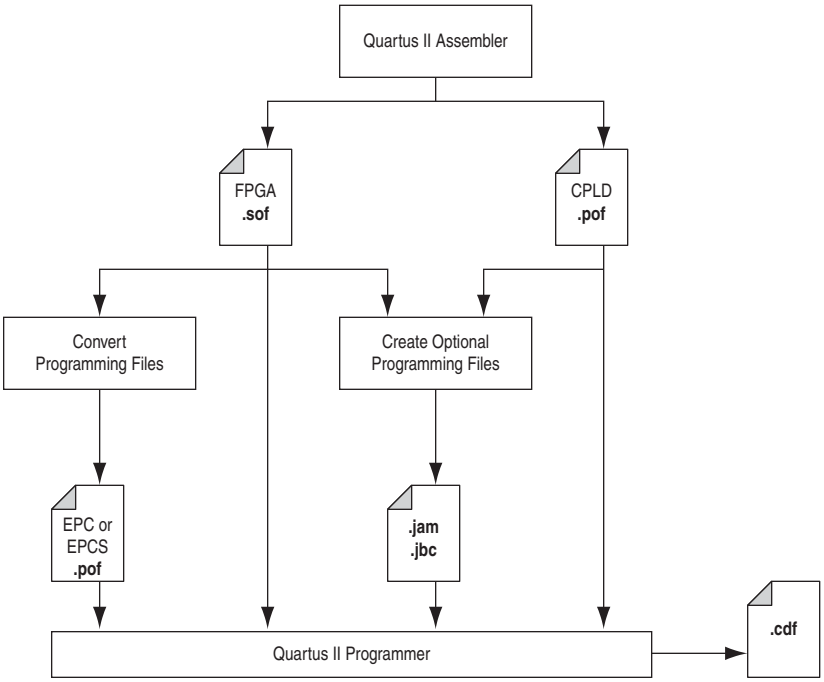


Table 22–1 shows the programming and configuration file formats supported by Altera FPGAs, CPLDs, configuration devices, enhanced configuration devices, and serial configuration devices. Chain description files (**.cdf**) are described in *About Programming* in the Quartus II Help.

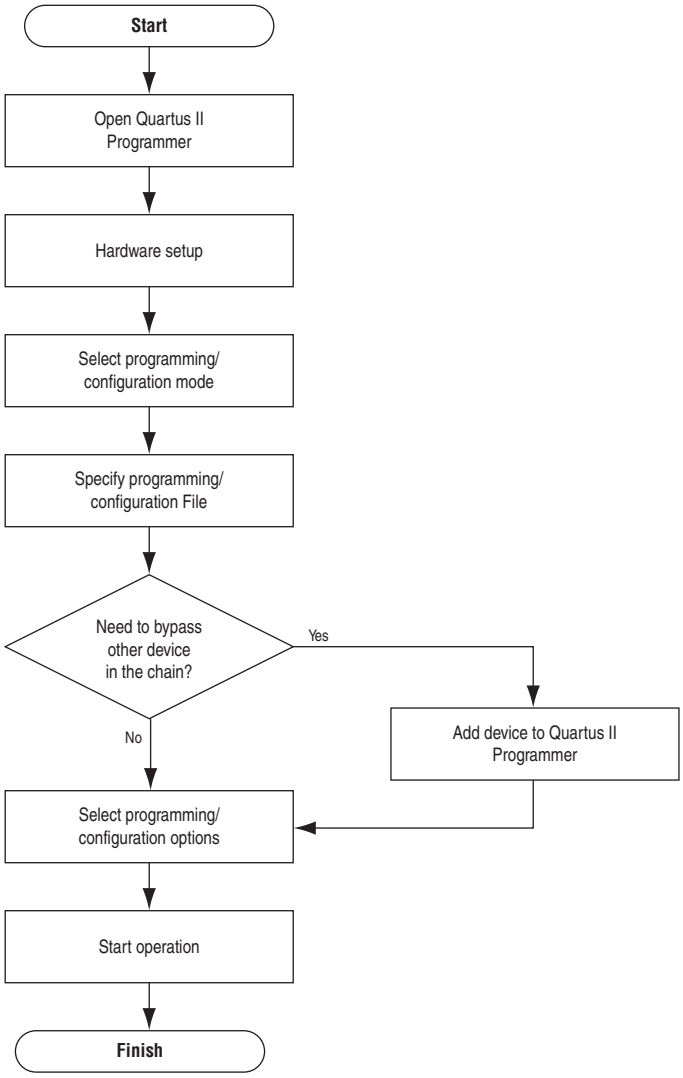
Table 22–1. Programming and Configuration File Format

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 22–2 shows the programming flow using the Quartus II Programmer.

Refer to "Device Programming and Configuration" on page 22–5 for information about programming or configuring the device.

Figure 22–2. Programming Flow



Quartus II Programmer GUI

The Quartus II Programmer 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**. The status of each operation, whether programming is successful or not, is reported in the Quartus II message window.

For a description of the Programmer window, refer to *Programmer Window* in Quartus II Help.

Table 22–2 shows the programming and configuration options supported by Altera devices.

Table 22–2. 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 22–2:

(1) Only MAX II and MAX 7000B CPLDs support the ISP Clamp feature.
(2) **The .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 a description of these options, refer to *Programmer Page (Options Dialog Box)* in Quartus II Help.

Programming and Configuration

The following section describes the Quartus II Programmer and the Programmer configuration modes in more detail.

Configuration Modes

The Quartus II Programmer supports the four configuration modes, including JTAG, Passive Serial (PS), Active Serial (AS), and In-Socket modes (ISM).

For more information about how to use these different configuration modes, refer to *About Programming* in Quartus II Help.

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.

For more information about PS configuration mode and PS pin connection, refer to the *Configuration Handbook* or the chapter about configuration in the appropriate FPGA device handbook.

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.

Refer to www.altera.com or the Quartus II Help for a list of programming adapters available for Altera devices.

Table 22–3 shows the programming and configuration modes supported by Altera devices.

Table 22–3. 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 22–3:

(1) MAX II CPLDs do not support in-socket programming mode.

Device Programming and Configuration

With the Quartus II Programmer you can program and configure one or more devices. This section provides references to Quartus II Help procedures that describe how to program or configure Altera devices, as well as how to bypass Altera and non-Altera devices in a JTAG chain.

To program or configure a single device with the Quartus II Programmer, refer to *Programming Devices* in Quartus II Help.

To program or configure a device in JTAG and AS programming modes, refer to *Programming Devices* in Quartus II Help.

For a description of Chain Description Files, refer to *About Programming* in Quartus II Help.

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*.

Programming Files

The Quartus II software can generate optional programming or configuration files in various formats you can use with programming tools other than the Quartus II Programmer. When you compile a design in the Quartus II software, the Programmer automatically generates either a **.sof** or **.pof** file. Programmer also allows you to convert FPGA configuration files to programming files for configuration devices. For more information, refer to *About Optional Programming Files* in Quartus II Help.

Refer to file format topics in 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 Secondary Programming Files

The Quartus II software generates programming files of various formats for use with different programming tools. File types generated by the Quartus II software and also supported by the Quartus II Programmer include **.sof**, **.pof**, **.jam**, **.jbc**, and **.jic**. File types generated by the Quartus II software that are not supported by Programmer include **.svf**, **.isc**, **.hexout**, **.rbf**, **.ttf**, and **.rpd**.

For more information, refer to *Generating Secondary Programming Files* in Quartus II Help.

Convert Programming Files Dialog Box

The **Convert Programming Files** dialog box contains the following three sections:

- **Conversion setup file**—Allows you to import input file information from other files and save the conversion setup information for future use.
- **Output programming file**—Allows you to specify programming file types, configuration device types and configuration modes, file names, and Remote/Local update difference files. You can also add new files.
- **Advanced Options**—Allows you to debug configuration issues.

CAUTION

Use the **Advanced** option only when you encounter an issue with device configuration.

The **Convert Programming Files** dialog box in the Programmer allows you to convert programming files from one file format to another. For example, to store the FPGA data in configuration devices, you can convert the **.sof** data to another format and then program the configuration device. You can also convert FPGA data into other file formats, including **.pof**, **.hexout**, **.rbf**, **.rpd**, or **.jic**.

You must have the Programmer open to access the **Convert Programming Files** dialog box. Click **File** on the Quartus II main menu, and then click **Convert Programming Files**. You can then perform the following tasks:

- Configure multiple devices, such as combining multiple **.sof** files into one **.pof** file.
- Configure multiple devices using an external host, such as a microprocessor or CPLD. For example, you can combine multiple **.sof** files into one configuration file.

For more information about converting programming files with the Quartus II software, refer to the *Configuration File Formats* chapter of the *Configuration Handbook*.

You can use the **Advanced** option in the **Convert Programming File** dialog box to debug your configuration.

When you change settings in the **Advanced** option, the change affects **.pof**, **.jic**, **.rpd**, and **.rbf** files.

You must choose the setting that applies to your Altera device. You can force the Quartus II software to enable the option by turning the options on or off in the **Advanced Options** dialog box.

Table 22–4 describes the **Advanced Options** settings in further detail. Table 22–5 lists possible symptoms you may encounter when a configuration fails, as well as describes the **Advanced Options** you should use to debug your configuration.

Table 22–4. Advanced Options Settings

Option Setting	Description
Disable EPCS ID check	■ FPGA skips the EPCS silicon ID verification. ■ Default setting check box is grayed (EPCS ID check is enabled). ■ Applies to the single and multi device (AS) configuration modes on all FPGA devices.
Disable AS mode CONF_DONE error check	■ FPGA skips the CONF_DONE error check. ■ Default setting check box is grayed (EPCS ID check is enabled). ■ Applies to single- and multi-device (AS) configuration modes on all FPGA devices.
Program length count adjustment	■ Specifies the offset to be applied to the computed program length count of the entire bitstream. ■ Default setting is 0. The value must be an integer. ■ Applies to single- and multi-device (AS) configuration modes on all FPGA devices.
Post-chain bitstream pad bytes	■ Specifies the number of pad bytes appended to the end of an entire bitstream. ■ Default value is set to 0 if the bitstream of the last device is uncompressed. Set to 2 if the bitstream of the last device is compressed.
Post-device bitstream pad bytes	■ Specifies the number of pad bytes appended to the end of the bitstream of a device. ■ Default value is 0. No negative integer. ■ Applies to all single-device configuration modes on all FPGA devices.
Bitslice padding value	■ Specifies the padding value used to prepare bitslice configuration bitstreams, such that all bitslice configuration chains simultaneously receive their final configuration data bit. ■ Default value is 1. Valid setting is 0 or 1. ■ Use only in 2, 4, and 8-bit Passive Serial, when you use an EPC device with the decompression feature enabled. ■ Applies to all FPGA devices that support enhanced configuration devices.

Table 22–5. Failure Symptoms and Options Settings (Part 1 of 2)

Failure Symptoms	Disable EPCS ID check	Disable AS mode CONF_DONE error check	Program length count adjustment	Post-chain bitstream pad bytes	Post-device bitstream pad bytes	Bitslice padding value
Configuration failure occurs at end of configuration cycle. Decompression feature is enabled. Encryption feature is enabled.	—	—	—	—(3)	—(4)	—
CONF_DONE stays low at end of configuration cycle.		—	— (1)	— (3)	— (4)	

Table 22–5. Failure Symptoms and Options Settings (Part 2 of 2)

Failure Symptoms	Disable EPCS ID check	Disable AS mode CONF_DONE error check	Program length count adjustment	Post-chain bitstream pad bytes	Post-device bitstream pad bytes	Bitslice padding value
CONF_DONE goes high momentarily at end of configuration cycle.		—	— (2)			
FPGA does not enter user mode even though CONF_DONE is released high.				— (3)	— (4)	
Configuration failure occurs at beginning of configuration cycle.	—					
Newly introduced EPCS is used, such as EPCS128.	—					
Failure in **.pof** generation for EPC device using Quartus II Convert Programming File Utility when decompression feature is enabled.						—

Notes to Table 22–5:

(1) Start with positive offset to Program Length Count Adjustment.
(2) Start with negative offset to Program Length Count Adjustment.
(3) Use only for multi-device chain.
(4) Use only for single-device chain.

Refer to Quartus II Help or the *Configuration File Formats* chapter of the *Configuration Handbook* for more information about the programming and configuration file formats.

Input files to convert

The **Input files to convert** option in the **Convert Programming Files** dialog box allows you to add one or more files. You can also convert files when you click **Generate** in the **Convert Programming Files** dialog box.

For more information about converting programming files with the Quartus II software, refer to the *Configuration File Formats* chapter of the *Configuration Handbook*.

For more information about the **Convert Programming Files** dialog box, refer to *Convert Programming Files Dialog Box* in Quartus II Help.

Flash Loaders

Parallel and serial configuration devices do not support the JTAG interface and cannot be programmed directly through the normal JTAG programming. You can use a flash loader to program serial configuration devices in-system via the JTAG interface. To do so, use an FPGA as a bridge between the JTAG interface and the serial configuration device. Altera has two types of flash loaders: parallel and serial.

For more information, refer to *About Flash Loaders* in Quartus II Help.

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.

For hardware settings, refer to *Setting Up Programming Hardware* in Quartus II Help.

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. 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 add or remove JTAG servers, refer to *Programming Devices* in Quartus II Help.

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.

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 purposes.

JTAG Chain Debugger Example

Perform the following steps to use the JTAG Debugger to check the FPGA user code:

1. Open the JTAG Debugger.
2. Click **Device** on the Assignment menu.

3. Select **Device and Pin Option** and click the **General** tab.
4. In the **JTAG user code** column, type `abcdefcb`.
5. Click **OK**.
6. Click **JTAG Chain Debugger** on the Tools menu.
7. Send an instruction to the JTAG chain by setting up the debugger as follows:
   - Command: Scan Instruction Register
   - TAP State: End scan in RTI state
   - Clocks: 10
   - TDI: 0x07
8. Click **Run**.
9. Verify that the TDO value is `0x155`.
10. Read the value in the data register by setting up the debugger as follows:
    - Command: Scan Data Register
    - Tap State: End scan in RTI state
    - Clock:40
    - TDI: 0000 0000 (any value)
11. Click **Run**.
12. Verify that the TDO value is `0x00ABCDEFAB`.

For more information, refer to *About the JTAG Chain Debugger* in Quartus II Help.

Stand-Alone Quartus II Programmer

If you do not have the full version of the Quartus II software, Altera offers the free Quartus II Stand-Alone Programmer. The stand-alone programmer has the same full functionality as the Quartus II Programmer in the Quartus II software, 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.

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 22–1 shows a command that programs a device:

Example 22–1. 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 ↵
```

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 ↵
```

The help switch does not reference the -n switch, which you may find to be the most useful switch. `jtagconfig -n` enumerates each node for each jtag device.

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.

Document Revision History

Table 22–6 shows the revision history for this chapter.

Table 22–6. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Added links to Quartus II Help ■ Deleted screen shots
November 2009	9.1.0	No change to content
March 2009	9.0.0	■ Added a row to Table 21–4 on page 21–7 ■ Changed references from "JTAG Chain Debug" to "JTAG Chain Debugger" ■ Updated figures
November 2008	8.1.0	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
May 2008	8.0.0	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.

Take an *online survey* to provide feedback about this handbook chapter.

About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 10.0.

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 10.0 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.
	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 10.0

Volume 4: SOPC Builder

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

QII5V4-10.0

Copyright © 2010 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 vii

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
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 SOPC Builder Systems 1-6
Operating System Support 1-6
Talkback Support 1-7
Document Revision History 1-7

Chapter 2. System Interconnect Fabric for Memory-Mapped Interfaces

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
Native Address Alignment 2-8
Arbitration for Multimaster Systems 2-9
Traditional Shared Bus Architectures 2-9
Slave-Side Arbitration 2-10
Arbiter Details 2-11
Arbitration Rules 2-12
Burst Adapters 2-14
Interrupts 2-15
Individual Requests IRQ Scheme 2-15
Priority Encoded Interrupt Scheme 2-15
Assigning IRQs in SOPC Builder 2-16
Reset Distribution 2-16
Document Revision History 2-17

Chapter 3. System Interconnect Fabric for Streaming Interfaces

High-Level Description 3-1
Avalon Streaming and Avalon Memory-Mapped Interfaces 3-2

Adapters . . . 3-3
Data Format Adapter . . . 3-3
Timing Adapter . . . 3-4
Channel Adapter . . . 3-4
Error Adapter . . . 3-4
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
Document Revision History . . . 3-7

Chapter 4. SOPC Builder Components

Component Providers . . . 4-1
Component Hardware Structure . . . 4-2
Component Instances Inside the SOPC Builder System . . . 4-2
Components Outside the SOPC Builder System . . . 4-3
Exported Connection Points—Conduit Interfaces . . . 4-3
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 . . . 4-7
Upgrading from Earlier Versions . . . 4-8
Component Structure . . . 4-8
Component Description File (_hw.tcl) . . . 4-8
Component File Organization . . . 4-9
Component Versioning . . . 4-9
Classic Components in SOPC Builder . . . 4-10
Document Revision History . . . 4-10

Chapter 5. Using SOPC Builder with the Quartus II Software

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
Document Revision History . . . 5-7

Chapter 6. Component Editor

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
HDL Parameters Tab . . . 6-6
Library Info . . . 6-7

Saving a Component 6-7
Editing a Component 6-8
Software Assignments 6-8
Component Parameterization 6-8
Document Revision History 6-8

Chapter 7. Component Interface Tcl Reference
Information in a Hardware Component Description File 7-1
Component Phases 7-2
Writing a Hardware Component Description File 7-2
Providing Basic Information 7-3
Declaring Parameters 7-3
Declaring Interfaces 7-4
Adding Files and Guiding Generation 7-5
Default Behaviors 7-5
Validation Phase Behavior 7-6
Elaboration Phase Behavior 7-6
Generation Phase Behavior 7-7
Edit Phase Behavior 7-7
Overriding Default Behaviors 7-8
Validation Callback 7-9
Elaboration Callback 7-9
Generation Callback 7-10
Editor Callback 7-11
Hardware Tcl Command Reference 7-12
Module Definition 7-14
Parameters 7-21
Display Items 7-29
Interfaces and Ports 7-32
Generation 7-39
Deprecated Commands and Properties 7-41
Document Revision History 7-42

Chapter 8. Archiving SOPC Builder Projects
Limitations 8-1
Required Files 8-2
Document Revision History 8-3

Section II. Building Systems with SOPC Builder

Chapter 9. SOPC Builder Memory Subsystem Development Walkthrough
Example Design 9-1
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

On-Chip RAM and ROM . . . 9-6
Component-Level Design for On-Chip Memory . . . 9-6
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
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
SOPC Builder System-Level Design for SRAM and Flash Memory . . . 9-17
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
Example Design with SRAM and Flash Memory . . . 9-20
Document Revision History . . . 9-24

Chapter 10. SOPC Builder Component Development Walkthrough

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
Sharing Components . . . 10-6
System Information Files (.sopcinfo) . . . 10-7
Document Revision History . . . 10-8

Section III. Interconnect Components

Chapter 11. Avalon Memory-Mapped Bridges

Structure of a Bridge . . . 11-1
Reasons for Using a Bridge . . . 11-2
Address Mapping for Systems with Avalon-MM Bridges . . . 11-5

Avalon-MM Pipeline Bridge 11-7
Component Overview 11-7
Functional Description 11-8
Clock Crossing Bridge 11-10
Choosing Clock Crossing Methodology 11-10
Functional Description 11-11
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
Document Revision History 11-23

Chapter 12. Avalon Streaming Interconnect Components

Interconnect Component Usage 12-1
Address Mapping 12-2
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
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 10.0 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: *July 2010*
Part Number: *QII54001-10.0.0*

Chapter 2 System Interconnect Fabric for Memory-Mapped Interfaces
Revised: *July 2010*
Part Number: *QII54003-10.0.0*

Chapter 3 System Interconnect Fabric for Streaming Interfaces
Revised: *July 2010*
Part Number: *QII54019-10.0.0*

Chapter 4 SOPC Builder Components
Revised: *July 2010*
Part Number: *QII54004-10.0.0*

Chapter 5 Using SOPC Builder with the Quartus II Software
Revised: *July 2010*
Part Number: *QII54023-10.0.0*

Chapter 6 Component Editor
Revised: *July 2010*
Part Number: *QII54005-10.0.0*

Chapter 7 Component Interface Tcl Reference
Revised: *July 2010*
Part Number: *QII54022-10.0.0*

Chapter 8 Archiving SOPC Builder Projects
Revised: *July 2010*
Part Number: *QII54017-10.0.0*

Chapter 9 SOPC Builder Memory Subsystem Development Walkthrough
Revised: *July 2010*
Part Number: *QII54006-10.0.0*

Chapter 10 SOPC Builder Component Development Walkthrough
Revised: *July 2010*
Part Number: *QII54007-10.0.0*

Chapter 11 Avalon Memory-Mapped Bridges
Revised: *July 2010*
Part Number: *QII54020-10.0.0*

Chapter 12 Avalon Streaming Interconnect Components
Revised: *July 2010*
Part Number: *QII54021-10.0.0*

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

QII54001-10.0.0

Quick Start Guide

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. 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, for example a Nios® II processor. An *instance* is a parameterization of a component that's been added to a system, for example `cpu_0`.

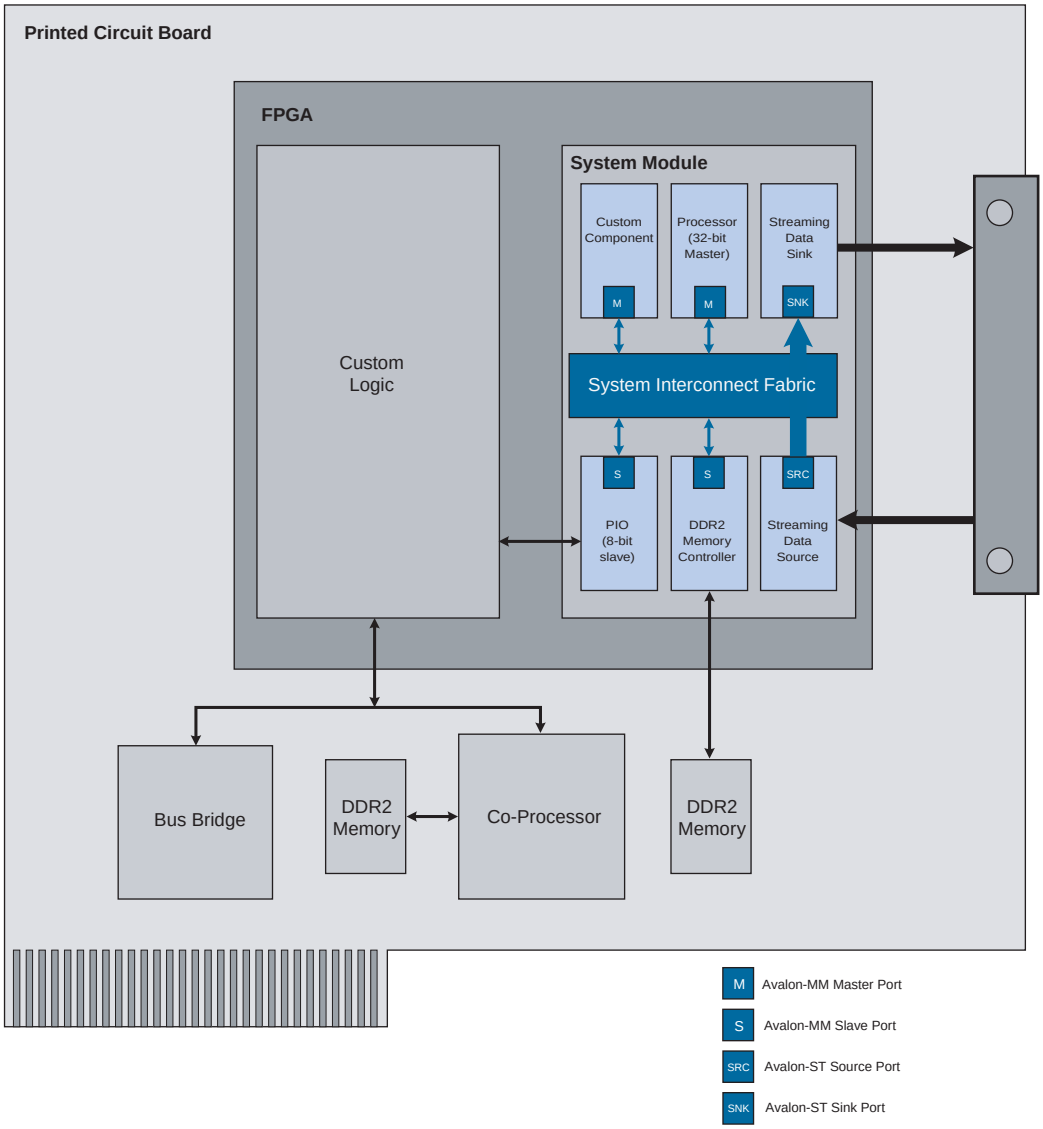
SOPC Builder modules are the building blocks for creating an SOPC Builder system. SOPC Builder modules use Avalon® interfaces, such as memory-mapped, streaming, and IRQ, for the physical connection of components. You can use SOPC Builder to connect any logical device (either on-chip or off-chip) that 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 may include constraints, software drivers, simulation models, and reference designs when applicable.

To find SOPC Builder Ready 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 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 SOPC Builder 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, instance name, interface type, or using custom tags. For example, you can use filtering to view only instances that include an Avalon-MM interface or instances 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 details on installation and licensing, refer to the *Altera Software Installation and Licensing Manual*.

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 Altera components you use, including: interface types, interface properties, parameter names and values, clocking, and software assignments. For components from Altera, Talkback sends the component parameter values to help understand what features of the component are being used. For non-Altera components, Talkback collects information about how interfaces such as Avalon-MM are being used. Connectivity between components is not sent. The Talkback file does not include information about system connectivity, interrupts, or the memory map seen by each master in the system. Talkback collects the same very general information about your proprietary components.

The Talkback feature is enabled by default. You can disable Talkback from within the Quartus II software if you do not wish to share your usage data with Altera.

Document Revision History

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

Table 1–1. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Revised opening paragraph.
November 2009	9.1.0	■ Clarified information collected by Talkback.
March 2009	9.0.0	■ 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.
November 2008	8.1.0	■ Expanded description of **.sopcinfo** file ■ Changed page size to 8.5 x 11 inches
May 2008	8.0.0	■ 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.

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

Take an online survey to provide feedback about this handbook chapter.

2. System Interconnect Fabric for Memory-Mapped Interfaces

QII54003-10.0.0

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 less logic, 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 2–14
- "Interrupts" on page 2–15
- "Reset Distribution" on page 2–16

For details about the Avalon-MM interface, refer to the *Avalon Interface Specifications*.

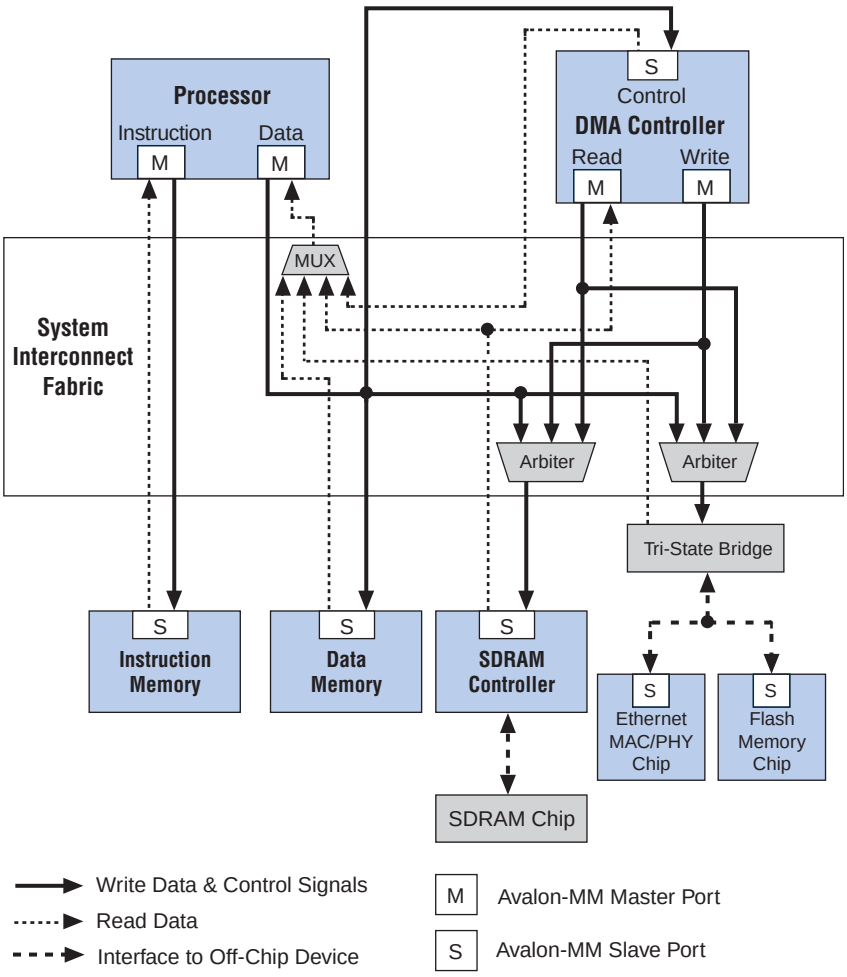
System interconnect fabric for memory-mapped interfaces supports the following items:

- 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 2–14
- "Interrupts" on page 2–15
- "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

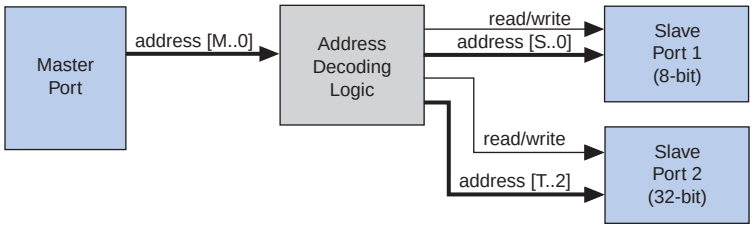
Address decoding logic in the system interconnect fabric forwards appropriate addresses to each slave. Address decoding logic simplifies component design in the following ways:

- 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

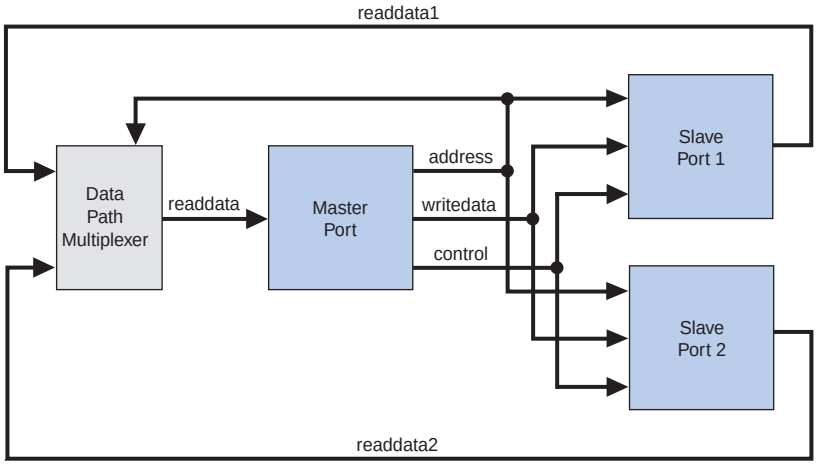
Module Name	Description	Base	End	IRQ
⊟ cpu	Nios II Proces...			
instruction_master	Master port			
data_master	Master port	IRQ 0	IRQ 31	
jtag_debug_mod...	Slave port	0x02120000	0x021207FF	
⊞ ext_flash	Flash Memory...	0x00000000	0x007FFFFF	
⊞ ext_ram	IDT71V416 S...	0x02000000	0x020FFFFF	
⊞ ext_ram_bus	Avalon Tri-St...			
⊞ button_pio	PIO (Parallel I/O)	0x02120860	0x0212086F	2
⊞ high_res_timer	Interval timer	0x02120820	0x0212083F	3

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



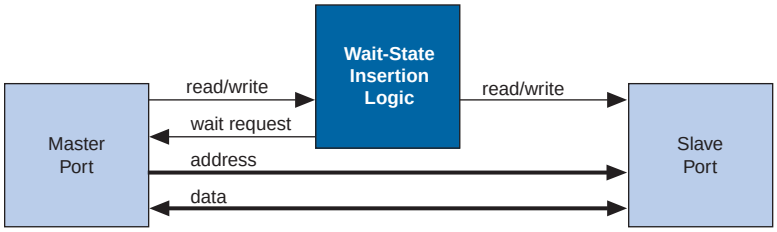
In SOPC Builder, the generation of datapath multiplexing logic is specified using the connections panel on the **System Contents** tab.

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]

When connecting a wide master to a narrow slave port that uses native addressing, the following addressing formula should be used to determine what address to present to the system interconnect fabric:

```
<master address> = <slave base address> + (<slave word offset> *
<master data width in bytes>)
```

For example, a 64-bit master needs to write to the second word of a 32-bit slave that uses native addressing the formula would reduce to:

```
<master address> = <slave base address> + (1 * 8)
```

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.

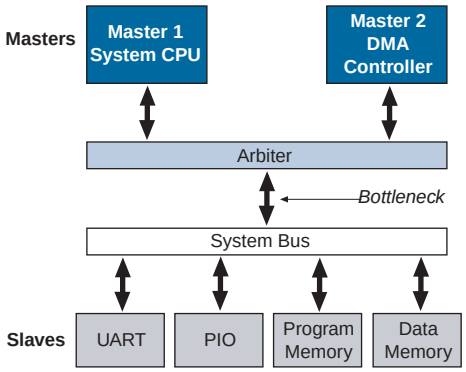
Traditional Shared Bus Architectures

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

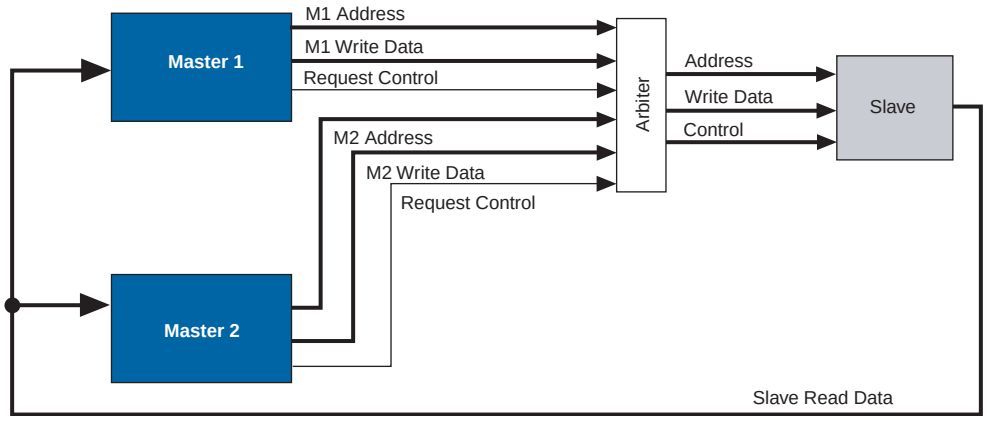


Slave-Side Arbitration

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



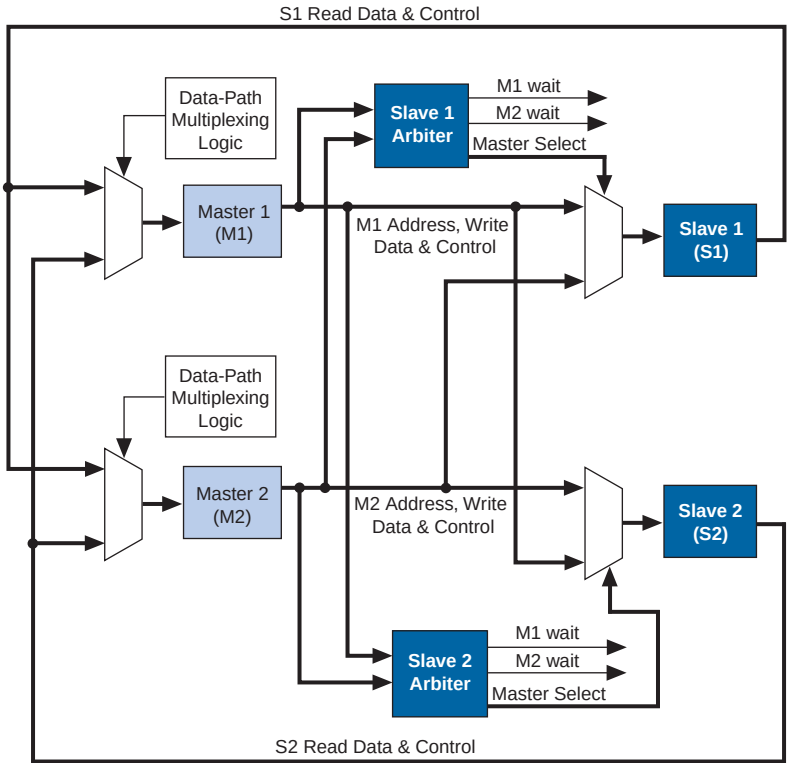
Arbiter Details

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



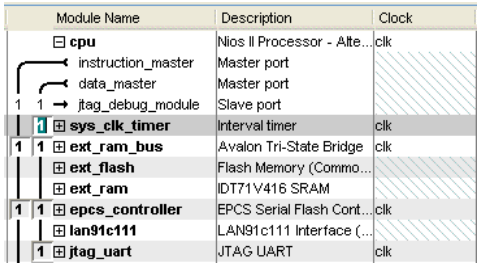
Arbitration Rules

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



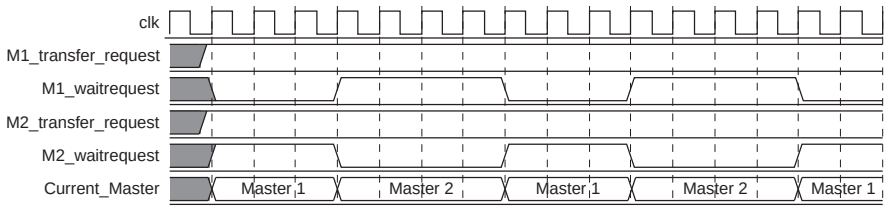
The arbitration settings are hidden by default. To see them, on the View menu, click **Show Arbitration**.

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.

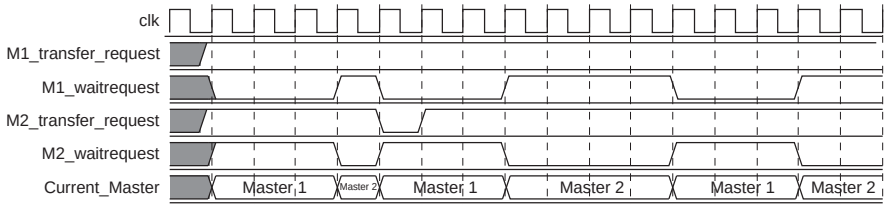
For example, assume that two masters continuously attempt to perform back-to-back transfers to a slave. Master 1 is assigned three shares and Master 2 is assigned four shares. In this case, the arbiter grants Master 1 access for three transfers, then Master 2 for four transfers. This cycle repeats indefinitely. 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 these cases the burst adapter is not capable of adapting the master and slave pairing. An adapter is generated; however, if the master performs a burst transaction to the slave that crosses the slave burst boundary data corruption can occur. To avoid a functional failure, you should ensure the master posts bursts of length one until the master burst boundary has been reached. The master burst boundary has an alignment of `<master_data_width> × <master_maximum_burst_length>`.

Any burst transaction that begins on a master burst boundary is guaranteed to not cross the burst boundary of the slave port regardless of the slave port's maximum burst length. Typically the only Avalon-MM interfaces that support burst wrapping are burst capable SDRAM controllers.

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.

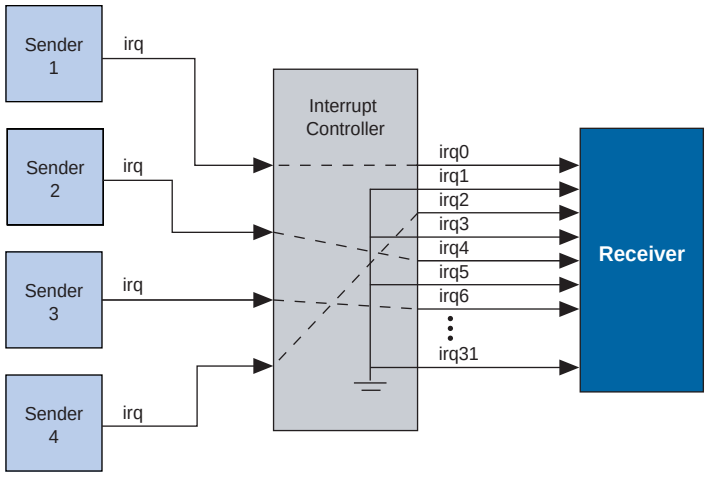
For further information, refer to the *Interrupt Interfaces* chapter in the *Avalon Interface Specifications*.

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

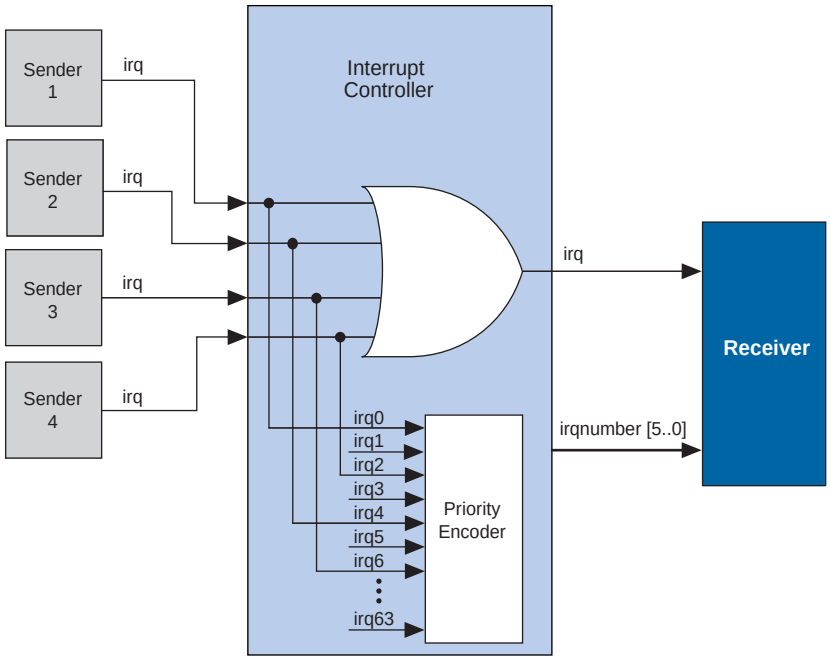


Priority Encoded Interrupt Scheme

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 ORed 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.

Document Revision History

Table 2–4 shows the revision history for this chapter.

Table 2–4. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	■ Removed reference to `chipselect` signal which has been removed in favor of both `read` and `write` signals.
November 2009	9.1.0	■ Revised discussion of a non-linewrap burst master connected to a slave with the `linewrapBursts` property set to `TRUE`.
March 2009	9.0.0	■ Added table showing the behavior of the burst adapter for master and slaves with and without `linewrapBursts` set to `TRUE`.
November 2008	8.1.0	■ 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
May 2008	8.0.0	■ 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.

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

Take an online survey to provide feedback about this handbook chapter.

3. System Interconnect Fabric for Streaming Interfaces

QII54019-10.0.0

The interconnect fabric for Avalon® Streaming 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

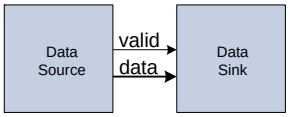
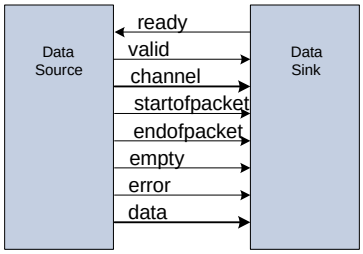


Figure 3–2 illustrates a more extensive interface that includes signals indicating the start and end of packets, channel numbers, error conditions, and back pressure.

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*.

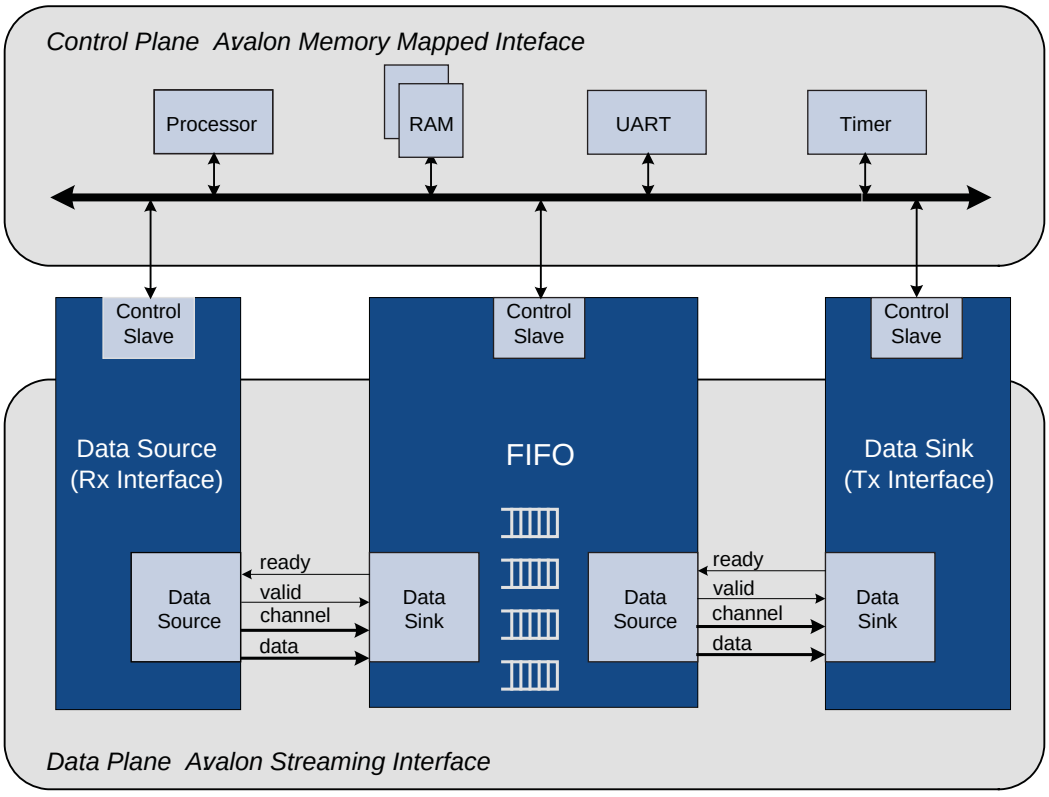
Avalon Streaming and Avalon Memory-Mapped Interfaces

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.

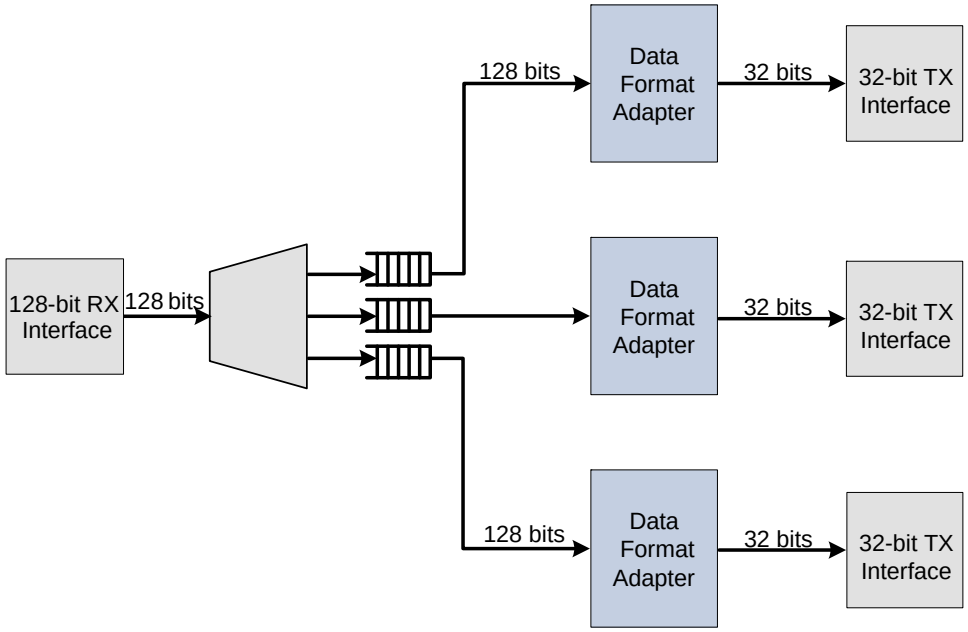
For complete information about these adapters, refer to the *Avalon Streaming Interconnect Components* chapter in volume 4 of the *Quartus II Handbook*.

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

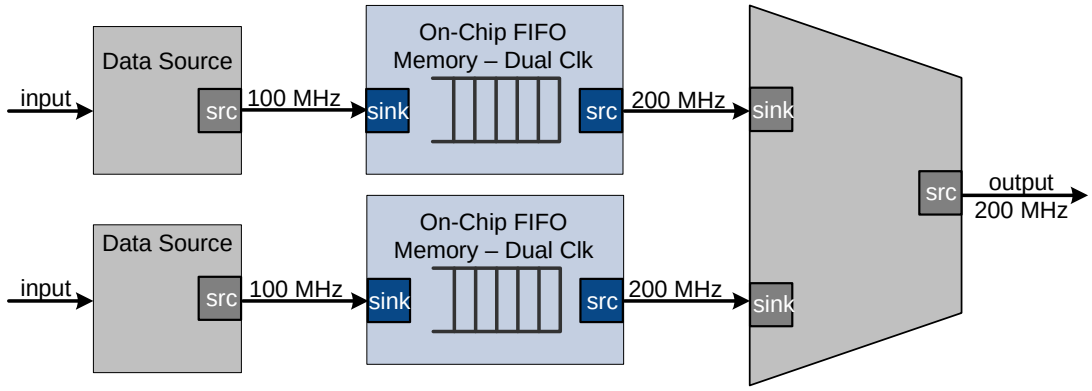
You can combine these adapters with streaming components to create datapaths whose input and output streams have different properties. The following sections provide examples of datapaths constructed using SOPC Builder in which the output stream is higher performance than the input stream:

- 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.

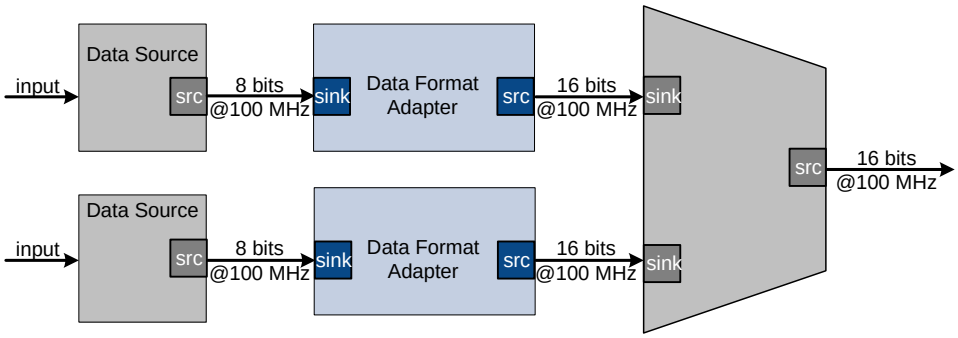
Figure 3–5. Datapath that Doubles the Clock Frequency



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

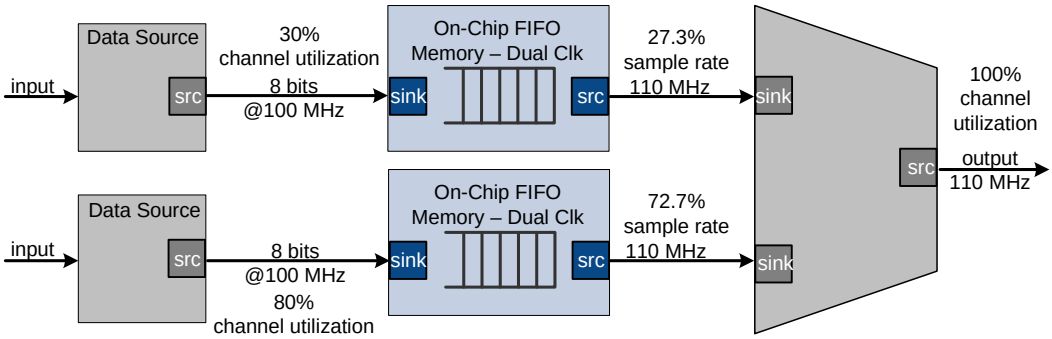


Example to Boost the Frequency

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



Document Revision History

Table 3–1 shows the revision history for this chapter.

Table 3–1. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	No changes from previous release.
November 2009	9.1.0	No changes from previous release.
March 2009	9.0.0	No changes from previous release.
November 2008	8.1.0	■ Added information on error adapter. ■ Changed page size to 8.5 x 11 inches
May 2008	8.0.0	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.

Take an online survey to provide feedback about this handbook chapter.

4. SOPC Builder Components

QII54004-10.0.0

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 the parameters that determine the operation of the component.
- A GUI for parameterizing 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"
- "Component Hardware Structure" on page 4–2
- "Exported Connection Points—Conduit Interfaces" on page 4–3
- "SOPC Builder Component Search Path" on page 4–4
- "Component Structure" on page 4–8
- "Classic Components in SOPC Builder" on page 4–10

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.

For more information about the **_hw.tcl** file, refer to the *Component Editor* chapter in volume 4 of the *Quartus II Handbook*.

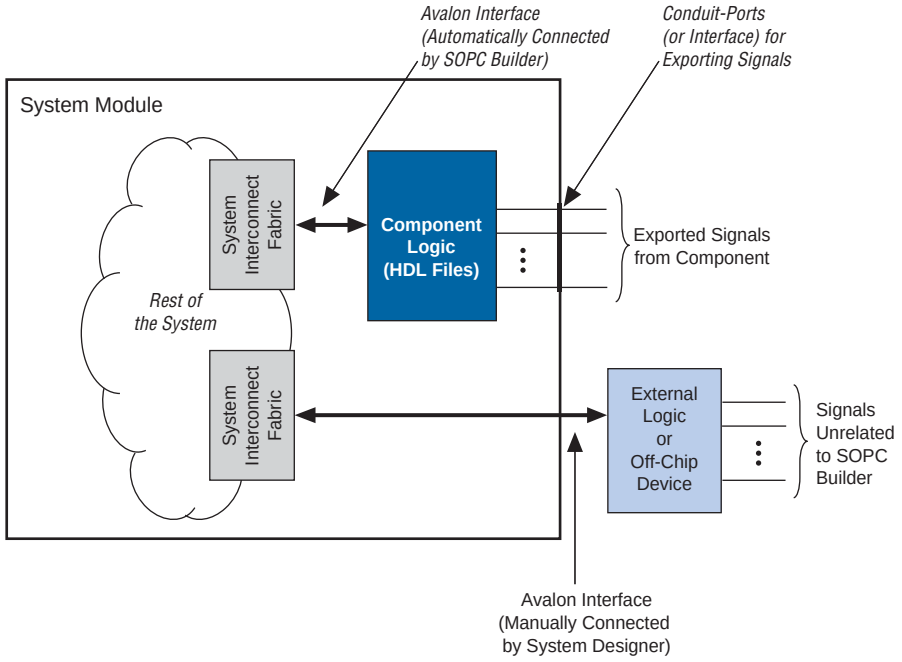
Component Hardware Structure

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



Component Instances 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 that accept Verilog HDL parameters or VHDL generics. Examples of parameters that can be expressed as Verilog HDL parameters or VHDL generics are address and data widths and FIFO depths. These components have HDL files that are not generated as a function of the parameterization, and are referred to as static HDL components. SOPC Builder automatically generates the top-level HDL wrapper file to apply parameter values for static components.

Generated HDL Components

Alternatively, you can also define a component whose HDL is generated based on the value of its declared parameters. These components use a custom generation callback to generate the HDL for each use of the component, instead of having SOPC Builder create an HDL wrapper that specifies these values. An example of a parameter that requires generated HDL is a parameter that controls the number of interfaces.

Composed HDL Components

Composed components are constructed from combinations of other components.You can use a compose callback to connect and parameterize a composed component; however, a custom compose callback may not be necessary for very simple composed components.

For more information about defining your own generation or compose callback procedure, refer to the *Generation Callback* and *Compose Callback* sections 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.

This type of component is deprecated and will not be available in future versions of the Quartus II software.

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 the directories in the IP search path 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_INSTALLDIR/../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. The search path is ultimately defined by the file, *<$QUARTUS_INSTALLDIR>***/sopc_builder/bin/root_components.ipx**.

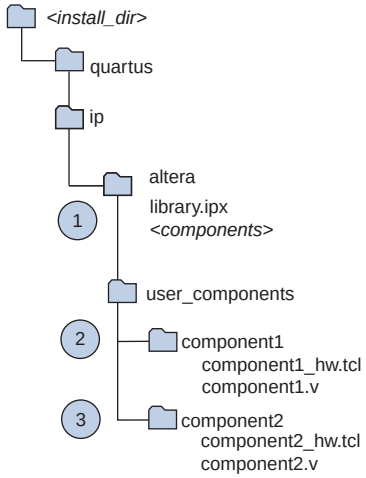
Installing Additional Components

There are a few different ways to make your components available to SOPC Builder projects. The following sections describe some of 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.

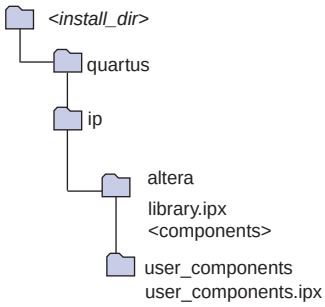
1. SOPC Builder recursively searches the ***<install_dir>*/ip/** directory by default. It finds the file in the `altera` subdirectory, which tells it about all of the Altera components. **library.ipx** includes listings for all components found in its subdirectories. The recursive search stops when SOPC 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**. When SOPC Builder finds that component, the recursive descent stops so that no components in subdirectories of **component1** are found.
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 **_hw.tcl** file in the *<install_dir>***/ip/** directory, SOPC Builder finds your **_hw.tcl** 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.

You can verify that components are available and also decrease the time it takes to launch SOPC Builder by using two utilities, `ip-catalog` and `ip-make-ipx`. The following sections describe these utilities.

ip–catalog

Shows the a catalog of components in either plain text or XML format.

Usage

```
ip-catalog --project-dir[=<directory>] --name[=<value>]
--verbose[=<true|false>] --xml[=<true|false>] --help
```

Options

- `--project-dir[=<directory>]`. Optional. Components can be found in certain locations relative to the project, if any. By default, the current directory, '.' is used. To exclude any project directory, use ".
- `--name[=<value>]`. Optional. This argument provides a pattern to filter the names of the components found. To show all components, use a * or ' '. By default, all components are shown. The argument is not case sensitive.

- `--verbose[=<true|false>]`. Optional. When true, reports the progress of the command.
- `--xml[=<true|false>]`. Optional. When true, prints the output in XML format instead of a line- and colon-delimited format.
- `--help`. Shows help for the `ip-catalog` command.

ip-make-ipx

This command creates an index file for the directory specified. It returns a 0 for successful completion and a non-zero value for failure.

Usage

```
ip-make-ipx --source-directory[=<directory>] --output[=<file>]
--relative-vars[=<value>] --thorough-descent
--message-before[=<value>] --message-after[=<value>] --help
```

Options

- `--source-directory=<directory>`. Optional. The directory to index. The default directory is "**.**". You can also provide a comma separated list of directories.
- `--output[=<file>]`. Optional. The name of the file to generate. The default name is **./components.ipx**.
- `--relative-vars[=<value>]`. Optional. Causes the output file to include references relative to the specified variable or variables where possible. You can specify multiple variables as a comma-separated list.
- `--thorough-descent[=<true|false>]`. Optional. If set, a component or **.ipx** file in a directory does not prevent subdirectories from being searched.
- `--message-before[=<value>]`. Optional. A message to print to `stdout` when indexing begins
- `--message-after[=<value>]`. Optional. A message to print to `stdout` when indexing completes
- `--help`. Show help for this command

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*>.

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.
  - <*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.

Component Versioning

You can create and maintain multiple versions of the same component using one of the following options:

- Define the module property `version` in your **_hw.tcl** file.

  For more information, refer to the *Component Interface Tcl Reference* chapter in volume 4 of the *Quartus II Handbook*

- If multiple versions of the component are defined in your component libraries, you can add a different the version of a component by right-clicking on the component and selecting **Add version** <*version_number*>.
- You can create an **.ipx** file in the same directory as your SOPC Builder project to control the search path for your project.

Classic Components in SOPC Builder

If you use classic components created with an version 7.2 of SOPC Builder or earlier, 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.

Document Revision History

Table 4–1 shows the revision history for this chapter.

Table 4–1. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	Added section on component versions.
November 2009	9.1.0	■ Added `ip-catalog` and `ip-make-ipx` commands. ■ Updated description of static components. Changed the term to describe components that use a generation callback to set the value of parameters from *dynamic component* to *generated component*. ■ Added description of composed components.
March 2009	9.0.0	Added 2 paragraphs introducing custom generations for dynamic components.
November 2008	8.1.0	■ 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.
May 2008	8.0.0	Added paragraph about IP Search Path.

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

Take an online survey to provide feedback about this handbook chapter.

5. Using SOPC Builder with the Quartus II Software

QII54023-10.0.0

This chapter describes the Quartus® II software features that are used with SOPC Builder, including the following:

- "Quartus II IP File"
- "Quartus II Incremental Compilation"
- "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:

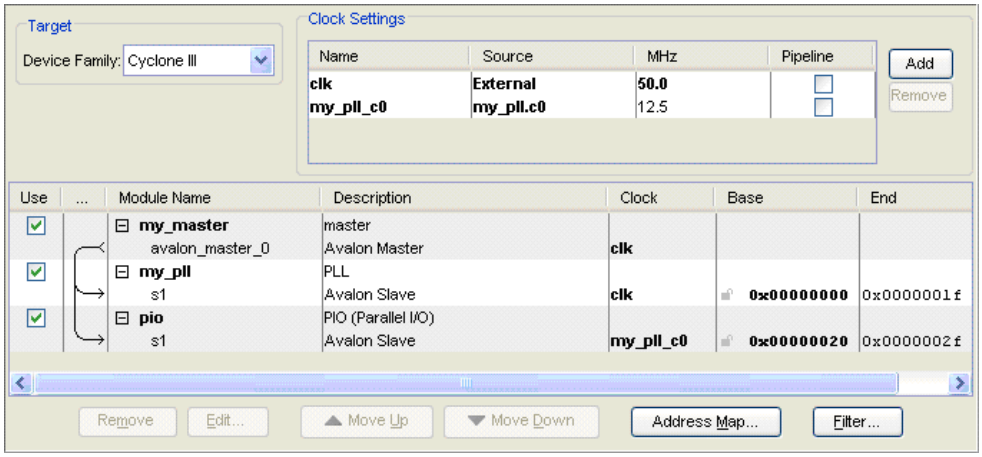
- Use the `derive_pll_clocks` command to derive clocks for all PLL outputs in the design. This is the best method.
- Use the `create_generated_clock` command to designate each clock output.
- Use the `-create_base_clocks` option of the `derive_pll_clock` assignments to designate the base clock feeding 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[*]}]
```

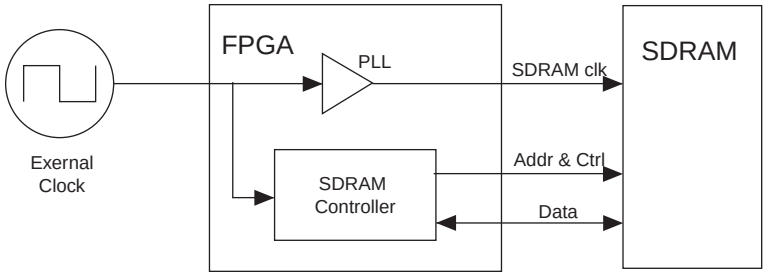
Because design contains a 4-bit PIO, `filter *_pio[*]` includes the following I/O pins.

- `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	^tAC (3)		5.5		5.5	ns	
	CL = 2	^tAC (2)		7.5		8	ns	
	CL = 1	^tAC (1)		17		17	ns	
Address hold time		^tAH	1		1		ns	
Address setup time		^tAS	1.5		2		ns	
CLK high-level width		^tCH	2.5		2.75		ns	
CLK low-level width		^tCL	2.5		2.75		ns	
Clock cycle time	CL = 3	^tCK (3)	6		7		ns	23
	CL = 2	^tCK (2)	10		10		ns	23
	CL = 1	^tCK (1)	20		20		ns	23
CKE hold time		^tCKH	1		1		ns	
CKE setup time		^tCKS	1.5		2		ns	
CS#, RAS#, CAS#, WE#, DQM hold time		^tCMH	1		1		ns	
CS#, RAS#, CAS#, WE#, DQM setup time		^tCMS	1.5		2		ns	
Data-in hold time		^tDH	1		1		ns	
Data-in setup time		^tDS	1.5		2		ns	
Data-out High-Z time	CL = 3	^tHZ (3)		5.5		5.5	ns	10
	CL = 2	^tHZ (2)		7.5		8	ns	10
	CL = 1	^tHZ (1)		17		17	ns	10
Data-out Low-Z time		^tLZ	1		1		ns	
Data-out hold time		^tOH	2		2.5		ns	

The mapping of external memory timing to FPGA I/O delays is shown in Table 5–1. This table 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 + 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.

Document Revision History

Table 5–2 shows the revision history for this chapter.

Table 5–2. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	No changes from previous release.
November 2009	9.1.0	Corrected `set_output_delay` expression for the SDRAM clock pin minimum in Example 5–2 on page 5–6.
March 2009	9.0.0	No changes to content from previous release.
November 2008	8.1.0	■ No changes to content from previous release. ■ Changed page size to 8.5 x 11 inches
May 2008	8.0.0	Initial release.

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

Take an online survey to provide feedback about this handbook chapter.

6. Component Editor

QII54005-10.0.0

The SOPC Builder 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.
- "HDL Parameters Tab" on page 6–6.
- "Saving a Component" on page 6–7.
- "Editing a Component" on page 6–8.
- "Component Parameterization" 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 Synopsis Design Constraint File (**.sdc)** for your component using the **Synth** option. Only files that you mark for **Synth** are added to the Quartus II IP File (**.qip)** 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

You can use the **Create HDL Template** to generate an HDL template for the component. Then, 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.

HDL Parameters Tab

You specify the parameters that users of your component can set to configure your component on the **HDL Parameters** tab. The **Parameters** table included on this tab displays Verilog HDL parameters or VHDL generics that you declared in the top-level HDL module. Using the **Parameters** table, you can specify the following information about each parameter:

- Default value
- Whether or not it is user-editable
- Type
- Group
- Tool tip

Click **Preview the Wizard** at any time to see how the component GUI appears.

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.

Refer to *Component Interface Tcl Reference* chapter in the *Quartus II Handbook* for detailed information about creating and displaying parameters using Tcl scripts.

Library Info

The **Library Info** tab allows you to specify the following information about your component:

- 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.
- Documentation—Allows you to specify multiple documents that pertain to your component. 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 documentation using the `add_documentation_link` Tcl command.

For more information refer to the `add_documentation_link` command in the *Component Tcl Interface Reference*.

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 assignments are added to the SOPC information file (**.sopcinfo)**, available for use for downstream components.

Component Parameterization

To edit component instance parameters, select a component in the **System Contents** tab of the SOPC Builder window and click **Edit**.

Document Revision History

Table 6–2 shows the revision history for this chapter.

Table 6–2. Document Revision History (Sheet 1 of 2)

Date	Version	Changes Made
July 2010	10.0.0	The **Component Wizard** tab was split into **HDL Parameters** and **Library Info** tabs.
November 2009	9.1.0	No changes from the previous release.
March 2009	9.0.0	■ 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.

Table 6–2. Document Revision History (Sheet 2 of 2)

Date	Version	Changes Made
November 2008	8.1.0	■ Added information about new HDL template feature ■ Changed page size to 8.5 x 11 inches
May 2008	8.0.0	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.

Take an online survey to provide feedback about this handbook chapter.

7. Component Interface Tcl Reference

QII54022-10.0.0

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"
- "Component Phases" on page 7–2
- "Writing a Hardware Component Description File" on page 7–2
- "Overriding Default Behaviors" on page 7–8
- "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.

Depending on your component design, your **_hw.tcl** file may be one of the following two types:

- Static—A static **_hw.tcl** file defines the top-level HDL file and associated component files. The HDL that describes a static component is created by the component author and is not changed by users of the component. HDL parameters are available when instantiating the component.
- Generated—A generated **_hw.tcl** file provides a user-defined program to generate the component's HDL. The HDL can be different for different parameterizations of the component.

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 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. The first command in a **_hw.tcl** file should specify the version of the **_hw.tcl** API to use, with the following Tcl command:

```
package require -exact sopc <version>
```

The version number is a Quartus II release version such as 10.0. SOPC builder guarantees that a valid **_hw.tcl** file that requests a particular `sopc package` behaves identically in future versions of the tool. Because of differences between versions of the Quartus II software, you cannot assume that an HDL file that works with one sopc package automatically works with other versions of the package.

This chapter describes the behavior of components that request the `sopc 10.0` package. For earlier releases, refer to the documentation for that release.

An excellent source of information about Tcl syntax is the Tcl Developer Xchange website.

Example 7–1. Basic Information for _hw.tcl File

```
# The package command must be the first command in the file
package require -exact sopc 10.0

# 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 users of your component to parameterize it in different ways. Each parameter has a number of properties such as its name, type, display name, and default value that can be used to control how the parameter is displayed and used. Example 7–2 illustrates the use of parameters that can be configured by 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}
```

Parameters can be divided into three types: user parameters, system information parameters and derived parameters. The following sections describe these parameter types.

User Parameters

User parameters are parameters that users have control over and that are exposed in the component GUI.

Derived Parameters

Derived parameters are parameters that are inferred by the component itself from user parameters or other derived parameters. For example, a clock period parameter can be derived from a data rate parameter.

SYSTEM_INFO Parameters

You can use `SYSTEM_INFO` parameter to request that certain parameter values are populated with information about the system. For example, you might want to know the frequency of the clock that ends up being connected to your clock input. When you declare `SYSTEM_INFO` properties, you provide an `<info-type>` and further arguments. The `<info-type>` is the type of information you want, such as `clock_rate`, and you use the additional arguments to specify things, such as which clock input interface you require. Example 7–3 illustrates the use of the `SYSTEM_INFO` parameter. For more information about the `SYSTEM_INFO` parameter properties refer to Table 7–5 on page 7–26

Example 7–3. Syntax of Tcl Command using the SYSTEM_INFO Parameter

```
set_parameter_property my_parameter SYSTEM_INFO {<info-type> [<arg>]}
```

Declaring Interfaces

To declare an interface, use the `add_interface` command. Then use the `set_interface_property` and `add_interface_port` commands to set its properties and indicate which signals belong to it. 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 are not associated with clocks (such as clock interfaces themselves), omit the associated clock interface, or use the word *asynchronous*. Example 7–4 illustrates interface declaration.

Example 7–4. 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 Synopsis Design Constraints files (**.sdc**). 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 7–5 illustrates the files that are typically required for generation and downstream tools.

Example 7–5. 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 SYNTHESIS

# Indicate which of the added HDL files holds the top-level module/entity
# 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 editor, validation, elaboration, and generation 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 7–8.

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 is displayed.

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

If the main program does not explicitly define the widths of all ports to constant values or to an expression, then SOPC Builder's default elaboration process calls quartus_map to determine the correct port widths. If you define all port widths in the main program, quartus_map is not called.

Automatic Port Widths

When port widths are not specified, or have a value of '-1', quartus_map is used to determine port widths as a function of the parameter set. While this process makes authoring a component easier, SOPC Builder can end up spending a lot of time calling quartus_map. When using automatic port widths, you can indicate that a certain parameter does not affect any port widths or interfaces by setting that parameter's affects_elaboration property to false, meaning that quartus_map is not called when the parameter's value is changed by your user. Be careful with this— indicating that a parameter does not affect elaboration when it really does can lead to problems that are mysterious and difficult to debug.

As an alternative to the automatic port widths, you can set port widths to simple HDL expressions using the width_expr property. width_expr is a string that holds an expression describing the port width. By using the width_expr property, you can define port widths as an expression that is evaluated without needing to analyze the HDL file or set them in an elaboration callback. The syntax for width expressions is the same as the HDL language that you use; however, only the addition, subtraction, multiplication, and division operators are allowed. For more complex port widths, the width of the port can be set as an arbitrary function of the component's parameters in an elaboration callback. The width expression is the last argument to the add_interface_port command. Example 7–6 illustrates the use of mathematical operators and the width_expr property.

Example 7–6. Defining Port Widths Using Simple Mathematical Operators

```
add_interface_port din din_data data input {WIDTH * SYMBOLS}
set_port_property din_data width_expr WIDTH
```

Parameterized Parameter Widths

For VHDL users, SOPC Builder allows a `std_logic_vector` parameter to have a width that is defined by another parameter. When adding a parameter of type `std_logic_vector` you can also specify its width as a parameter property. The width can be a constant or the name of another parameter. The commands below add a `std_logic_vector` parameter called `myParameter` whose width is set by another parameter, called `dataWidth`.

```
add_parameter myParameter STD_LOGIC_VECTOR
set_parameter_property myParameter WIDTH dataWidth
```

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.

Edit 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 7–23 lists the properties of parameters.

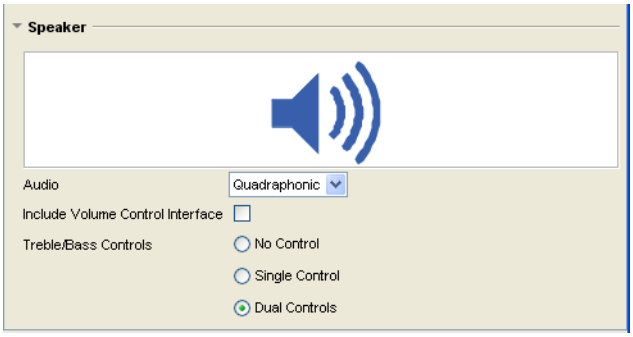
You can place parameters in logical groups and provide images and text to create a custom GUI for your component. Example 7–7 defines four parameters and illustrates the use of the `add_display_item` command and the `DISPLAY_HINT` and `ALLOWED_RANGES` parameters.

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 Speaker sound parameter
add_display_item Speaker volume_control parameter
add_display_item Speaker separate_control parameter

# 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 7–1 shows the GUI that the Tcl commands in Example 7–13 produces.

Figure 7–1. 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 SOPC Builder design file (**.sopc**). 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.
set_module_property VALIDATION_CALLBACK my_validation_callback

# Add the BAUDRATE_PRESCALE parameter, and indicate that it's derived
add_parameter BAUDRATE_PRESCALE int 600
set_parameter_property BAUDRATE_PRESCALE DERIVED true

# Add the PARITY parameter
add_parameter PARITY 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. All of the functionality available in the validation callback can also be used in the elaboration callback; separate callbacks for validation and elaboration are not required.

The elaboration callback will not be called when parameters with AFFECTS_ELABORATION=false are changed by the user of the component.

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
# 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

# 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 the status interface
  if { $use_status } {
    set_interface_property status_slave ENABLED true
  }

}
```

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_file` 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* or *.sv*> 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. If a static supporting file is only needed in some parameterizations of the component, you should add it from the main program and turn it on or off by setting its SYNTHESIS and SIMULATION properties appropriately from the elaboration callback.

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]

# Your callback needs to write $outdir$outputname.v here,
# perhaps by using exec to call an external program.

  # 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 replace 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 instead of displaying the parameterization GUI, 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_ENABLED ]

  # 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 7–21
- "Display Items" on page 7–29
- "Interfaces and Ports" on page 7–32
- "Generation" on page 7–39

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.

Starting with Quartus II software version 9.1, all Tcl commands that you can use in the validation callback are also available in the elaboration callback. With this change, you may be able to omit the custom validation callback by including some validation commands in your elaboration callback.

Table 7–2. Command Summary *(Note 1)* (Sheet 1 of 2)

Command	Full Description
Module Definition	
package <require> -exact sopc <*version*>	page 7–14
get_module_properties	page 7–15
get_module_property <*propertyName*>	page 7–16
set_module_property <*propertyName*> <*propertyValue*>	page 7–17
get_module_ports	page 7–17
get_module_assignments	page 7–17
get_module_assignment <*moduleName*>	page 7–18
set_module_assignment <*moduleName*> *[value]*	page 7–18
get_files	page 7–18
add_file filename *[<fileProperties> . . .]*	page 7–19
add_documentation_link <*docType*> <*title*> <*fileOrUrl*>	page 7–19
get_file_properties	page 7–19
get_file_property <*filename*> <*propertyName*>	page 7–20
set_file_property <*filename*> <*propertyName*> <*propertyValue*>	page 7–20
send_message <*messageLevel*> <*messageText*>	page 7–21
Parameters	
add_parameter <*parameterName*> <*parameterType*> *[<defaultValue> <description>]*	page 7–22
get_parameters	page 7–22
get_parameter_properties	page 7–22
get_parameter_property <*parameterName*> <*propertyName*>	page 7–27
set_parameter_property <*parameterName*> <*propertyName*> <*value*>	page 7–27
get_parameter_value <*parameterName*>	page 7–28
set_parameter_value <*parameterName*> <*value*>	page 7–28
decode_address_map <*address_map_XML_string*>	page 7–28
Display Items	
add_display_item <groupName> <id> <type> [<additionalInfo>]	page 7–29
get_display_items	page 7–31
get_display_item_properties	page 7–31
get_display_item_property <*itemName*> <*propertyName*>	page 7–31
set_display_item_property <*itemName*> <*propertyName*> <*value*>	page 7–31
Interfaces and Ports	
add_interface <*interfaceName*> <*interfaceType*> <*direction*> *[<associatedClock>]*	page 7–33
get_interfaces <*interfaceName*>	page 7–33
get_interface_property <*interfaceName*> <*propertyName*>	page 7–34
set_interface_property <*interfaceName*> <*propertyName*> <*value*>	page 7–35

Table 7–2. Command Summary *(Note 1)* (Sheet 2 of 2)

Command	Full Description
`add_interface_port <interfaceName> <portName> <portRole> [<direction> <width_expr>]`	page 7–35
`get_interface_ports [<interfaceName>]`	page 7–36
`get_port_properties`	page 7–36
`get_port_property <portName> <propertyName>`	page 7–37
`set_port_property <portName> <propertyName> [<value>]`	page 7–37
`get_interface_assignments`	page 7–38
`get_interface_assignment <interfaceName> <name>`	page 7–38
`set_interface_assignmet <interfaceName> <name> [<value>]`	page 7–38
Generation	
`get_generation_property <propertyName>`	page 7–40
`get_generation_properties`	page 7–39

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.

package

The `package` command allows you to specify a particular version of the SOPC Builder software to avoid software compatibility issues. You should use the `package` command at the beginning of your **_hw.tcl** file. When used, the component files behave as if they are interpreted by the version of the SOPC Builder software that you specify. When the `package` command is not used, version 9.0 of the SOPC Builder software is assumed. For components designed before 9.0, you can set the required package to 9.0. This document describes the behavior of component which start with `package require -exact sopc 10.0` For earlier releases, refer to the documentation for that release.

`package` is a standard Tcl command. For more information on this command refer to the following web page: **http://www.tcl.tk/man/tcl8.0/TclCmd/package.htm**

package		
Callback availability	Main (before any other commands in the file)	
Usage	`package require -exact sopc <version>`	
Returns	None	
Arguments	`version`	The version of SOPC Builder that you require, specified as decimal number
Example	`package require -exact sopc 10.0`	

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 (Sheet 1 of 2)

Property Name	Property Type	Can Be Set	Description
AUTHOR	String	Main program	The module's author.
DESCRIPTION	String	Main program	The description of the module, such as "Example SOPC Builder Module."
DISPLAY_NAME	String	Main program	The name to display when referencing the module, such as "My SOPC Component."
EDITABLE	Boolean	Main program	Indicates if the component is editable in the component editor.
EDITOR_CALLBACK	String	Main program	The name of the editor callback. The default parameterization UI is displayed if this property is not set.
ELABORATION_CALLBACK	String	Main program	The name of the elaboration callback. For static and generated components, the default elaborations used if this property is not set.
GENERATION_CALLBACK	String	Main program	The name of the generation callback.
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.
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.

Table 7–3. Module Properties (Sheet 2 of 2)

Property Name	Property Type	Can Be Set	Description
MODULE_DIRECTORY	String	Can only be read, not set	The directory containing the **_hw.tcl** file. All relative file names within the Tcl file are resolved relative to this directory. This directory is set as the current directory when running the main program or a callback.
MODULE_TCL_FILE	String	Can only be read, not set	The path to the **_hw.tcl** file.
NAME	String	Main program	The name of the module, such as my_sopc_component.
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.
VALIDATION_CALLBACK	String	Main program	The name of the validation callback. This callback is run in addition to the default validation.
VERSION	String	Main program	The module's version, such as 8.1.

The INSTANTIATE_IN_SYSTEM_MODULE, TOP_LEVEL_HDL_MODULE and GENERATION_CALLBACK commands are used to select the type of generation used by the component. You must set only one of these in the main program of your file.

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, boolean, or file	
Arguments	propertyName	One of the properties listed in Table 7–3 on page 7–15
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 7–15
	`propertyValue`	The new value of the property
Example	`set_module_property VERSION 10.0`	

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	`String`	
Arguments	None	
Example	`get_module_ports`	

get_module_assignments

This command returns names of the module assignment variables.

get_module_assignments	
Callback availability	Main, validation, elaboration, and compose
Usage	`get_module_assignments`
Returns	`String`
Arguments	None
Example	`get_module_assignments`

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.

get_module_assignment		
Callback availability	Main, validation, elaboration, and compose	
Usage	`get_module_assignment <name>`	
Returns	`String`	
Arguments	`name`	The name whose value is being retrieved
Example	`get_module_assignment embedded.sw.CMacro.colorSpace`	

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*.

set_module_assignment

This command sets the value of the specified argument.

set_module_assignment		
Callback availability	Main, validation, elaboration, and compose	
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`	

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]`

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	Main and generation	
Usage	`add_file filename [<fileProperties> . . . ]`	
Returns	`String`	
Arguments	`filename`	The file name to be added, relative to the directory containing the **_hw.tcl** file
	`fileProperties`	Files support the following 3 properties: ■ `SIMULATION`—File for simulation ■ `SYNTHESIS`—File for synthesis ■ `SDC`—TimeQuest constraints (SDC behaves like a synthesis file)
Example	`add_file my_component.v {SIMULATION SYNTHESIS}`	

add_documentation_link

This command allows you to add multiple documentation links for a single component.

add_documentation_link		
Callback availability	Main	
Usage	`add_documentation_link filename <docType> <title> <fileOrUrl>`	
Returns	None	
Arguments	`docType`	One of the following document types: `USER_GUIDE`, `RELEASE_NOTES`, `WEBLINK`, `ERRATA`, `DATASHEET`, `REFERENCE_MANUAL`, `WAVEFORM`, `SCHEMATICS`. `TUTORIAL`, `OTHER`
	`title`	The title of the document for use on menus and buttons.
	`fileOrUrl`	A path to the component documentation, using a syntax that provides the entire URL, not a relative path. For example: **http://www.mydomain.com/my_memory_controller.html** or **file:///datasheet.txt.**
Example	`add_documentation_link USER_GUIDE "Avalon Verification IP Suite User Guide"` `http://www.altera.com/literature/ug/ug_avalon_verification_ip.pdf`	

get_file_properties

This command returns the list of all properties that have been defined for a file.

get_file_properties	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	`get_file_properties`
Returns	`List of strings`
Arguments	None
Example	`get_file_properties`

get_file_property

This command returns the value of a single file property. The file name passed as an argument may 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	`Boolean`	
Arguments	`filename`	The file name whose properties are being retrieved
	`propertyName`	The file name 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	`Boolean`	
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. The message text is normally interpreted as HTML. The <*b*> element can be used to provide emphasis. If you do not want the message text to be interpreted as HTML then pass a list like `{ info text }` as the message level.

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 "<b>param1</b> 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.

add_parameter

This command adds a parameter to your component. Most of the parameter types are self-explanatory because they are used in the C programming language or HDL. However, the `string_list` and `integer_list` parameters that are used to create tables in GUIs require some explanation.

- When you use the `add_parameter` command with a `string_list` or `integer_list` parameter type, the parameter you define is displayed in a variable-sized table that includes **add** and **remove** buttons.
- If you define multiple parameters of type `string_list` or `integer_list`, you can also use the `add_display_item` command to specify that parameters should each be displayed as a column in a table, each parameter of type `string_list` or `integer_list` becomes a column in the table. Example 7–12 illustrates the use of the `integer_list` parameter types to create a multi-column table.

Example 7–12. Creating Tables Using the string_list and integer_list Parameter Types

```
add_parameter bitsWide INTEGER
add_parameter divider INTEGER
add_parameter coefficients INTEGER_LIST
add_parameter positions INTEGER_LIST
add_display_item myTable coefficients TABLE
add_display_item myTable positions TABLE
```

add_parameter		
Callback availability	Main program	
Usage	`add_parameter <parameterName> <parameterType> [<defaultValue> <description>]`	
Returns	`String`	
Arguments	`parameterName`	A name that you, the component author, choose for your parameter
	`parameterType`	The following types are supported: `Integer`, `Natural`, `Positive`, `Boolean`, `Std_logic`, `Std_logic_vector`, `String`, `String_list`, and `Integer_list`.
	`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_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 strings`
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 (Sheet 1 of 3)

Property Name	Type/ Default	Can Be Set	Description
AFFECTS_ELABORATION *(1)*	Boolean, refer to DESCRIPTION	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.
AFFECTS_GENERATION	Boolean, refer to DESCRIPTION	Main program	The default value of AFFECTS_GENERATION is false if you provide a top-level HDL module, it is true if you provide a custom generation callback. Set AFFECTS_GENERATION to false if the value of a parameter does not change the results of system generation.
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. You can also assign longer strings to be displayed in the GUI to string variables. For example, ALLOWED_RANGES {"dev1:Cyclone IV GX" "dev2:Stratix V GT"}Refer to Example 7–7 on page 7–8 and Figure 7–1 on page 7–8 for additional examples illustrating the use of this property.
DERIVED	Boolean/ false	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.
DESCRIPTION	String, ""	Main program	A user-visible description of the parameter.

Table 7–4. Parameter Properties (Sheet 2 of 3)

Property Name	Type/ Default	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. ■ fixed_size—for string_list and integer_list parameters, the fixed_size DISPLAY_HINT eliminates the **add** and **remove** buttons from tables. Refer to Example 7–7 on page 7–8 and Figure 7–1 on page 7–8 for examples illustrating the use of this property.
DISPLAY_NAME	String, ""	Main program	This is the GUI label that appears to the left of the parameter.
DISPLAY_UNITS	String, ""	Main program	This is the GUI label that appears to the right of the parameter.
ENABLED	Boolean, true	Main program, validation, and elaboration, callbacks	When false, the parameter is disabled, meaning that it is displayed, but greyed out, indicating that it is not editable on the parameterization GUI.
HDL_PARAMETER	Boolean, false	Main program	When true, the parameter must be passed to the HDL component description. The default value is false.
NEW_INSTANCE_VALUE	String, ""	Main program	This property allows you to change the default value of a parameter without affecting older components that have assigned a default value to this parameter using the defaultValue argument. The practical result is that older components will continue to use defaultValue for the parameter and newer components can use the value assigned by NEW_INSTANCE_VALUE.

Table 7–4. Parameter Properties (Sheet 3 of 3)

Property Name	Type/ Default	Can Be Set	Description
SYSTEM_INFO	String, ""	Main program	Allows you to assign information about the instantiating system to a parameter that you define. SYSTEM_INFO requires a keyword 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
UNITS	String, ""	Main program	Sets the units of the parameter. The following values are possible: picoseconds, nanoseconds, microseconds, milliseconds, seconds, hertz, kilohertz, megahertz, gigahertz, address, bits, bytes, kilobytes, megabytes, gigabytes, bitspersecond, kilobitspersecond, megabitspersecond, gigabitspersecond. For example, `set_parameter_property frequency UNITS gigahertz`
VISIBLE	Boolean, true	Main program, validation, and elaboration, callbacks	Indicates whether or not to display the parameter in the parameterization GUI.

Note to Table 7–4:

(1) The AFFECTS_ELABORATION property was called AFFECTS_PORT_WIDTHS before version 9.0 of the Quartus II software.

Table 7–5 lists the properties that you can use with the `system_info` parameter property.

Table 7–5. SYSTEM_INFO Properties (Sheet 1 of 2)

Property	Type	Description
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>}`
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>}`
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>}`
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>}`
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}`
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>}`

Table 7–5. SYSTEM_INFO Properties (Sheet 2 of 2)

Property	Type	Description
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>}`
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>}`

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`, `boolean`, or `units` depending on property refer to Table 7–4 on page 7–23	
Arguments	`parameterName`	The name of the parameter whose property value is being retrieved
	`propertyName`	One of the properties listed in Table 7–4 on page 7–23
Example	`get_parameter_property parameter1 GROUP`	

set_parameter_property

This command sets a single parameter property.

set_parameter_property		
Callback availability	Main, validation, and elaboration	
Usage	`set_parameter_property <parameterName> <propertyName> <value>`	
Returns	`string`, `boolean`, or `units` depending on property	
Arguments	`parameterName`	Specifies the parameter that is being set
	`propertyName`	Specifies the property of `parameterName` that is being set, refer to Table 7–4 on page 7–23 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)*, compose. generation, and editor	
Usage	`get_parameter_value <parameterName>`	
Returns	`String`	
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. If `affects_generation=false` then it is not available from the generation callback.

set_parameter_value

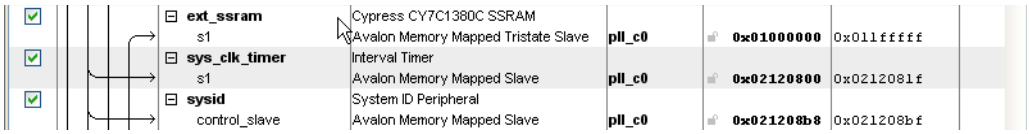
This command sets a parameter value. The values of derived parameters can be set from the validation and elaboration callbacks. The values of parameters which are not marked as `derived` or `system_info` can be set from the editor callback.

set_parameter_value		
Callback availability	Validation, elaboration, 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. The XML code describing each slave includes: its name, start address, and end address + l. Figure 7–2 shows a portion of an SOPC Builder system with three Avalon-MM slave devices.

Figure 7–2. SOPC Builder System with Three Avalon-MM Slaves



Example 7–13 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. Using `decode_address_map` 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.

Altera recommends that you use the code provided in the description of Example 7–13 to enumerate over the components within an address map, rather than writing your own parser.

Example 7–13. 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>
```

decode_address_map		
Callback availability	Validation, compose. 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.

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)"`
	`}`	

Display Items

You specify your component GUI using the display commands.

add_display_item

You can use this command to specify the following 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 the display items for them in the **_hw.tcl** file.
- You can create multi-column tables to present a component's parameters. Refer to Example 7–12 on page 7–21 for an example that illustrates multi-column tables.

- You can specify an image to provide a pictorial representation of a parameter or parameter group.
- You can create a button by adding a display item of type `action`. The display item includes the name of the callback to run when the action is performed.

You create a display group by adding display items to it.

add_display_item		
Callback availability	Main program	
Usage	`add_display_item <groupName> <id> <type> [<additionalInfo>]`	
Returns	`String`	
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. The following types are defined: ■ `icon`–a **.gif**, **.jpg**, or **.png** file ■ `parameter`–a parameter in the instance ■ `text`–a block of text ■ `group`–a group. if the `groupName` is also defined, the new group is a child of the `groupName` group. if `groupName` is an empty string, the group is top-level. ■ `action`–an action defined by a callback procedure when you click the button labeled by *`actionName`*.
	`additionalInfo`	Provides extra information required for display items. The following examples illustrate how you use the `additionalInfo` argument for the various types: ■ `add_display_item` *`groupName id`* `icon` *`path-to-image-file`* ■ `add_display_item` *`groupName parameterName`* `parameter` (`additionalInfo` not required) ■ `add_display_item groupName id text "your-text"` The `your-text` argument is a block of text that is displayed in the GUI. Some simple HTML formatting is allowed, such as <b> and <i>, if the text starts with `"html>"`. ■ `add_display_item` *`parentGroupName childGroupName`* `group [tab]` The `tab` is an optional parameter. If present, the group appears in separate tab in the GUI for the instance. ■ `add_display_item` *`parentGroupName actionName`* `action buttonClickCallbackProc`
Examples	`add_display_item timing read_latency parameter` `add_display_item sound speaker icon speaker.jpg`	

get_display_items

This command returns a list of all items to be displayed as part of the parameterization GUI.

get_display_items	
Callback availability	Main, elaboration, generation, and editor
Usage	`get_display_items`
Returns	`List of strings`
Arguments	`None`
Example	`get_display_items`

get_display_item_properties

This command returns a list of names of the properties of display items that are part of the parameterization GUI.

get_display_item_properties	
Callback availability	Main
Usage	`get_display_item_properties`
Returns	`List of strings`
Arguments	`None`
Example	`get_display_item_properties`

get_display_item_property

This command returns the value of specific property of a display item that is part of the parameterization GUI.

get_display_item_property		
Callback availability	Main	
Usage	`get_display_item_property <itemName> <propertyName>`	
Returns	`String`	
Arguments	`itemName`	The item whose property value is being retrieved
	`propertyName`	The property whose value is being retrieved
Example	`set my_label [ get_display_item_property my_action DISPLAY_NAME ]`	

set_display_item_property

This command sets the value of specific property of a display item that is part of the parameterization GUI.

set_display_item_property		
Callback availability	Main	
Usage	`set_display_item_property <itemName> <propertyName> <value>`	
Returns	`String`	
Arguments	`itemName`	The item whose property value is being set
	`propertyName`	The property whose value is being set
	`value`	The value to set
Example	`set_display_item_property my_action DISPLAY_NAME “Click Me”` `set_display_item_property my_action DESCRIPTION “clicking this button runs the click_me_callback proc in the hw.tcl file”`	

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. The common properties, `ENABLED` and `ASSOCIATED_CLOCK` apply to all interface types. Refer to the *Avalon Interface Specifications* for a description of other properties.

add_interface		
Callback availability	Main program, and elaboration	
Usage	`add_interface <interfaceName> <interfaceType> <direction> [<associatedClock>]` *(1)*	
Returns	`String`	
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` — `end`
		`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 a 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, validation, elaborations, and editor	
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. Refer to the *Avalon Interface Specifications* for more information about interface properties.

The interface properties that are common to all interface types are listed below in Table 7–6.

Table 7–6. Interface Properties Common to All Interface Types

Property	Type	Description
ASSOCIATED_CLOCK	String	The name of the clock interface that this interface is synchronous to.
ENABLED	Boolean	Specifies whether or not interface is enabled.

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	`string`, `boolean`, or `units` depending on property Refer to the *Avalon Interface Specifications* for more information about interface properties	
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. This property is either ENABLED or ASSOCIATED_CLOCK or a property name defined by the interface.
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 and elaboration	
Usage	`set_interface_property <interfaceName> <propertyName> <value>`	
Returns	`String`	
Arguments	`interfaceName`	The name of an interface that includes this property
	`propertyName`	The name of the property whose value you want to set, which is `ENABLED` or `ASSOCIATED_CLK` or a name from the *Avalon Interface Specifications.*
	`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 width and direction must be set by the end of the elaboration phase. The port width can be set with one of the following mechanisms:

- A constant width or a width expression can be set in the main program
- A constant width can be set in the elaboration callback

☞ Without an elaboration callback, for static components `quartus_map` determines the port width from the HDL

add_interface_port		
Callback availability	Main program and elaboration	
Usage	`add_interface_port <interfaceName> <portName> <portRole> [<direction> <width_expr>]`	
Returns	`String`	
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 Interface Specification*. Refer to the *Avalon Interface Specifications* for the `signal types` available for each interface type.
	`direction`	The `direction` can be `input`, `output`, or `bidir`
	`width_expr`	The port's width expression. In simple cases, this is just 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	`String`	
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.

get_port_properties		
Callback availability	Main, validation, elaboration, generation, and editor	
Usage	`get_port_properties <portName>`	
Returns	`String`, `boolean`, or `units` depending on property refer to Table 7–4 on page 7–23	
Arguments	`portName`	The name of the port whose properties are required. The following 7 port properties are supported: ■ `direction` ■ `termination` ■ `termination_value` ■ `vhdl_type` ■ `width` ■ `width_expr` Refer to Table 7–7 for a description of these properties.
Example	`get_port_properties mm_slave`	

Table 7–7 describes the available port properties

Table 7–7. Port Properties (Sheet 1 of 2)

Name	Type	Description
`direction`	`input, output, bidir`	The direction of the port from the component's perspective.
`termination`	`boolean`	When `true`, instead of connecting the port to the SOPC Builder system, it is left unconnected for `OUTPUT` and `BIDIR` or set to a fixed value for `INPUT`. Has no effect for components that implement a generation callback instead of using the default wrapper generation.
`termination_value`	`integer`	The constant value to drive an input port.

Table 7–7. Port Properties (Sheet 2 of 2)

Name	Type	Description

`vhdl_type`	`std_logic`
	`std_logic_vector`
	`auto`
		indicates the type of a VHDL port. The default value, `auto`, selects `std_logic` if the width is fixed at 1, and `std_logic_vector` otherwise.

`width`	`integer`	The width of the port in bits.
`width_expr`	`string`	The width expression of a port. Setting the `width` and `width_expr` properties have the same effect; they both update the effective width expression. The `width`/`width_expr` properties can be set to an `integer` at any time. They can only be set to arithmetic expressions in the main program. The values of the `width` and `width_expr` properties behave differently when `get_port_property` is used. `width` always returns the current integer width of the port. `width_expr` always returns the unevaluated width expression.

get_port_property

This command returns the value of single port property for the specified port.

get_port_property		
Callback availability	Main, validation, elaboration, generation, and editor	
Usage	`get_port_property <portName> <propertyName>`	
Returns	Depends on the type of the property	
Arguments	`portName`	The name of the port
	`propertyName`	One of the supported properties described in Table 7–7.
Example	`get_port_property rdata WIDTH`	

set_port_property

This command sets a single port property.

set_port_property		
Callback availability	Main program, elaboration, and generation	
Usage	`set_port_property <portName> <propertyName> [<value>]`	
Returns	`String`, `boolean`, or `units` depending on property refer to Table 7–4 on page 7–23	
Arguments	`portName`	The name of the port
	`propertyName`	One of the supported properties described in Table 7–7.
	`value`	The value to set
Example	`set_port_property rdata WIDTH 32`	

get_interface_assignments

This command returns the value of all interface assignments for the specified interface.

get_interface_assignments		
Callback availability	Main, validation, and elaboration	
Usage	`get_interface_assignments <interfaceName>`	
Returns	`String`	
Arguments	`interfaceName`	The name of the Avalon interface whose assignment is being retrieved
Example	`get_interface_assignments s1`	

get_interface_assignment

This command returns the value of the specified name for the specified interface.

get_interface_assignment		
Callback availability	Main, validation, and elaboration	
Usage	`get_interface_assignments <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`	

set_interface_assignment

This command sets the value of the specified assignment for the specified interface.

set_interface_assignment		
Callback availability	Main, validation, and elaboration	
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`	

For more information about the use of the `set_interface_assignment` command, refer to the "*Publishing Component Information to Embedded Software*" chapter in the *Nios II Software Developer's Handbook: Studio Edition*.

Generation

This section covers the commands that 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. Generation properties are provided to the generation callback to support per-instance HDL generation.

get_generation_properties	
Callback availability	Main, validation, elaboration, generation, and editor
Usage	`get_generation_properties`
Returns	`String`. The following generation properties are supported: ■ `hdl_language` ■ `output_directory` ■ `output_name` Refer to Table 7–8 for a description of the generation properties.
Arguments	None
Example	`get_generation_properties`

Table 7–8 describes the generation properties.

Table 7–8. Generation Properties

Name	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`	`OUTPUT_NAME` is `module_0` and the `HDL_LANGUAGE` is `verilog`, the file **module_0.v** or **module_0.sv** _must be generated and must contain the module, `module_0`.

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`, `boolean`, or `units` depending on property refer to Table 7–4 on page 7–23	
Arguments	`propertyName`	One of the 3 generation properties: ■ `HDL_LANGUAGE` ■ `OUTPUT_DIRECTORY` ■ `OUTPUT_NAME`
Example	`get_generation_property OUTPUT_DIRECTORY`	

Deprecated Commands and Properties

Table 7–9 lists commands and properties that were available in previous versions of the Quartus II software and the command that have replaced them.

Table 7–9. Deprecated Commands and Properties

Deprecated Command	Replacement
`add_clock_interface <name>`	`add_interface <name> clock input`
`add_port_to_clock_interface <name> <role> <interface>`	`add_interface_port <interface> <name> <role> 1`
`add_port_to_interface <interface> <name> <role>`	`add_interface_port <interface> <name> <role> <input> 1`
`get_generation_setting <property>`	`get_generation_property <property>`
`get_list_of_ports <direction>`	`get_interface_ports`
`set_module <name>`	`set_module_property name <name>`
`set_module_description <description>`	`set_module_property description <description>`
`set_port_direction_and_width <name> <direction> <width>`	`set_port_property <name> direction <direction>`; `set_port_property <name> width <width>`
`set_source_file <file>`	`set_module_property top_level_hdl_file <file>`

`get_project_property`	To determine device family, use the following commands: In the main program:
	`add_parameter DEVICE_FAMILY string "unknown"`
	`set_parameter_property DEVICE_FAMILY SYSTEM_INFO device_family`
	In the validation, generation callback read it using the following command:
	`get_parameter_value DEVICE_FAMILY`

Deprecated Module Properties	
`datasheet_url`	`use: add_documentation_link command.`
`libraries`	`Unnecessary`
`class_name`	`name`
`module_file_name`	`top_level_hdl_file`
`preview_<n>_callback`	`<n>_callback`
Deprecated Parameter Properties	
`affects_port_widths`	`affects_elaboration`
`GROUP`	Use `add_display_item` instead
Deprecated Generation Properties	
`language`	`hdl_language`

Document Revision History

Table 7–10 shows the revision history for this chapter.

Table 7–10. Document Revision History (Sheet 1 of 2)

Date	Version	Changes Made
July 2010	10.0.0	■ Added the following new commands: set_display_item_property, get_display_item_property, and add_documentation_link ■ Added ability to create multi-column tables in the component GUI using `string_list` and `integer_list` parameter types. ■ Added a new display item of type action to create buttons that carry out an action defined by a callback procedure. ■ Added `NEW_INSTANCE_VALUE` parameter property. ■ Deprecated the `DATASHEET_URL` module property.
November 2009	9.1.0	■ Tcl interpreter now stops upon error rather than continuing to interpret commands. ■ Changes `validation` and `elaboration` callbacks to be more similar. All commands permitted in the `validate` callback are also permitted in the `validation` callback. ■ Added the `package` command. ■ Added ability to optimize a component to wires or constant values using the `driven_by` argument. ■ Added parameter to set port width using an expression. ■ Changed syntax for setting the width of a `std_logic_vector`. ■ Renamed `TERMINATION_WIDE TERMINATION_VALUE` in the "get_port_properties" on page 7–36. ■ Corrected Example 7–11 on page 7–12. `set result = [...]` should be `set result [...]` ■ Added Table 7–9 listing deprecated commands and properties and their replacements.
March 2009	9.0.0	■ Added `add_display_item` commands. ■ Added `DISPLAY_HINT`, `IS_HDL_PARAMETER`, `DERIVED`, and `SYSTEM_INFO` parameters to Table 7–4 on page 7–23. 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.
March 2009 (continued)		■ Explained that before the elaboration phase, parameters may have values of 0 or -1 that are determined during HDL analysis.

Table 7–10. Document Revision History (Sheet 2 of 2)

Date	Version	Changes Made
November 2008	8.1.0	■ 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
June 2008	8.0.1	■ Reformatted command information in tables.
May 2008	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.

Take an online survey to provide feedback about this handbook chapter.

8. Archiving SOPC Builder Projects

QII54017-10.0.0

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 design file (**.sopc**)	*<sopc_builder_system>*.**sopc**	Yes
SOPC Builder classic system description for generation, a peripheral template file (**.ptf**) *(1)*	*<sopc_builder_system>*.**ptf**	Yes
SOPC Builder report file, an SOPC information file (**.sopcinfo**)	*<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 (**.qpf**)	*<project_name>*.**qpf**	No
Quartus II settings file (**.qsf**)	*<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.

Document Revision History

Table 8–2 shows the revision history for this chapter.

Table 8–2. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	No change from previous release.
November 2009	9.1.0	No change from previous release.
March 2009	9.0.0	No change from previous release.
November 2008	8.1.0	Changed page size to 8.5” × 11”.
May 2008	8.0.0	Renumbering from Chapter 7 to 8.

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

Take an online survey to provide feedback about this handbook chapter.

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

QII54006-10.0.0

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

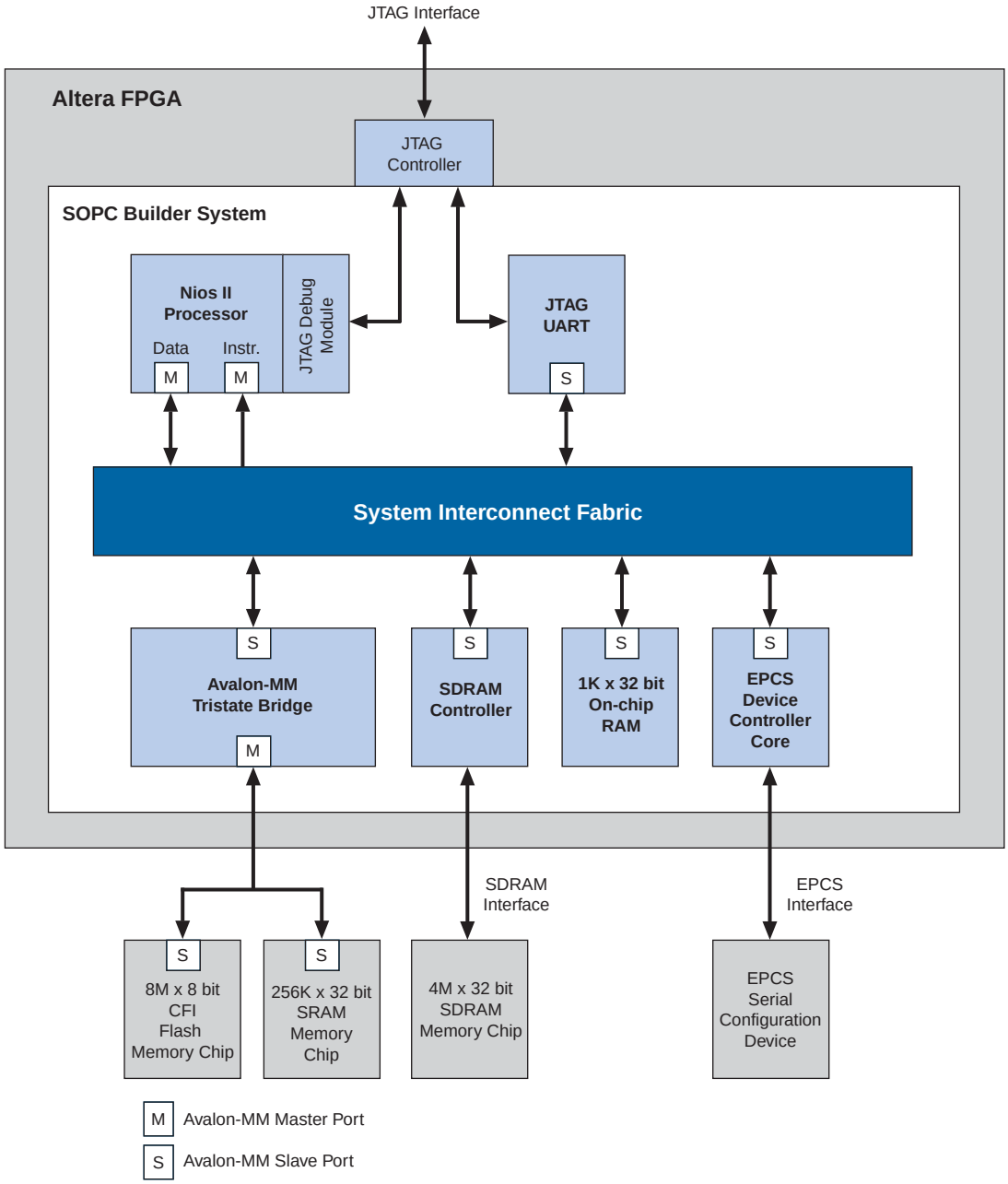
This chapter demonstrates the process for building a system that contains one of each type of memory as shown in Figure 9–1. Each section of the chapter builds on previous sections, culminating in a complete system.

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.

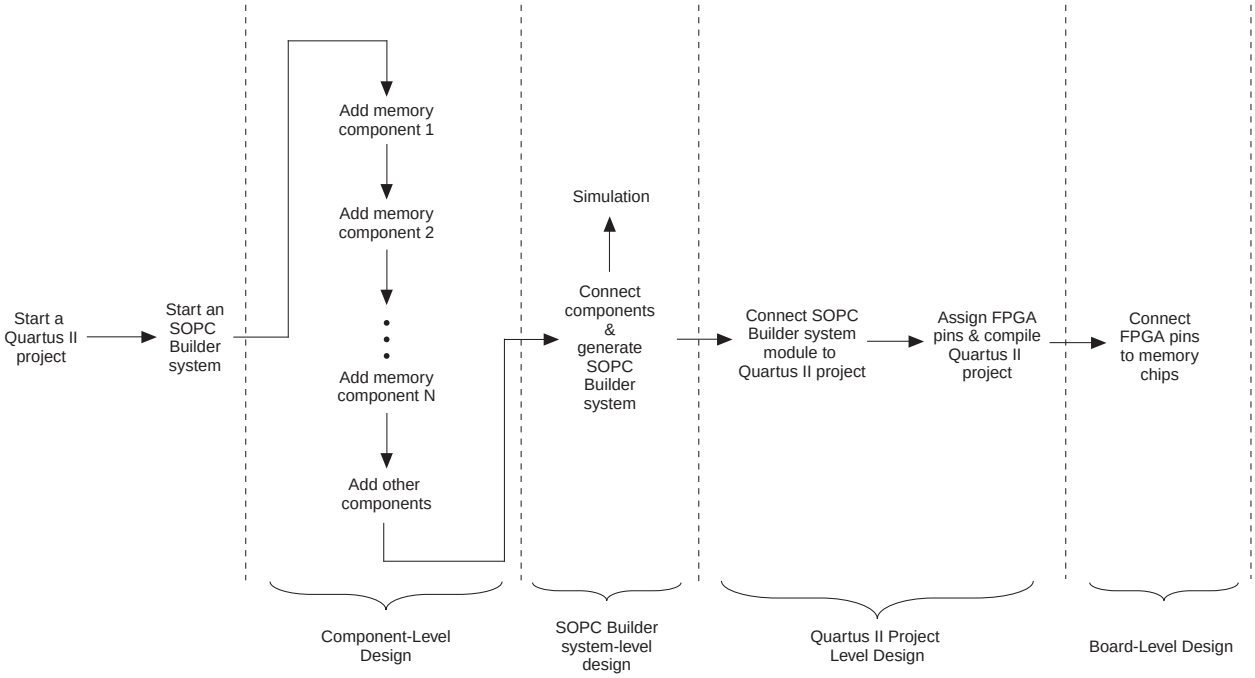
This chapter does not describe downloading and verifying a working system in hardware. Therefore, there are no hardware requirements for the completion of this chapter. However, the example memory subsystem has been tested in hardware.

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.

  CAUTION: 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.

  CAUTION: 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

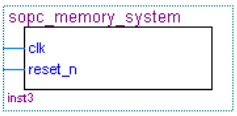
The on-chip memory is embedded inside the SOPC Builder system, and there is nothing to connect at the board level.

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 the *Embedded Peripherals IP User Guide*.

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

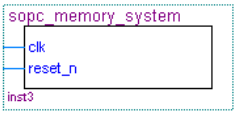
You must connect the EPCS device to the FPGA as described in the Altera *Configuration Handbook*. No other connections are necessary.

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 the *Embedded Peripherals IP User Guide*.

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 the *Embedded Peripherals IP User Guide*.

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 the *Embedded Peripherals IP User Guide*.

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

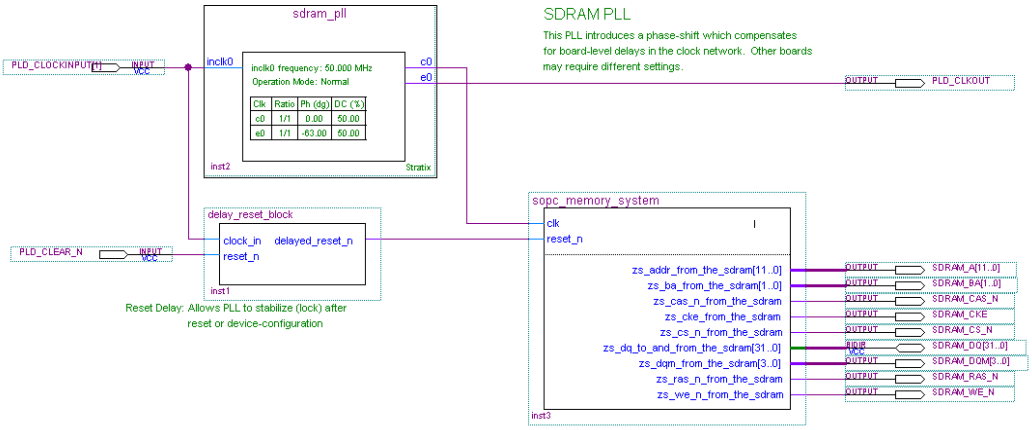
Memory requirements largely dictate the board-level configuration of the SDRAM device or devices. The SDRAM controller core can accommodate various configurations of SDRAM on the board, including multiple banks and multiple devices.

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 the *Embedded Peripherals IP User Guide*.

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 9–20.

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.

CAUTION

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

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
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
...	...	...	...	...	...

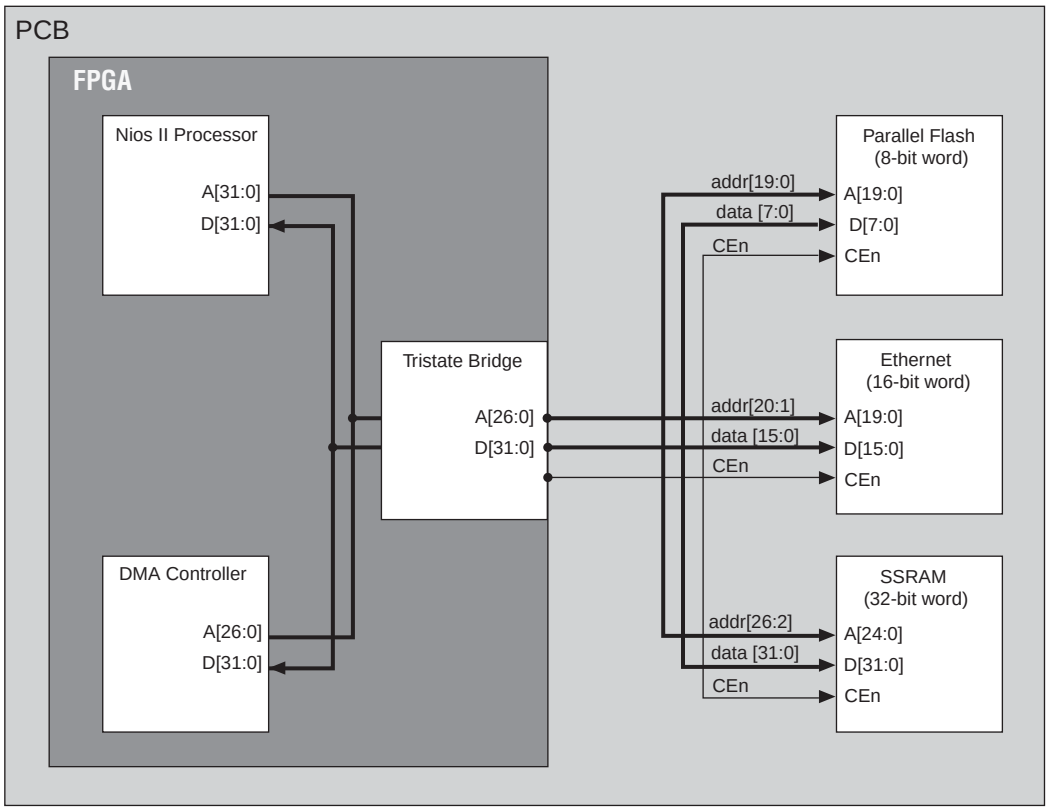
CAUTION

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 Different Address Widths and 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

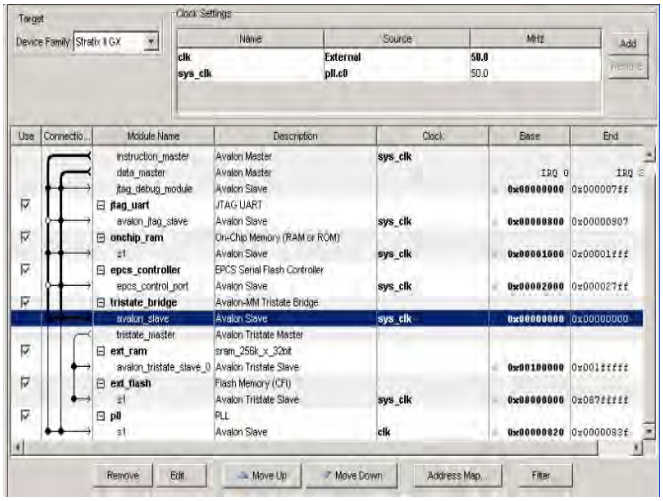
Adding the SRAM Interface

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

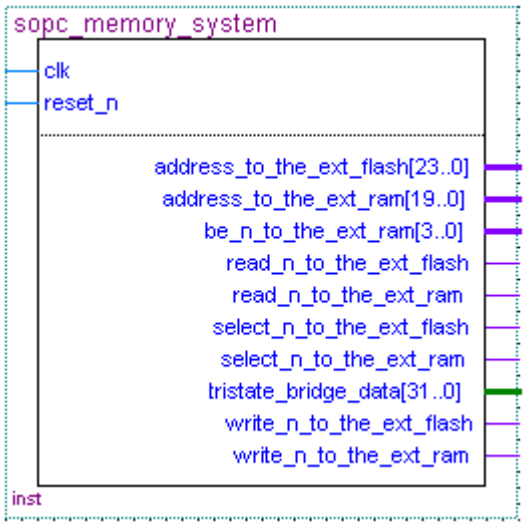


Figure 9–10 shows the pin assignments in the Quartus II Assignment Editor for some of the SRAM and flash memory pins. The correct pin assignments depend on the target board.

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

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
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

Document Revision History

Table 9–4 shows the revision history for this chapter.

Table 9–4. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	No changes from previous release.
November 2009	9.1.0	No changes from previous release.
March 2009	9.0.0	Minor updates to clarify text.
November 2008	8.1.1	■ Removed private comments
November 2008	8.1.0	■ 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	8.0.0	■ 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.

Take an online survey to provide feedback about this handbook chapter.

10. SOPC Builder Component Development Walkthrough

QII54007-10.0.0

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 10–6. This section shows you how to use components in other systems, or share them with other designers.
- "System Information Files (.sopcinfo)" 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);
```

   SOPC Builder only supports the VHDL port types `std_logic` and `std_logic_vector`.

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

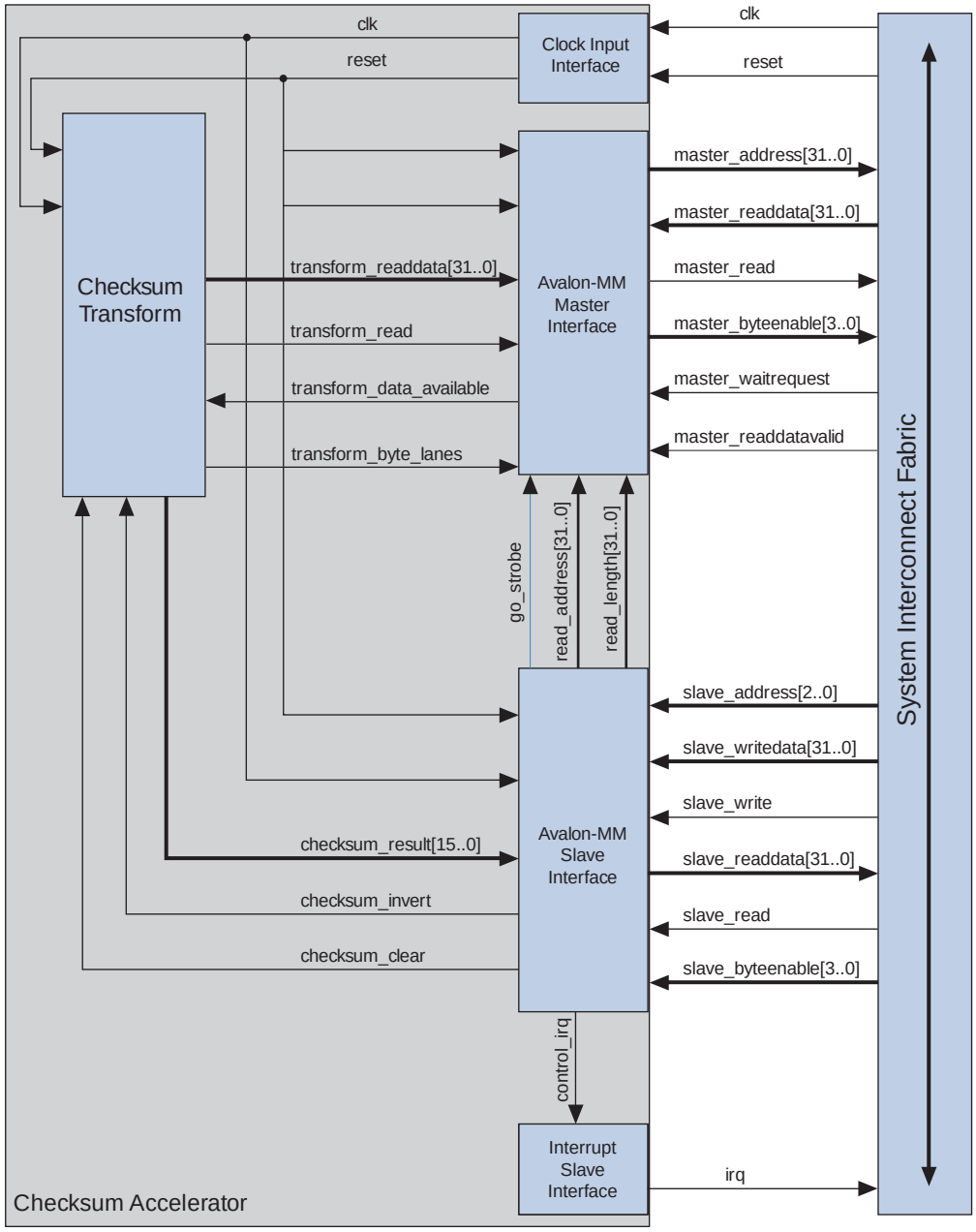
Altera has provided a checksum hardware accelerator design example to demonstrate the steps to create a component and instantiate it in a system. This design example is available for download from the Altera literature website. Included in the compressed download file is a **readme.pdf** that describes 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.

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.

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*.

System Information Files (.sopcinfo)

Every time SOPC Builder generates a system, a <*mysystem*>**.sopcinfo** 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.

Document Revision History

Table 10–1 shows the revision history for this chapter.

Table 10–1. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	No changes from previous release.
November 2009	9.1.0	■ Added statement that SOPC Builder only supports the VHDL `std_logic` and `std_logic_vector` port types.
March 2009	9.0.0	Corrected direction of transform_data_available and transform_byte_lanes signals in Figure 10–1 on page 10–5.
November 2008	8.1.0	■ 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.
May 2008	8.0.0	■ 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.

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

Take an online survey to provide feedback about this handbook chapter.

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

QII54020-10.0.0

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:

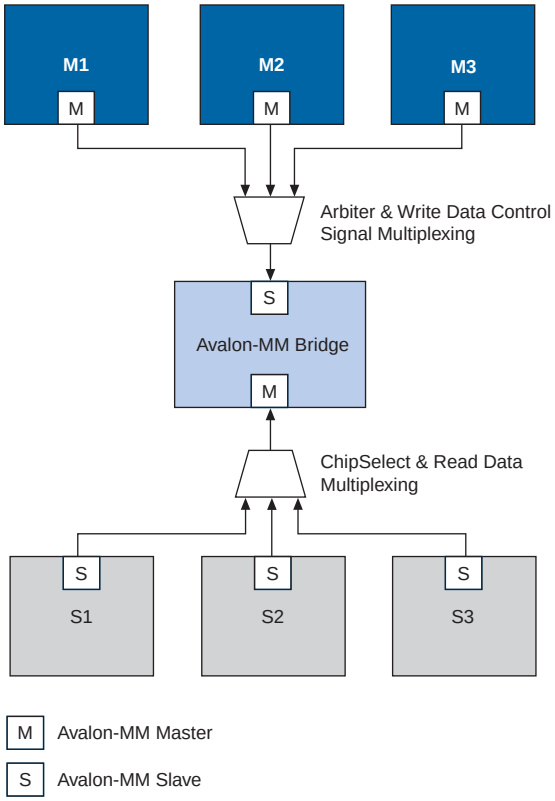
- "Avalon-MM Pipeline Bridge" on page 11–7
- "Clock Crossing Bridge" on page 11–10
- "Avalon-MM DDR Memory Half-Rate Bridge" on page 11–18

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

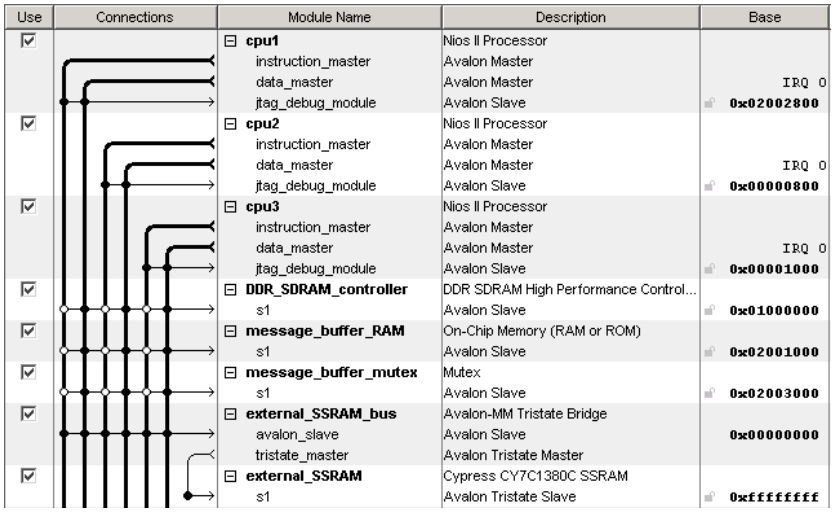


Figure 11–3 illustrates the default system interconnect fabric for the system in Figure 11–2.

Figure 11–3. Example System without Bridges—System Interconnect View

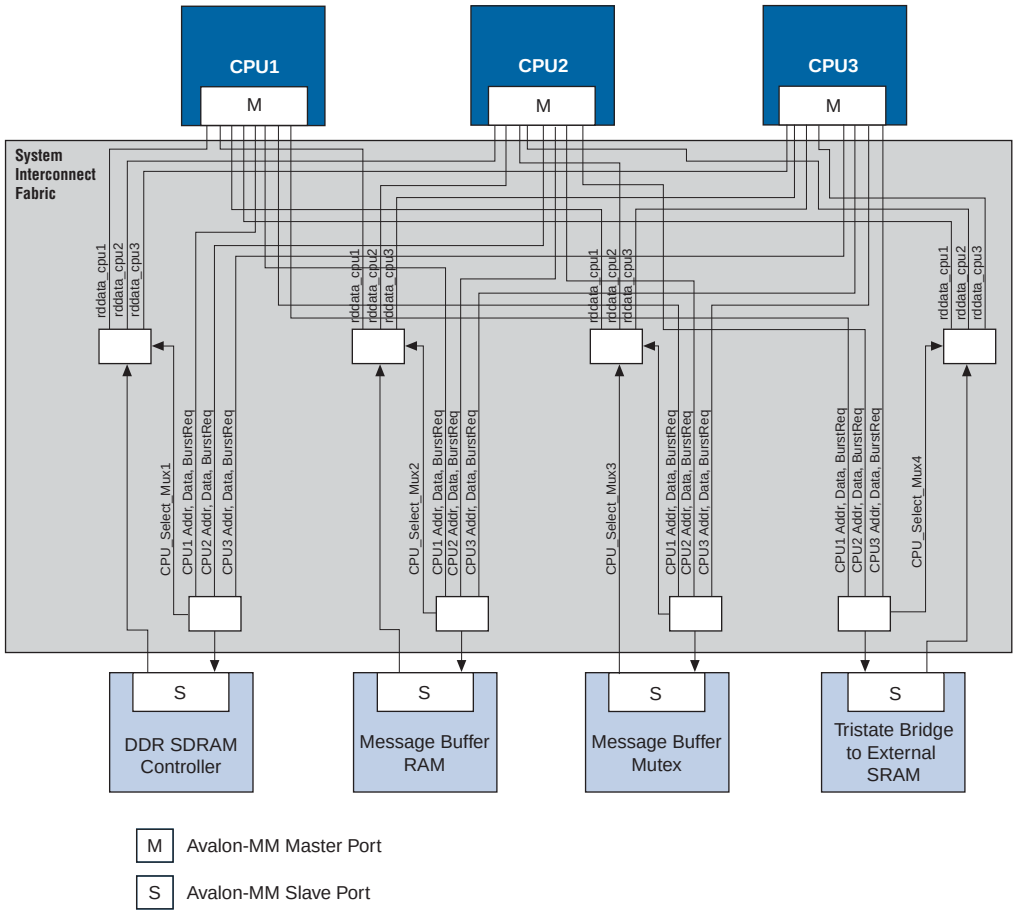


Figure 11–4 and Figure 11–5 show how inserting bridges can affect the generated logic. For example, if the DDR SDRAM controller can run at 166 MHz and the CPUs accessing it can run at 120 MHz, inserting an Avalon-MM clock-crossing bridge between the CPUs and the DDR SDRAM has the following benefits:

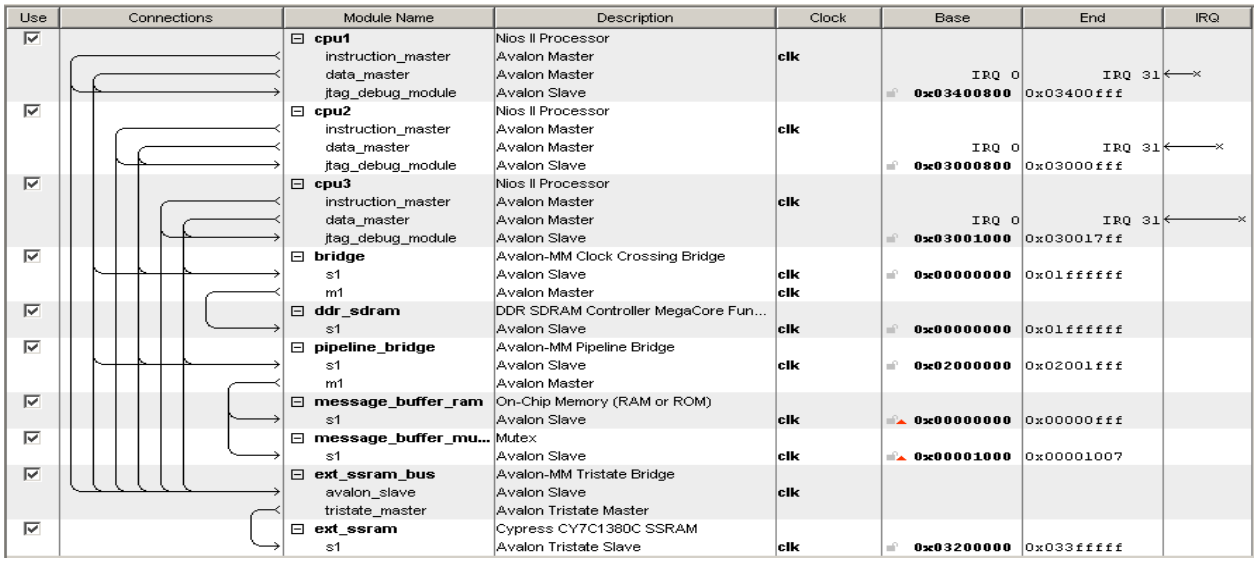
- Allows the CPU and DDR interfaces to run at different frequencies.
- Places system interconnect fabric for the arbitration logic and multiplexer for the DDR SDRAM controller in the slower clock domain.
- Reduces the complexity of the interconnect logic in the faster domain, allowing the system to achieve 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.

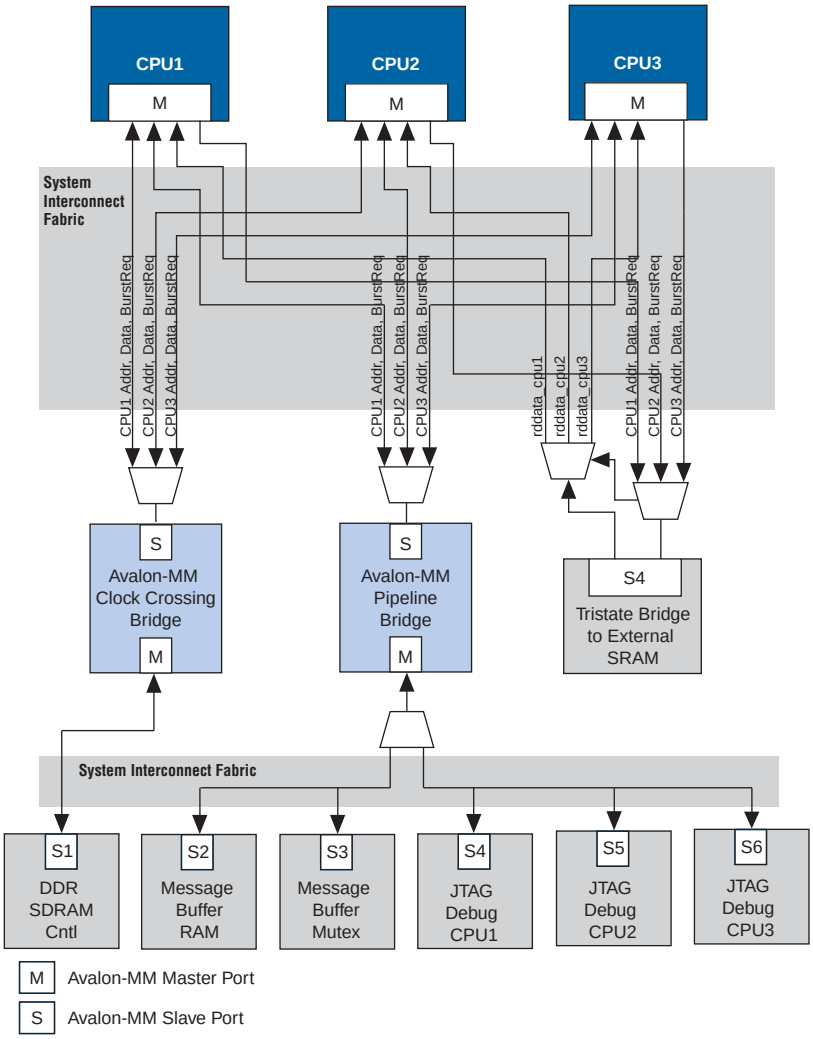
Figure 11–4. Example SOPC System with Bridges—SOPC Builder View



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).

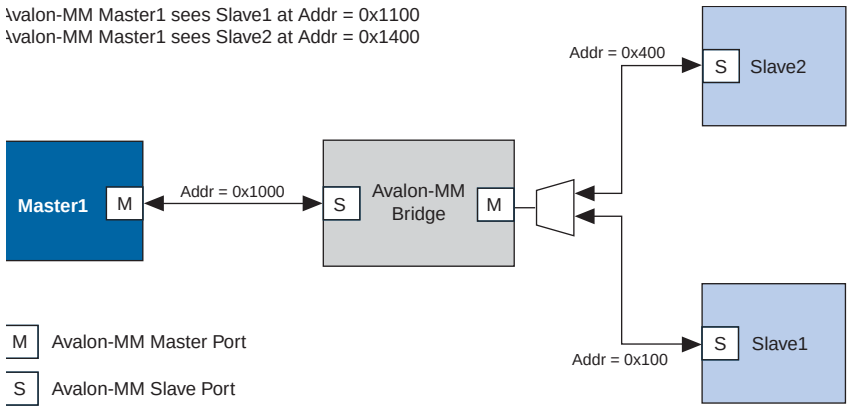
Equation 11–1.

$$range = [base_address \; .. \; (base_address + (span - 1))];$$

SOPC Builder follows several rules in constructing an address map for a system with Avalon-MM bridges:

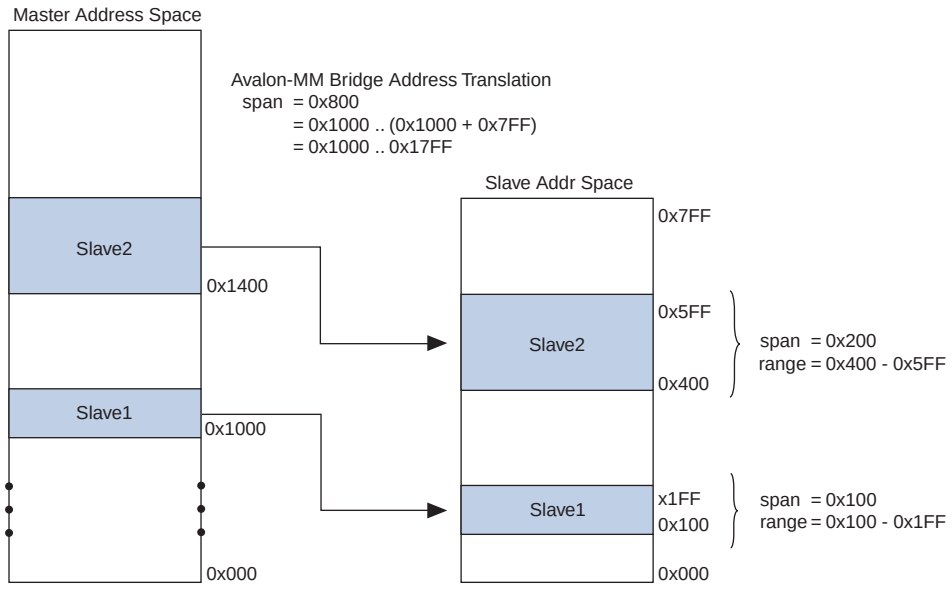
1. The address span of each Avalon-MM slave is rounded up to the nearest power of two.
2. Each Avalon-MM slave connected to a bridge has an address relative to the base address of the bridge. This address must be a multiple of its span. (See Figure 11–6.)

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.

CAUTION

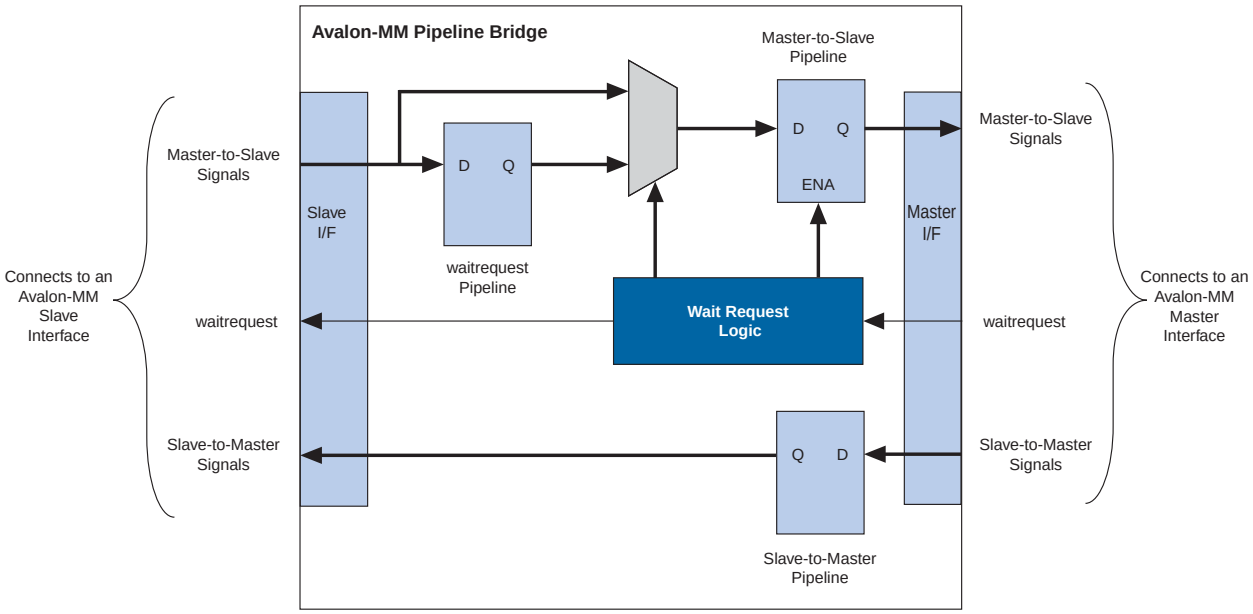
A pipeline bridge with no latency cannot be used with slaves that support pipelined reads. If a slave does not have read latency, you cannot connect it to a bridge with no pipeline stages, because the pipeline bridge slave port has a `readdatavalid` signal. Pipelined read components cannot have zero read latency. Some examples of 0 latency components available in SOPC Builder include the UART, Timer and SPI core. You 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.

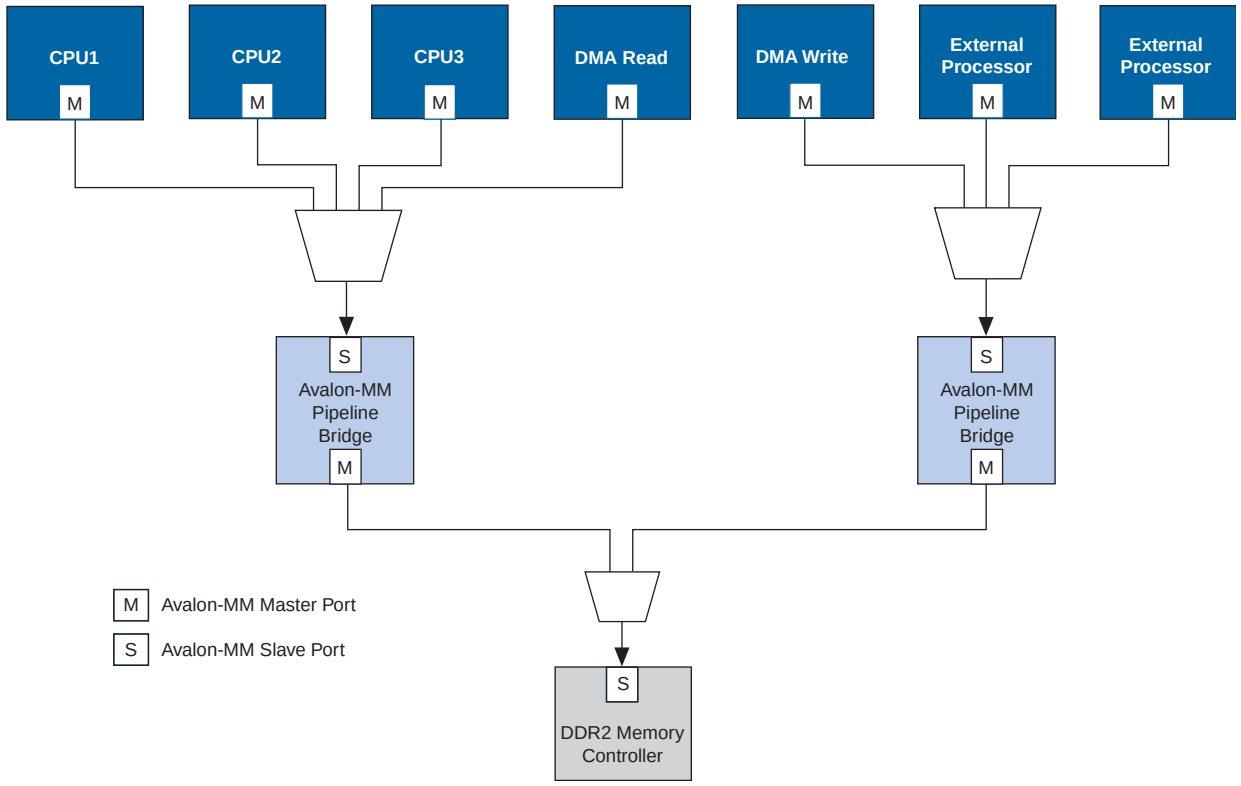
Burst Support

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.

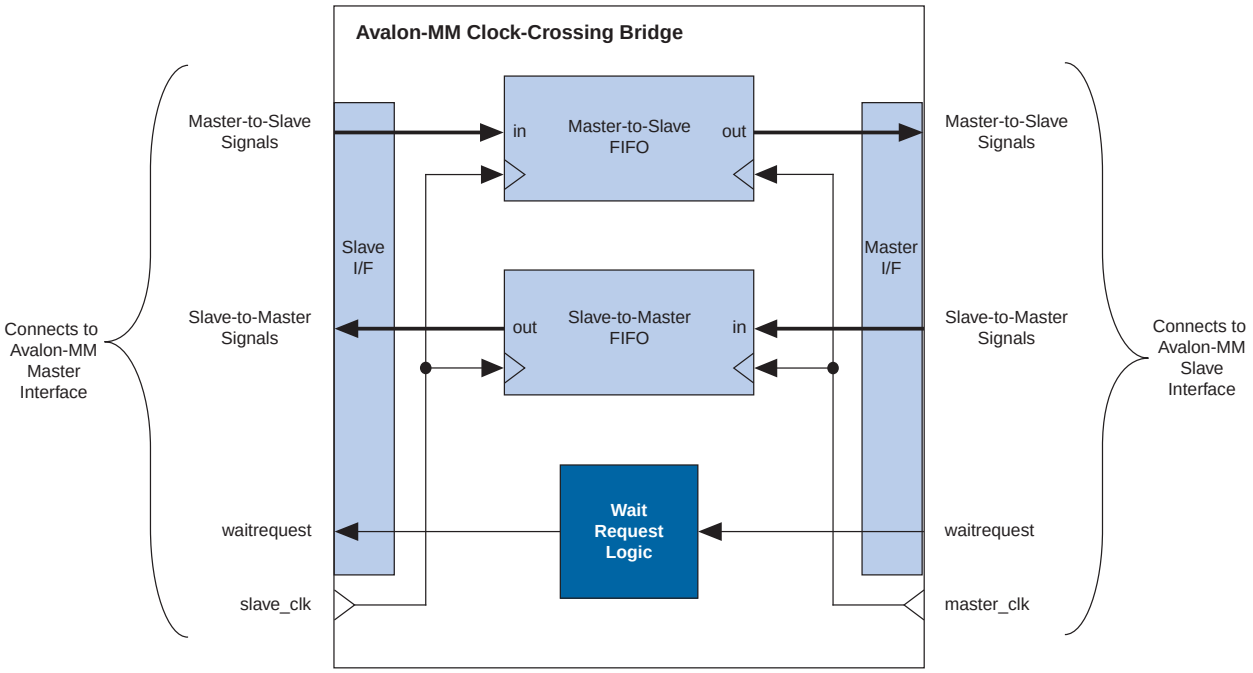
If you rely on the automatically generated clock crossing adapter to connect master and slave ports driven by separate clock inputs, there is a fixed latency penalty associated to each transfer. Each transfer becomes blocking, meaning that while one transfer is underway another cannot begin until the 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.

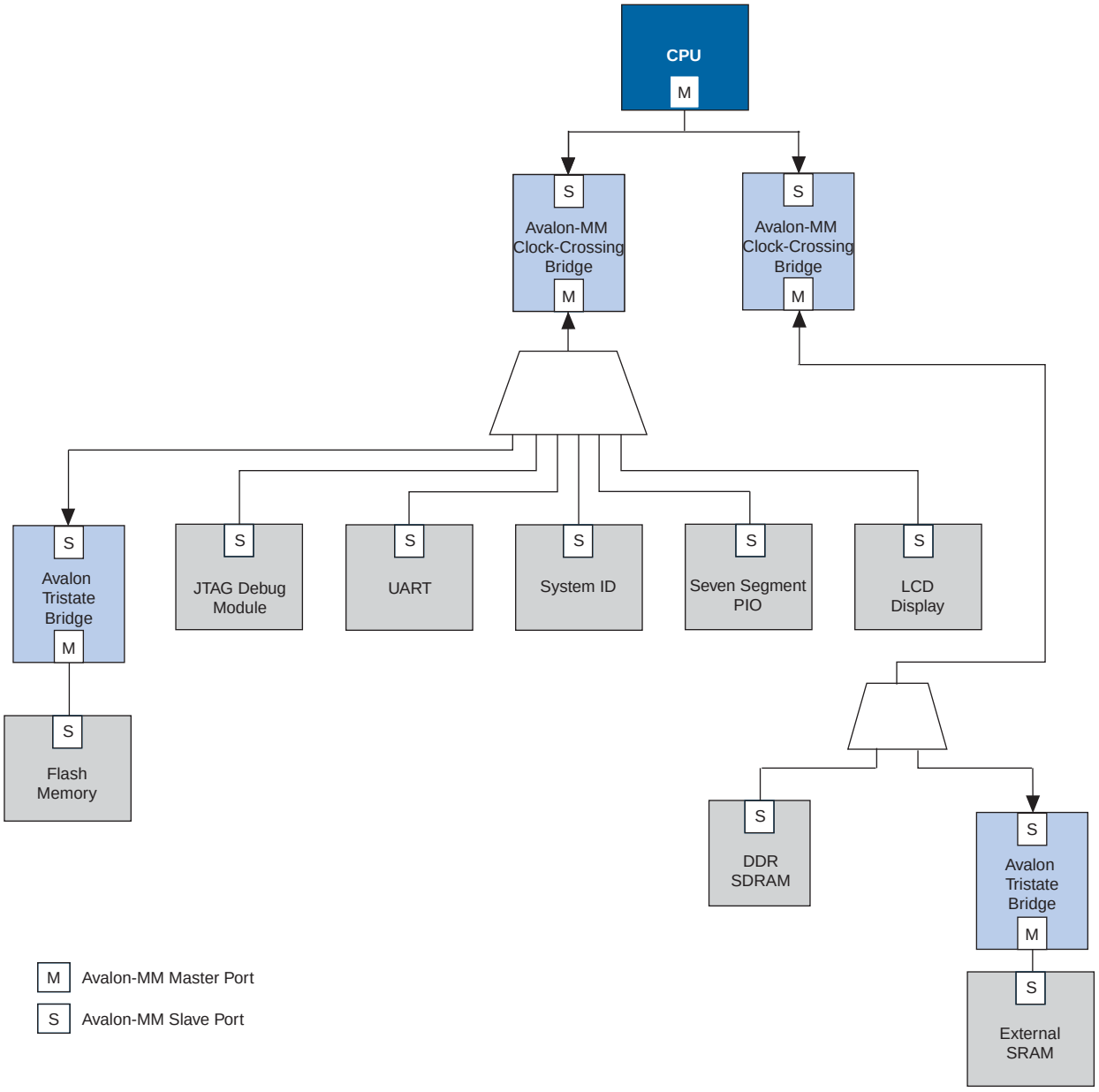
Example 11–1. Minimum Slave-To-Master FIFO Depth

```
= ((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.

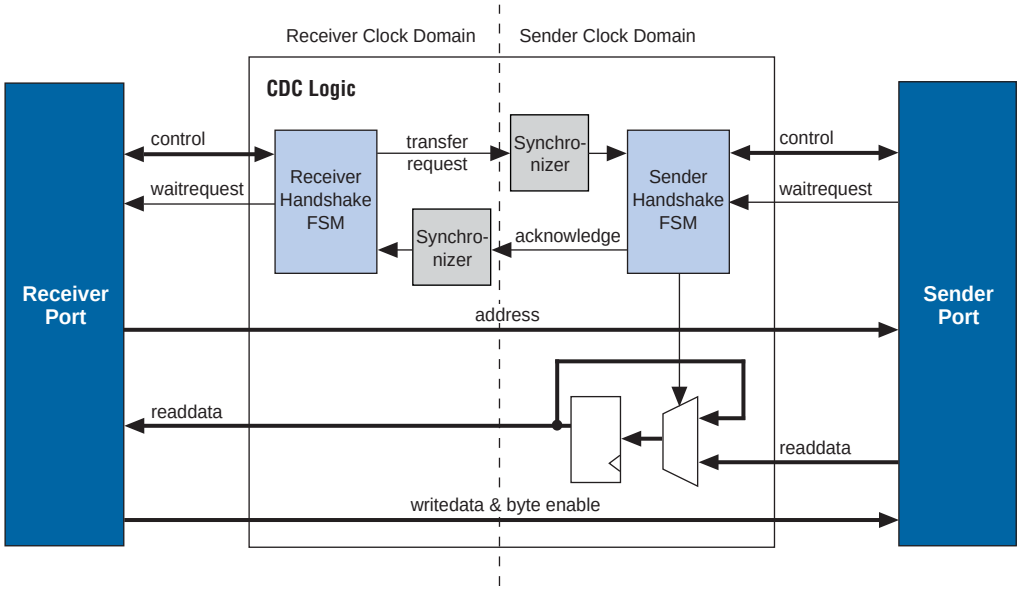
The clock-domain adapters in the system interconnect fabric provide the following benefits that simplify system design efforts:

- 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*.

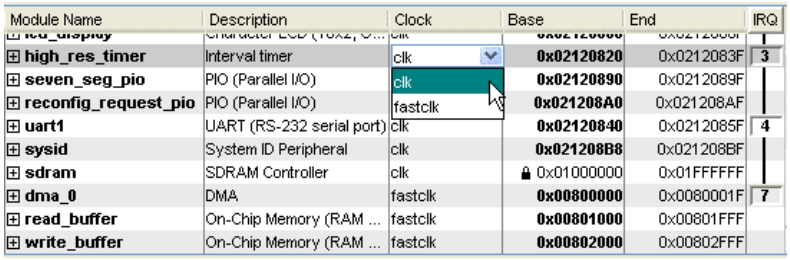
Implementing Multiple Clock Domains in SOPC Builder

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

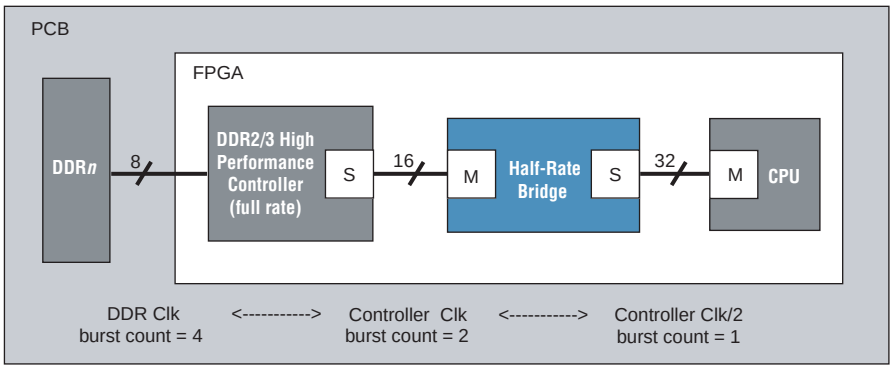


Avalon-MM DDR Memory Half-Rate Bridge

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

- 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 in version 9.1 of the Quartus II software with a data width of 16 bits, an address span of 24 bits.

Table 11–2. Resource Utilization Data for Stratix II and Stratix III Devices

Device Family	Combinational ALUTs	ALMs	Logic Register	Embedded Memory
Stratix II	61	134	153	0
Stratix III	60	138	153	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	Embedded Memory
233	152	33	84	121	0

Functional Description

The Avalon MM DDR Memory Half Rate Bridge works under two constraints:

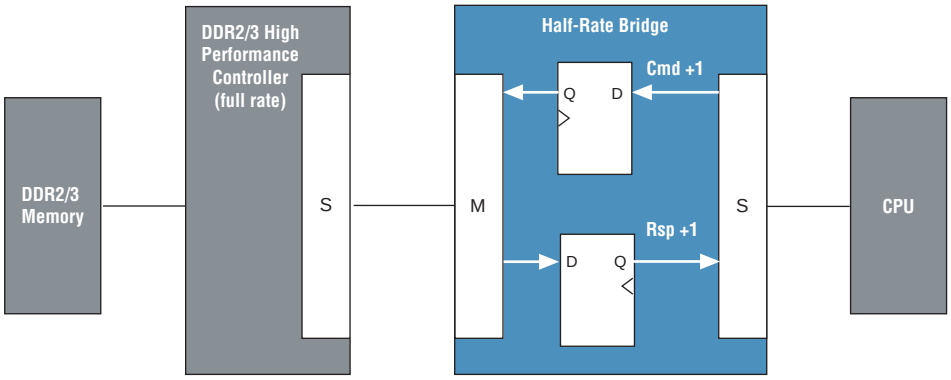
- Its memory-side master has a clock frequency that is synchronous (zero phase shift) to, and twice the frequency of, the CPU-side slave.
- Its memory-side master is half as wide as its CPU-side slave.

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	Avalon-MM Half-Rate Bridge
Arria® GX	Full	Full	Full
Arria II GX	Full	Full	Full
Stratix®	Full	Full	Full
Stratix II	Full	Full	Full
Stratix II GX	Full	Full	Full
Stratix III	Full	Full	Full
Stratix IV	Full	Full	Full
Stratix V	Full	Full	Full
Stratix IV	Full	Full	Full
Cyclone II	Full	Full	Full
Cyclone III	Full	Full	Full
Cyclone IV	Full	Full	Full
Hardcopy®	Full	Full	Full
HardCopy II	Full	Full	Full
HardCopy III	Full	Full	Full
HardCopy IV	Full	Full	Full
MAX®	Full	No support	No support
MAX II	Full	No support	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.

Document Revision History

Table 11–7 shows the revision history for this chapter.

Table 11–7. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	No changes from previous release.
November 2009	9.1.0	■ Added configuration information for resource utilization of half-rate bridge.
March 2009	9.0.0	■ Added information for synchronization when crossing clock domains.
November 2008	8.1.0	■ Clarified connection of clock signals. ■ Added section describing half-rate bridge. ■ Changed page size to 8.5 x 11 inches.
May 2008	8.0.0	■ 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.

Take an online survey to provide feedback about this handbook chapter.

12. Avalon Streaming Interconnect Components

QII54021-10.0.0

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:

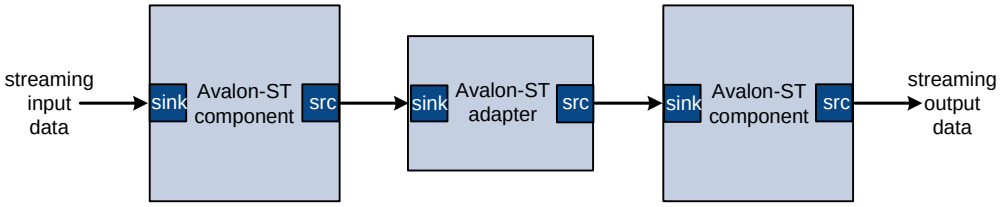
- Adding pipeline stages to adjust the timing of the `ready` signal
- Tying signals that are not used by either the source or sink to 0 or 1

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

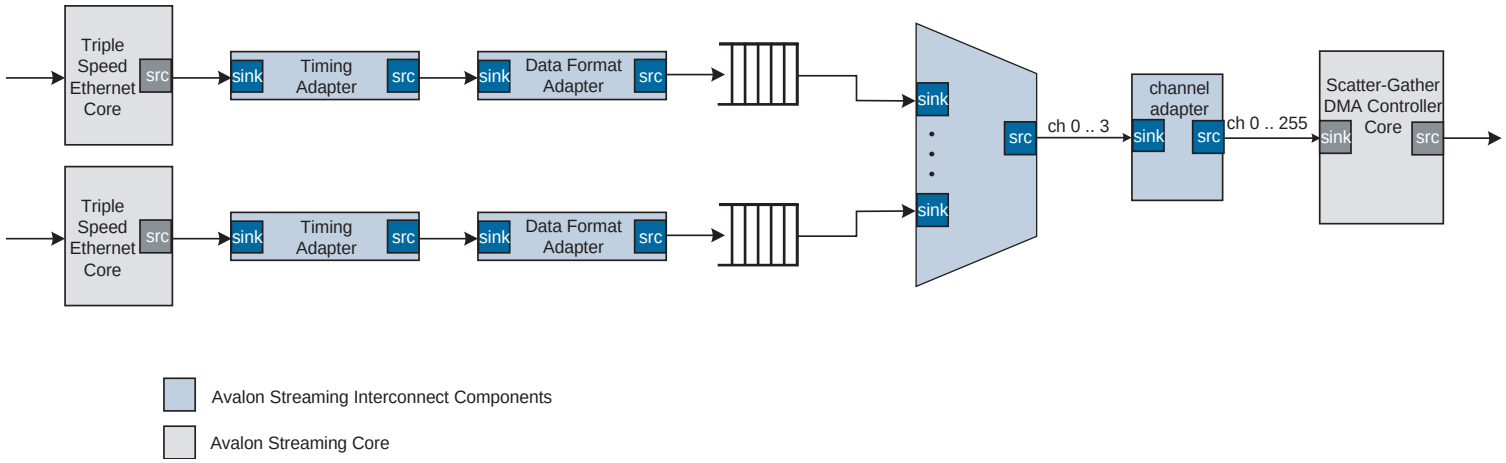


For details about the system interconnect fabric, refer to the *System Interconnect Fabric for Streaming Interfaces* chapter in volume 4 of the *Quartus II Handbook*. For details about the Avalon-ST interface protocol, refer to the *Avalon Interface Specifications*.

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 (Sheet 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 (Sheet 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 (Sheet 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 (Sheet 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 (Sheet 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 commas. 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 (Sheet 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.

Document Revision History

Table 12–10 shows the revision history for this chapter.

Table 12–10. Document Revision History

Date	Version	Changes Made
July 2010	10.0.0	Corrected description of Error Adapter, Error Description definition. Fields should be separated by commas, not semicolons..
November 2009	9.1.0	No changes from previous release.
March 2009	9.0.0	No changes from previous release.
November 2008	8.1.1	Removed private comments
November 2008	8.1.0	■ Added documentation for Avalon-ST error adapter. ■ Reformatted parameter settings in tables. ■ Changed page size to 8.5 x 11 inches.
May 2008	8.0.0	■ 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.

Take an online survey to provide feedback about this handbook chapter.

About this Handbook

This handbook provides comprehensive information about the Altera® Quartus® II design software, version 10.0.

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 10.0 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.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
	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.